White Paper

Is my C/C++ code covered?

This Whitepaper looks at the various applications of the term ‘coverage’ in the software development industry for software written in C and C++. We look at the industry definitions of the terms, applications of the techniques in various software standards and some challenges for measuring coverage you may not have considered. We highlight how modern software testing tools (such as QA Systems, Cantata) can help speed up and monitor your testing progress with coverage.

© Copyright QA Systems GmbH 2022

Contents

1 Introduction **4**

1.1 Four reasons errors are missed 4

2 Coverage concepts and terminology **5**

3 Structural code coverage **6**

3.1 Code coverage gaps and what to do about them 7

3.2 Code coverage metric definitions 8

3.2.1 Entry-point coverage 8

3.2.2 Call-return coverage 9

3.2.3 Statement coverage 9

3.2.4 Decision coverage 9

3.2.5 Modified condition / decision coverage (MC/DC) 9

4 Code coverage by software testing stage **10**

4.1 Code coverage at unit testing 10

4.2 Code coverage at software integration testing 11

4.3 Code coverage at embedded system testing 11

5 Why use code coverage metrics **12**

6 Coverage metrics & safety standards **13**

7 Using code coverage **13**

7.1 Techniques 13

7.2 Tools 14

7.3 How does code coverage affect the tests? 14

7.3.1 Memory 14

7.3.2 Expected behavior 15

7.4 Code coverage special considerations 15

7.4.1 Coverage by contexts 15

7.4.2 Inheritance context coverage 15

7.4.3 State context coverage 16

7.4.4 Thread context coverage 16

7.4.5 Build variant coverage 16

7.5 Coverage metrics in a CI environment 17

8 Coverage in the Cantata tool **17**

8.1 What is Cantata? 17

8.2 Certified coverage for software testing 18

8.2.1 Structural code coverage in Cantata 19

8.2.2 Achieving structural code coverage in Cantata 20

8.2.3 Code Coverage – Team Wide Reporting 21

9 What next? 22

10 References 23

11 QA Systems 24

12 QA Systems Tools 25

12.1 Cantata 25

12.2 Cantata Team Reporting 25

12.3 QA-MISRA 25

Copyright Notice

Subject to any existing rights of other parties, QA Systems GmbH is the owner of the copyright of this document. No part of this document may be copied, reproduced, stored in a retrieval system, disclosed to a third party, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of QA Systems GmbH.

© Copyright QA Systems GmbH 2022

1 Introduction

This paper looks at code coverage techniques that can be employed when testing C and/or C++ code. We look at the code coverage requirements of applicable standards for safety related embedded software and how code coverage metrics can be used effectively throughout your development lifecycle. It also explores some technical challenges to measuring code coverage you may not be familiar with and novel solutions to address them.

The ability to produce reliable technologies that rapidly follow market trends creates a competitive advantage in the digital world. Part of being a technology company is about producing reliable technology at a rapid pace. At the same time, it is not wise to sacrifice code quality just to deliver slightly faster. One of the primary tools for ensuring code quality while maintaining a rapid release schedule is writing good tests. Like any other skill, test writing is best developed through practice and experience. Monitoring development performance and knowing when you have tested enough are very valuable things to consider in any software development project.

Since you are reading this paper about code coverage, it is assumed that you appreciate the importance of a functioning test suite. This paper specifically outlines the code coverage considerations of a successful testing regime. Further information on software testing and other uses of coverage (such as requirements coverage and test coverage) can be found in other QA Systems white papers and publications. Of particular relevance would be "C & C++ Software Testing – Am I Covered?". All our white papers are all available for free from our website qa-systems.com.

1.1 Four reasons errors are missed

Many software developers of systems are surprised when the customer reports an error. We spend countless hours defining requirements, testing code and reviewing the final product. Despite this time investment, how is it that mistakes find their way into the deliverable unnoticed?

Assuming that the customer is reporting valid concerns, we can answer the question with one of the following statements:

- The customer has executed part of the application that has never been tested. Incomplete testing could be deliberate due to time or cost constraints.
- The order or process in which a customer has used the software is different to the use anticipated by the development team or, more likely, the testing team. This actual use was not built into the test suite.
- A combination of inputs was received by the application that were never tested. Software is rarely tested with every possible combination of input value. It is the job of the tester to select a reduced set of typical input conditions that reproduce real world usage. If the assumptions of the tester are wrong, errors slip through.
- The environment in which the software is being used differs between the develop/test teams and the customer. Typical discrepancies can be a different operating system version or hardware. Perhaps the real-world environment was not available to the test team, and it had to be simulated or assumed.

Software is almost never 100% tested. Unfortunately, this even applies to the more rigorously tested safety-critical applications. *[Ref. Hayhurst]* describes, for example, that in the case of a piece of flight control software which processes up to 36 different input variables, if we wanted to test all possible

input combinations and prove that there were no unwanted interrelations between the inputs, we would have to test for 21 years, even if we could create and run 100 test cases per second.

Various measurements of code coverage can be used to set and monitor testing progress and performance to help minimise the occurrence of errors in the field.

2 Coverage concepts and terminology

Throughout the software industry many commonly used terms have no concrete definition. The meaning of technical terms fluctuates depending on who you are talking to. Software testing is an essential activity in the software development and maintenance life cycles. It is a practice often used to decide and improve software quality. When it comes to measuring software testing performance and progress, it is therefore essential that everyone has the same understanding of the measurement terms (metrics) used.

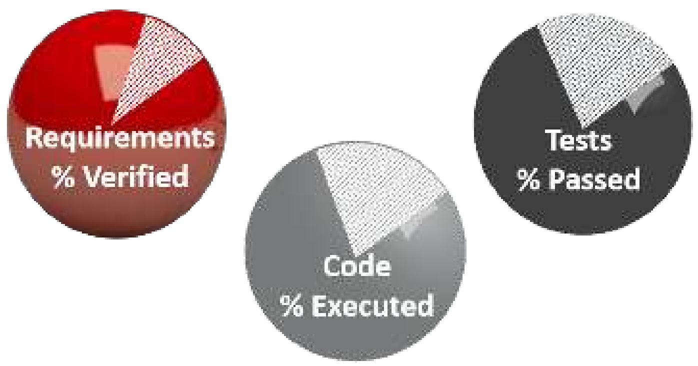
'Coverage' is a broad umbrella term that encompasses a number of useful numerical measures for developers of robust software systems. These measures, when used effectively, can be used both to define quality goals for your end product and track your progress towards achieving them.

In software testing, there are 3 basic types of items to which coverage measurements can be applied

- **Requirements** – various levels of detail defining: functional, safety or non-functional (such as performance or usability) what the software should do, and sometimes what it should not do.
- **Code** – implementation in software (and sometimes hardware or firmware) to meet the requirements.
- **Tests** – a means to verify that the software does what it should do (and sometimes what it should not do – often called robustness tests).

The 3 different uses of the term 'coverage' should not be confused.

- **Requirements Coverage -** measures the proportion of requirements which have been verified by requirements-based tests.
- **Structural Code Coverage -** measures the proportion of the code structures which have been executed by tests.
- **Test Coverage -** measures the proportion of tests which have been run and passed.



3 uses of the term 'Coverage' in software testing, measured as a percentage

Safety critical software standards, such as DO-178C (for airborne systems) and ISO26262 (for road vehicles), recommend use of all three types of coverage

> **Requirements Coverage –** the % verified by requirements-based tests

> **Structural Code Coverage –** the % of executable code exercised by any tests

> **Test Coverage –** the % of all tests run and passing

Correct use of these different coverage concepts can also help software developers outside of the safety critical arena. Appreciation of the terms and their use will help deliver a more reliable and robust application. In this paper we focus on Structural Code Coverage.

3 Structural code coverage

The amount of code that is covered in execution by a single test or collection of tests. For a procedural language like C, you can identify a function of interest, run some test cases on this function, and then measure what proportion (expressed as a percentage) of the code has been executed. The general rule is that the higher the coverage achieved, then the higher the confidence that it has been thoroughly tested.

Measurement of structural coverage of code is an objective means of assessing the thoroughness of testing. There are various industry standard metrics available for measuring structural coverage, these can be gathered easily with support from software tools. Such metrics do not constitute testing techniques, but a measure of the effectiveness of testing techniques.

A coverage metric is expressed in terms of a ratio of the code construct items executed or evaluated at least once, to the total number of code construct items. This is usually expressed as a percentage.

$$\%\text{ Code Coverage} = \frac{\text{Number of items executed at least once}}{\text{Total number of executable items}}$$

There is significant overlap between the benefits of many of the structural code coverage metrics.

Structural code coverage is a measure of the completeness of software testing showing which areas of the source code are exercised in the application during the test. This provides a convenient way to ensure that software is not released with untested code.

3.1 Code coverage gaps and what to do about them

The table below identifies reasons why some code constructs have been found to be untested using structural code coverage and the resulting actions which may be taken.

Reason for unexecuted code	Possible Resulting Actions
Code is 'Dead' (i.e. the construct is dynamically unreachable)	Code can be removed to reduce: • possibility of it becoming inadvertently executable after future code changes • future maintainability costs with clearer code Code can be left in place but commented out to make it non-executable.
Code is 'de-activated' or 'infeasible' (i.e. not supposed to be executed in a particular context, e.g. certain states, threads or system configurations).	An explanation of why the code is de- activated or infeasible to execute in a particular context can be documented (either internally using comments or externally)
Code is 'Untested' and is unnecessary (e.g. code from previous versions / variants of the SUT has been carried into the code base unnecessarily)	Code can be removed to reduce: • possibility of it being used in un- tested scenarios. • future maintainability costs with clearer code Code can be left in place but commented out to make it non-executable.
Code is 'Untested' but is necessary (e.g. code is indirectly related to or 'derived' from a requirement such as code is added for defensive programming, may not have been explicit enough for requirements driven test cases to be created)	Additional test cases can be added to exercise the 'untested' code. Requirements can also be refined to make them more explicit for 'derived' requirements, depending on the need for and granularity of requirements traceability.

3.2 Code coverage metric definitions

Different code coverage metrics measure the execution of different syntax constructs within the code. The most common code coverage metrics are:

- Function / Method Entry Points
- Function / Method Calls (and their Returns)
- Lines (of executable code)
- Statements
- Basic Blocks (of sequential Statements)
- Decision
- Conditions (Boolean operands)
- Relational Operators
- Loops
- Modified Condition / Decision Coverage (MC/DC) both Masking & Unique Cause forms

The fundamental strategic question of how much testing you should do is generally driven by available resources, both time and budget. If you are not required to measure tests against a specific set of structural code coverage metrics by a software safety standard, then the choice of which metrics and which thresholds to set as acceptable, can be determined by your own software quality policy. For more information on the advantages and disadvantages of different code coverage metrics see the QA Systems white paper "Which Code Coverage Metrics to Use".

For all the main software safety standards the required structural code coverage metrics (depending on the safety integrity level of the software under test) are:

- Entry-point Coverage
- Function Call Coverage
- Statement Coverage
- Decision Coverage
- Modified Condition Decision Coverage (MC/DC)

These structural code coverage metrics are explained in more detail below.

3.2.1 Entry-point coverage

Function / Method Entry-Point coverage measures the proportion of functions or methods in the source code which have been executed at least once. It is the easiest metric to achieve 100% code coverage in tests.

3.2.2 Call-return coverage

Call-Return coverage measures the proportion of function or method calls in the source code made and completed at least once. It is the most used coverage metric to measure integration level testing.

3.2.3 Statement coverage

Statement Coverage measures the proportion of executable statements in the source code which have been executed at least once. It can sometimes be referred to by these alternate names: C1, TER1, TER-S coverage. Statements includes all executable (logic rather than declarations) lines of code within a function. Statement coverage does not take into account loops or conditional statements, only statements within an executable line.

It could be considered that statement coverage is a slightly more useful form of Line coverage, in some cases, a single statement can span multiple lines of code or multiple statements can be present on a single line. Line coverage provides a basic measure of code coverage and is often used as a crude coverage measure in some dash boarding software.

3.2.4 Decision coverage

Decision Coverage measures the proportion of decision outcomes in the source code which have been evaluated at least once. It can sometimes be referred to by these alternate names: C2, Branch Coverage, TER2, TER-B coverage. Decisions includes constructs such as ‘if… else…’, ‘switch… case…’ and loops such as ‘while’ and ‘for’. Decision coverage contains Statement coverage but ignores the complexities of conditions within decisions.

3.2.5 Modified condition / decision coverage (MC/DC)

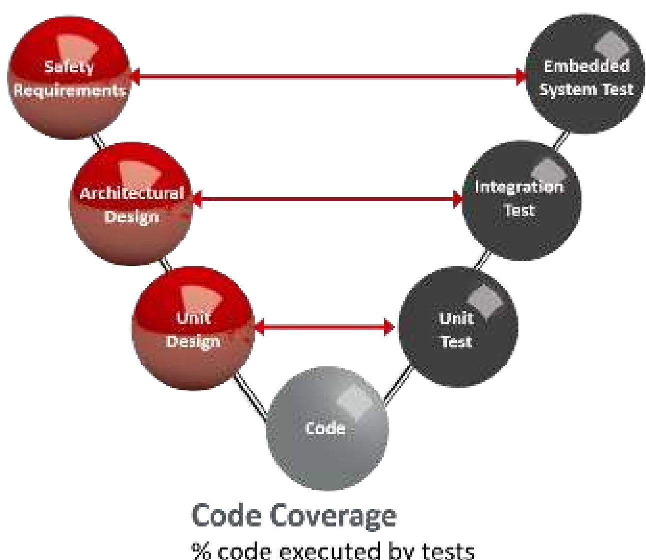
Modified Condition / Decision Coverage measures the proportion of operand Conditions which could independently affect the true/false outcome of the Decision expression that have been effective in doing so at least once. It can sometimes be referred to as a combination of Decision coverage and Boolean Operand Effectiveness coverage. MC/DC coverage demonstrates that every sub-condition can affect the outcome of the decision, independent of the other sub-condition values.

There are two methods for measuring MC/DC coverage: Unique Cause and Masking. The latter was created by Boeing to accommodate the short-circuiting evaluation of true / false expressions in C/C++. NASA has produced a free publication which goes into some depth on this metric and is useful reading. [Ref. Hayhurst].

MC/DC is the hardest metric to achieve 100% code coverage in tests requiring the most test cases.

4 Code coverage by software testing stage

Code coverage can be used as a measure of software test thoroughness (in addition to requirements coverage and test coverage) at all stages of testing as illustrated below.



V-model of stages of testing mapped to the corresponding levels of requirements / design

One of the most common causes of applications being deployed with bugs, is that programs experience unpredictable, and therefore untested, input combinations when in the field. These types of errors can be discovered more readily by applying structural code coverage as the backstop to requirements coverage throughout the unit testing, integration testing and system testing stages on host native platforms or the final embedded target architecture.

It is almost impossible to obtain 100% code coverage during system level tests alone. Typically, during this stage of testing, you can reach 70% Decision code coverage. The remaining 30% code coverage is only achievable when software is broken down into more manageable size and complexity through unit and integration testing.

As structural code coverage can be measured at each testing stage, the coverage obtained from each stage can be combined to create an aggregate view of how much of the code is executed by the various tests. An efficient code coverage strategy can therefore seek to use some test stages to plug gaps in code coverage achieved at other test stages. A common practice (especially for legacy applications) is to apply code coverage to existing system level tests, and supplement these with unit or integration level tests to plug gaps at edge condition where the code is difficult or uneconomic execute during system tests. These edge conditions often arise because of defensive programming in the code or the difficulty of simulating software and hardware error conditions at system tests.

4.1 Code coverage at unit testing

The first potential stage of testing is at the unit level, because individual units of code can be tested before other units are even implemented in code. Unit tests aim to verify the correct behavior of the smallest viable compliable 'unit' of code in isolation from the rest of the system. This unit under test is usually a single source file of functions, a single function, or a class. However, some unit tests maybe larger in scale and are more akin to small integration tests. Such unit tests are often referred to as 'module' or 'component' tests.

It is at this early testing stage achieving the highest levels of structural code coverage will be easiest, because techniques such as isolation white-box testing allow the tester greater control over unit under test to drive all potential behavior implemented in the code. It is also the stage which is the most likely discover insufficiently defined low-level functional requirements or designs.

Unit testing requires the use of test code in the form of drivers and simulations to isolate specific units from the rest of the application and to activate or drive these code units via test cases. Unit tests provide much greater control over the code being tested and are therefore easier achieve 100% code coverage (especially for harder to achieve coverage metrics), either on their own or combined with other stages of testing.

Unit test frameworks often directly provide or can be easily integrated with code coverage capabilities. Where fully integrated, a target requirement for the percentage of code coverage achieved by the test(s) can be set as a pass/fail check criterion in the unit test framework. This also has the advantage of being fully automated for regression runs of unit tests, without requiring manual checking or review of code coverage results (unless there is a failure to achieve the target).

4.2 Code coverage at software integration testing

Integration testing, when performed after unit testing, is focused on the interactions between units rather than their internal functionality. For this stage of testing various software safety standards (e.g. ISO 26262 for road vehicle software) require code coverage of function calls between units. Function entry-point coverage is insufficient for meeting this requirement, so the Call-Return coverage metric is used instead.

Integration testing requires the use of test code in the form of drivers and simulations to isolate the integrated software from the rest of the application and to activate or drive this integrated code via test cases. It is common to employ unit test frameworks to perform integration testing on tightly coupled units. It is also often more efficient to obtain code coverage of internal functionality of units during integration testing when that functionality relies on interfaces. This is because linking in more external software (such as library, operating system, or middleware code) to the test scope beyond units is cheaper and more realistic than pure isolation testing of the units at unit testing.

However, whether the integrated code under test is executed via a unit test framework or via an external test driver, code coverage can be measured checked against percentage thresholds and reported independently. In most cases the integrated source code under test is instrumented (logging points added) with the appropriate code coverage and built as normal. When executed the coverage tool will export in various ways the coverage results obtained during or at the end of the integration test.

4.3 Code coverage at embedded system testing

For system testing on an embedded environment, unit test frameworks are not usually appropriate and external drivers are used. The general approach for code coverage at embedded system testing is like that for software integration testing with external test drivers. However, the embedded environment may impose some constraints on the use of code coverage. Source code coverage instrumentation will increase the size of the software and may even affect the run-time functional behavior of hard rea-time systems.

For that reason, such tests can be run both with and without code coverage to ensure that the instrumentation does not affect the behavior.

Another consideration is simply the available memory on the embedded target platform. Where memory is constrained, code coverage tools need to provide suitable mechanisms to address or work around these limitations.

5 Why use code coverage metrics

Code coverage metrics used in testing can not only monitor the thoroughness of testing they can also guide test case creation to where something is missing or not verified.

There are some key criteria to consider when writing tests:

- Focus testing on parts of the application which are more critical, the parts where bugs are most likely to lead to a bad outcome for customers.
- Apply more thorough tests to parts of the code which are most likely to contain bugs.
- Using techniques such as equivalence class analysis (where test input values should have the equivalent effect on the code) avoids redundant duplication of test cases for code coverage. Where available make use of tool provided automatic test case optimization for coverage
- Define criteria for when code is tested enough. Testing cannot be exhaustive, so knowing when to stop testing some parts of the code, prevents ignoring other parts of the code.

Setting project goals around defined metrics such as coverage, has several benefits to project success.

Optimise the use of resources

There are never enough resources to do everything, so setting coverage goals can help you to prioritise. By allocating most time and budget to test what is most important you can help focus testing efforts. If you want to better manage your time on testing, a simple solution is to stop doing what doesn't need to be done.

Add clarity to project meetings

Knowing what you are trying to achieve means that you can tackle the question: "does this activity get me closer to my goal?" Setting code coverage goals enables you to clarify with other developers and testers what you are trying to do, and therefore what they need to do to contribute or support.

Easier measurement of project status

Code coverage goals allow you to measure how effectively you are moving towards completion.

An important consideration is knowing when to stop testing. For those working towards standards, the coverage goals will be mandated. For others an important first step is defining the targets of coverage to aim for.

Progress towards a code coverage goal does not follow a linear progression. The graph in figure 5 illustrates this point. In the early stages of the project code coverage metrics tend to increase in percentage achieved quickly. As time progresses and you are left with more difficult to test scenarios so increasing levels of code coverage becomes harder.

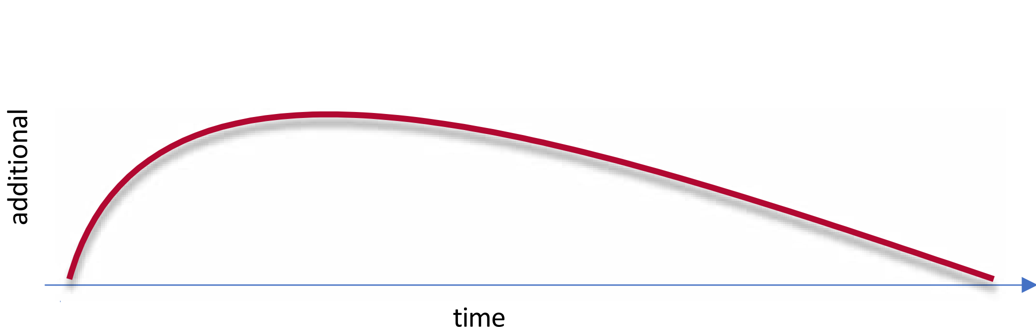


Figure 5 – An illustration of diminishing marginal return on code coverage of tests

6 Coverage metrics & safety standards

If you are working in a safety critical industry, it is likely that you will be working toward achieving certification in the relevant international software standard. The standard and integrity level within it that you are working towards, will determine the code coverage metrics and minimum target threshold percentage that you need to demonstrate in your project.

Figure 6 (below) sets out the minimum integrity level within each safety standard that dictates achieving 100% structural code coverage metrics.

Sector

Code Coverage Metric	ISO 26262	DO-178B/C	IEC 62304	IEC 61508	EN 50128	IEC 60880
100% Function Entry-Point	A	C	->	1	1	A
100% Function calls	B	C	->	1	1	A
100% Statements	B	C	->	2	2	A
100% Decisions	C	B	->	3	3	A
100% Conditions (MC/DC)	D	A	->	3	4	A
100% Control & Data Flow	-	A	-	-	3	-

Figure 6 – A summary of coverage across various safety critical standards.

Note that the IEC 62304 standard "Medical Device Software – Software Life-cycle Processes" does not explicitly state which structural code coverage metrics are appropriate for the testing of software in different Class devices, but instead refers to the IEC 61508 standard.

7 Using code coverage

7.1 Techniques

Measuring code coverage requires the recording of code as it is executed during tests. This recording can either be done on source code or on the compiled object code. The most common technique used is source code coverage, as this is usually easier to map to the source code for analysing the results.

The process of recording the code coverage information involves adding logging points into the source or object code to count execution the various code syntax constructs. This logging process is referred to as instrumentation. As the code is executed under tests, measuring the code coverage can be gathered at various points and then reported. Reporting can be done dynamically during tests or at the end of test runs as required. It is common for reporting to record code coverage by each test case and test run as well as calculating percentages all those syntax constructs which were executed by the tests. Where obtaining a minimum percentage code coverage for a metric is required (e.g., by a software safety standard), it is also helpful to have the coverage data achieved during tests checked against the minimum target percentage required.

7.2 Tools

Due to the complexities and repetitive nature of adding the code coverage instrumentation, reporting and checking the achieved coverage data, it is normal practice to make use of an automated tool. It is helpful if the code coverage tool is integrated into the tools for creating and running the tests. However, where custom test frameworks or manual tests are used, code coverage tools can be used to instrument and record code coverage data for anything test driver which executes the code.

Where measuring and reporting code coverage is required by software safety standards, the code coverage tool used will normally be required to have been certified or qualified as suitable for use on safety critical software testing under that standard. Use of tools which have not been certified or qualified can lead to problems and delays in proving the testing has been undertaken in accordance with the standard and therefore risk the compliance of the delivered software.

Where software is not subject to safety standards compliance requirements, it can be re-assuring to use code coverage tools which are independently certified as suitable for use in safety standards.

7.3 How does code coverage affect the tests?

Instrumenting the source or object code to measure code coverage makes the code size bigger. There are two ways in which the making the code size bigger may affect the tests. The first is that the bigger code requires more memory to execute. The second is that bigger code running more slowly may affect the expected behavior of the code under test. In both cases the amount of instrumentation and therefore the scale of the affect is principally determined by which code syntax constructs are measured for code coverage. The more complex the code syntax constructs, the greater the affect.

7.3.1 Memory

The RAM used in a test for code coverage can be most relevant for testers executing their tests on embedded target environments with limited available memory. The amount of data recorded varies by the code coverage metric. Code coverage tools can provide an estimate of the additional memory requirement for the code under test and selected coverage metric. For information on memory requirements for code coverage with QA Systems Cantata tool, see the Cantata Technical Note – Low Memory Targets.

7.3.2 Expected behavior

Instrumenting the source or object code may affect the speed at which it executes. The extra logging and data gathering code added, may have an impact on the execution flow of the code under test, especially if the program logic is hard real-time and execution behavior may change with larger code executing more slowly. For this reason, it is advised by most of the software safety standards that the same tests be run with and without code coverage, to check that the expected functional and on-functional behavior of the code under test is unaffected by measuring the code coverage.

7.4 Code coverage special considerations

In this section we explore some special challenges for code coverage, which you may not have considered.

7.4.1 Coverage by contexts

Traditional code coverage measures execution of source code constructs but does not take account of the context in which that code object executes. The same source code may behave differently depending on this object context. Examples are:

- Polymorphic base class code in multiple inheritances
- State machine code in different states
- Multi-threaded code in different threads

Without this contextual information it is not possible to identify whether the same code constructs are executed in the different contexts, which may lead to incomplete testing.

7.4.2 Inheritance context coverage

When testing derived classes, it is possible to gain a misleading impression of how well an underlying base class has been tested because traditional structural code coverage achieved on the base class can accumulate across multiple different inherited contexts.

Figure 10 below shows how coverage achieved on two derived class can give a misleading impression of coverage on the common base class. An example might be the changed behaviour of an inherited member function if it calls a virtual member function which has been overridden in the derived class.

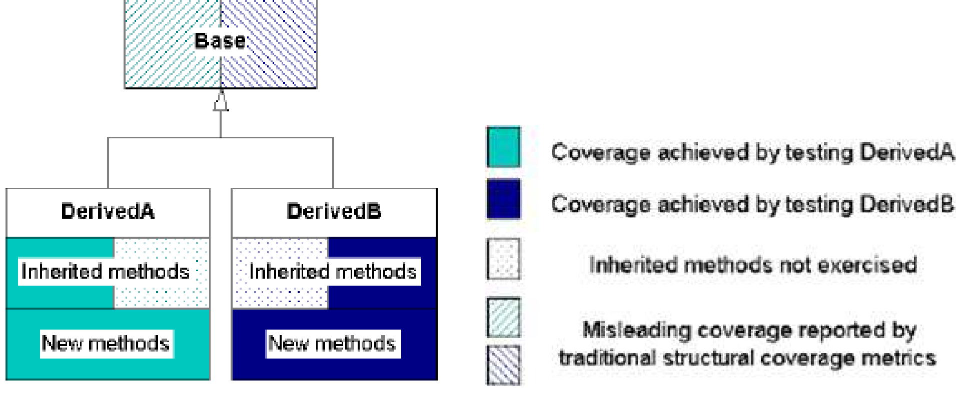


Figure 10 – Inheritance context coverage

The achievement of 100% code coverage of base class within each derived context has the additional benefit of automatically testing the design for conformance to the Liskov Substitution Principle (LSP), i.e., that the derived class is a correct implementation of a base class. The LSP is an important object-oriented design principle which helps ensure that inheritance hierarchies are well-defined.

7.4.3 State context coverage

When testing code in a finite state machine, the behaviour of functions may depend on the current state of the machine. It is possible to gain a misleading impression of how thoroughly state machine source code has been tested, because traditional structural coverage achieved on the source code, can accumulate across multiple states.

A state machine will exist in a current state. When an event occurs, the state machine may take an action and may make a transition to a new state. Achieving State Coverage is a common way to demonstrate that each state in a finite-state machine been reached and executed.

An example of a software safety standard requiring state coverage is the General Motors standard CG2999 "Component Software Validation and Verification Requirements" ™. Section 3.2.2.2.2. v. of that standard requires evidence of: *"State coverage: Each state in a finite-state machine been reached and executed".*

As the state context of a state machine may be implicitly or explicitly defined in the code, a code coverage tool will require a state definition to record the current state of the code as the code executes under test.

One way of defining this state context explicitly is to include a private method or file static function which returns the value of the current state. However, this has the disadvantage that additional code is added to the source code just to make it testable. A better way is to read the value of a local static or private variable. A further alternative is to include a context definition function in the test framework script which can also be more complex and can deal with implicit definition of the state context.

That state context data however defined, can then be used to record and report coverage while the state machine code executes within each state.

7.4.4 Thread context coverage

When testing multi-threaded code, the behavior of code can exhibit exactly the same characteristics as state machine code. The same approaches as above as for state machine code, can therefore be taken with multi-threaded code, to define threads and measure thread context coverage.

7.4.5 Build variant coverage

When testing the same source code built with different variants using pre-compile defines (#defines) the behavior of the compiled functions may depend on the build variant of the source code. Build Variant Coverage can improve C/C++ coverage data for source code executed over more than one variant.

Aggregating data for multiple build variants allows high levels of coverage to be reached. A report can also be generated with aggregate coverage data across all variants, which is suitable as certification evidence for all build variants of the source code.

```
=============================================================================
= Cantata Test Harness v9.0                                                 =
= (c) 2019 QA Systems GmbH                                                  =
=---------------------------------------------------------------------------=
= Cantata Build Variant Report                                              =
=============================================================================

Input Preprocessor Log Files:
/home/liam/test_cpp_preproc/tmptest/expected_test_file_1.cpl

------------------------ File: 1. test_file_1.c
Preprocessor Directives:
test_file_1.c (7) else
test_file_1.c (11) ifdef DEF_2 >> NOT INCLUDED
test_file_1.c (13) else
test_file_1.c (17) ifndef DEF_4
test_file_1.c (19) else >> NOT INCLUDED
...
------------------------ File: 2. test_file_2.c
Preprocessor Directives:
test_file_2.c (5) if DEF_0 >> NOT INCLUDED

=============================================================================
= File                                 Included  Not Included      RESULT =
=---------------------------------------------------------------------------=
= 1. test_file_1.c                           12            26   >> FAIL =
= 2. test_file_2.c                            0             1   >> FAIL =
=---------------------------------------------------------------------------=
= TOTALS                                     12            27   >> FAIL =
=============================================================================
```

```
#ifdef ALARM_PRESENT
    if (alarm_needed) {
        sound_alarm();
    }
#else
    write_status(status);
#endif
    return status;
}
```

Figure 11 – Example reporting on build variant coverage

7.5 Coverage metrics in a CI environment

A useful way of using code coverage is by adding an automated test stage to your build system. With a Continuous Integration (CI) environment, such as Jenkins, it is possible to automate building, executing and reporting on a suite of regression tests for any code check-in.

By setting a code coverage percentage threshold for each metric defined in your test, you can cause the build to fail when the achieved level of code coverage does not reach the required % target.

Further information on testing in a Continuous Integration or DevOps environment can be found at qa-systems.com. (https://www.qa-systems.com/resources/)

8 Coverage in the Cantata tool

8.1 What is Cantata?

Cantata is the safety certified unit and integration testing tool from QA Systems, enabling developers to verify standard compliant or business critical code on host native and embedded target platforms. It is therefore much more than just a tool for measuring code coverage (which it does), Cantata helps you achieve the desired levels of code coverage.

Cantata is a complete test development environment for C & C++ code. Tests can be created, traced to requirements for requirements coverage, executed with integrated or standalone code coverage, comprehensively analysed and results reported for certification compliance.

Built on Eclipse, Cantata integrates easily with developer desktop compilers and embedded target platforms.

8.2 Certified coverage for software testing

Cantata has been independently certified by SGS-TÜV SAAR GmbH as usable when developing safety related software, up to the highest safety integrity levels, for the following standards:

- **ISO 26262** (Road vehicles – Functional safety),
- **IEC 60880** (Nuclear Power),
- **IEC 62304** (Medical Device software – software life cycle processes),
- **IEC 61508** (Functional Safety of Electrical/ Electronic/Programmable Electronic Safety Related Systems),
- **EN 50128** (Railway Applications – Communication, signalling and processing systems)

Cantata has also been successfully qualified many times up to Software Level A for the avionics standards:

- **DO-178B/C** (Software Considerations in Airborne Systems and Equipment Certification).

8.2.1 Structural code coverage in Cantata

Cantata uses source code coverage instrumentation on a temporary copy of the source code, so your production code is never modified just to measure it. Code coverage is integrated with Cantata unit and integration tests. It can also be used in standalone mode to measure the coverage achieved whatever the test driver (e.g., a manual system test). With code coverage integrated into Cantata tests the process works as below:

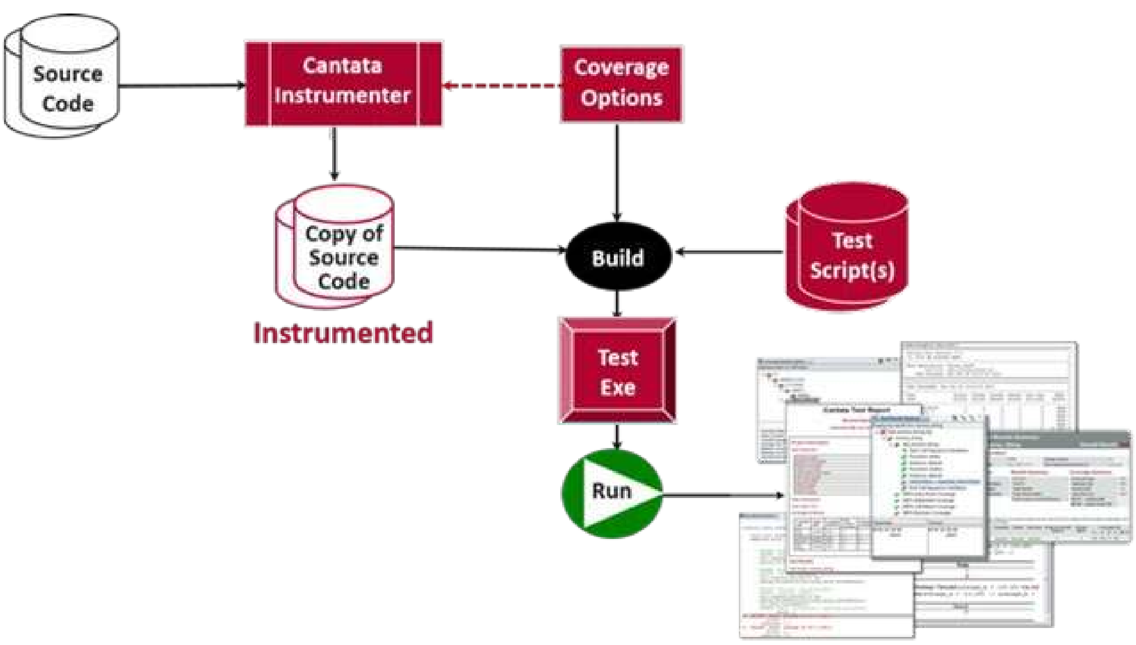


Figure 14 – Combined instrumentation and test process

The key code coverage features of Cantata are:

- Simplifies safety standards and integrity level compliance with code coverage rulesets
- Measures all the structural code coverage metrics in this paper
- Measures context code coverage
- Measures build variant coverage
- Measures code coverage on whatever test drives the code (e.g., manual system tests)
- Integrates with unit & integration tests (with checks on % coverage targets)
- Records code coverage by each test case and test run
- Aggregates code coverage over tests
- Displays code coverage in tree views drilled down to syntax within lines of code
- Filters all code coverage data for comprehensive diagnostics by tests and metrics
- Optimises Cantata test cases automatically to obtain a minimum set to achieve coverage
- Reports code coverage for management dashboards and certification evidence

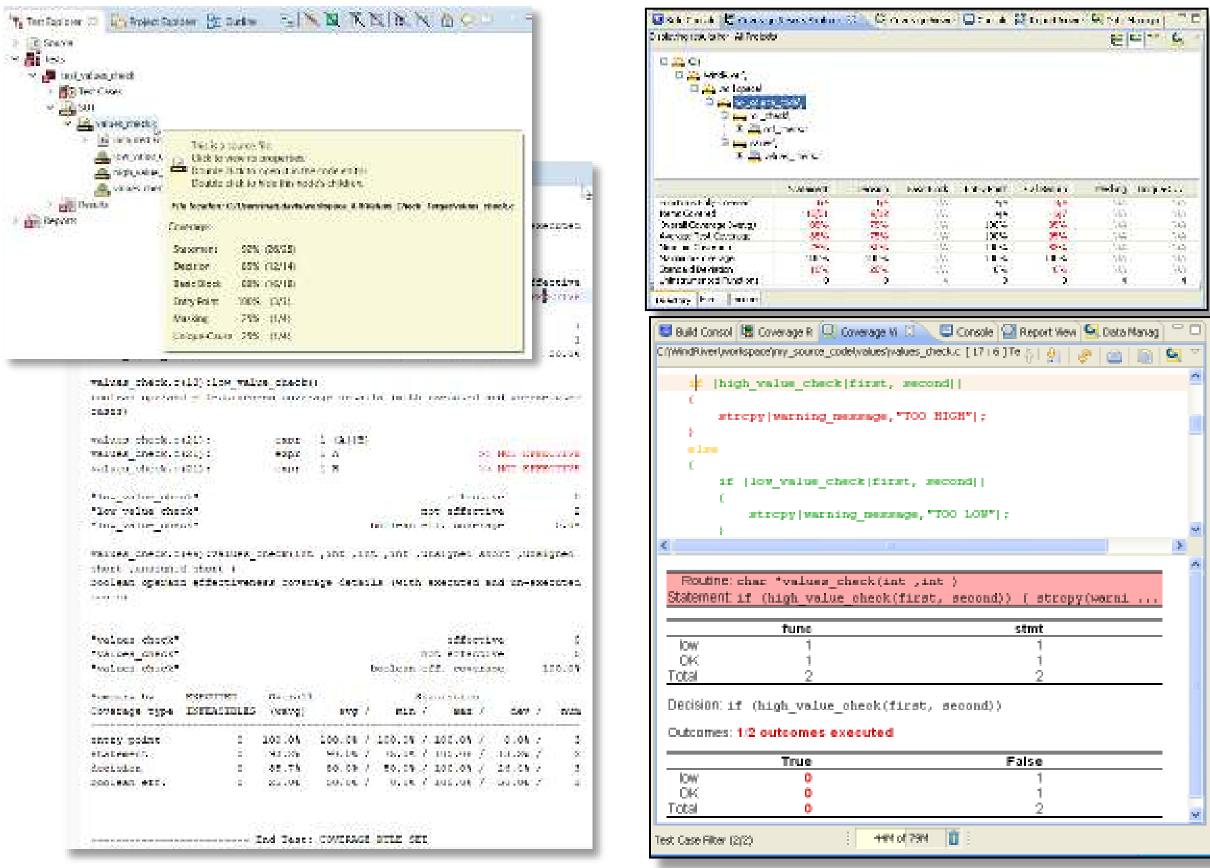


Figure 15 – An example set of code coverage views in Cantata

For more information on Cantata code coverage capabilities, see the Cantata code coverage webpage.

8.2.2 Achieving structural code coverage in Cantata

While measuring code coverage will tell you how thoroughly software tests have exercised the code, it will not help in achieving the desired target level of code coverage. That is where efficient or even automatic test case generation techniques can really help.

The Cantata unit and integration test framework provides a high degree of test generation to help testers reach their code coverage targets. The easiest way to achieve 100% code coverage for the following metrics is with Cantata AutoTest:

- 100% function Entry-points
- 100% Statements
- 100% Decisions
- 100% Unique Cause MC/DC

An algorithm creates test case vectors which exercise all required code paths, using the Cantata powerful white-box capabilities to set data, parameters and control function call interfaces. The test

vectors drive the code, and check the parameters passed between functions, values of accessible global data, order of calls and return values.
Cantata AutoTest generated cases, are editable in the same ways as user generated cases, and each test case has a description of what path through the code it was created to exercise, making them easy to maintain and link to requirements with Cantata Trace for requirements coverage.

Cantata AutoTest makes it easy to:

> Configure automatic test generation
> Identify code testability issues
> Generate tests with full code coverage
> Plug 'edge case' gaps in coverage from existing tests
> Create a thorough safety net of baseline regression tests
> Link generated test cases to requirements for requirements

For more information on Cantata AutoTest, see the Cantata AutoTest webpage.

8.2.3 Code Coverage – Team Wide Reporting

Cantata reports code coverage in various formats suitable to the needs of managers, engineers and QA / compliance functions. Cantata provides filterable drill-down diagnostics and safety standard certification ready test results evidence.

The Cantata Team Reporting add-on, additionally stores tests pushed from Cantata client desktops or build servers onto a centralised server with data accessible over a web interface and a REST API for integration into other test management tools. Cantata Team Reporting provides easy monitoring of current code coverage and test status, historical data and trends over multiple codebases.

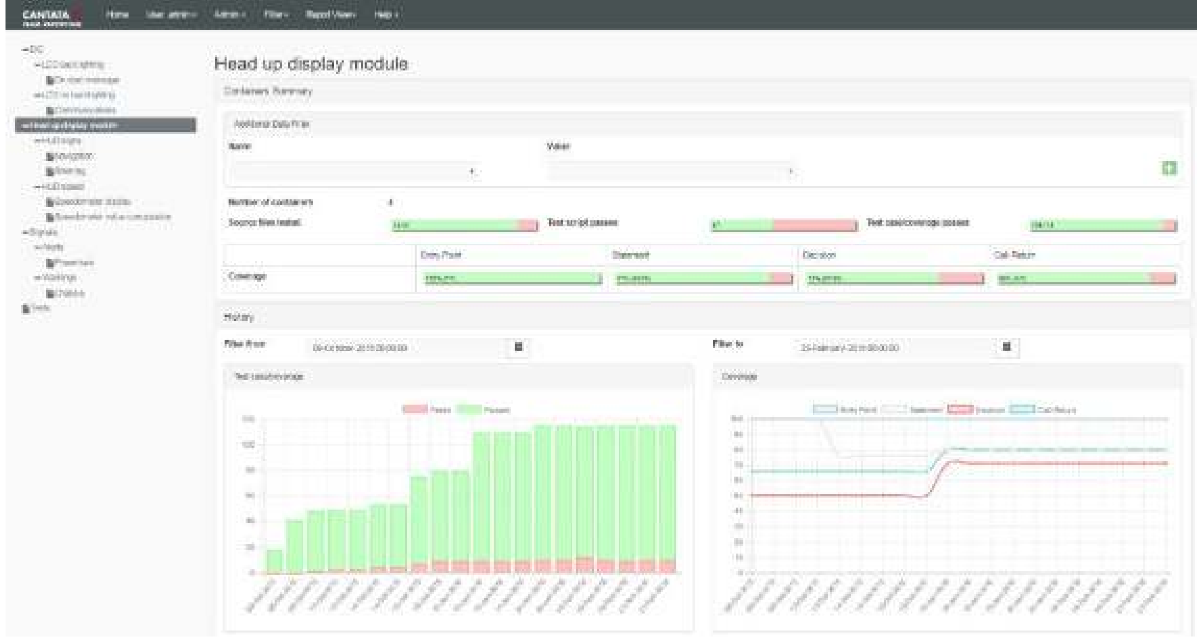


Figure 16 – An example management dashboard in Cantata Team Reporting

For more information on the Cantata test coverage and test status management dashboard capabilities, see the Cantata Team Reporting webpage.

9 What next?

This paper has presented some arguments and explanations as to why and how code coverage can be used to best guide software testing.

Cantata offers a comprehensive software testing tool which supports measuring and achieving requirements coverage, structural code coverage and test coverage. Cantata is available from QA Systems and its international network of authorised resellers. Further information on the Cantata product can be found at the QA Systems website: **https://www.qa-systems.com/tools/Cantata**

There you can request a demonstration, contact our software quality experts and request a free trial of Cantata & Cantata Team Reporting.

If you want to know the answer to “Is my C/C++ code covered?”, we look forward to hearing from you.

10 References

[1 Hayhurst] K. Hayhurst, D. Veerhusen, J. Chilenski, L. Rierson: »A practical Tutorial on Modified Condition/Decision Coverage«, NASA/TM-2001-210876.

[2] Which Coverage Metrics to use
A paper produced by QA Systems http://www.qa-systems.com/

[3] Cantata Datasheet
A document produced by QA Systems which highlights the functionality of Cantata. http://www.qa-systems.com/cantata.html

[4] MC/DC
A code coverage metric Modified Decision Condition Coverage (MD/DC) used at the highest Safety integrity Level of various standards level of integrity. This metric is supported in Cantata as Boolean operand effectiveness. http://en.wikipedia.org/wiki/Modified_condition/decision_coverage

[5] Cantata Tool Certification
The current certification is for Cantata 6.2 Build ID Release 6_2.14. The SGS-TÜV Saar GmbH certificate number reference is FS/71/220/14/0043 issued on 11 August 2014. http://www.qa-systems.com/cantata.html (under tab Tool Certification)

[6] SGS-TÜV Saar GmbH
SGS-TÜV GmbH, are an independent third party certification body for functional safety, accredited by Deutsche Akkreditierungsstelle GmbH (DAkkS) [accreditation ID: D-PL-12088-01-01]. http://www.sgs-tuev-saar.com/en.html

[7] Embedded Software Testing Practices
A paper produced by QA Systems http://www.qa-systems.com/

11 QA Systems

QA Systems tools automate unit testing, code coverage, integration testing and static analysis to optimise safety and business critical embedded software and accelerate standards compliance. Quality is the driving force behind QA Systems. With over 20 years of experience, our tools and services enable organizations worldwide to develop tested high-quality software which meets the stringent demands of industry safety standards.

All tools are independently certified by SGS TüV for use at the highest integrity level of safety related software development for all major safety standards (ISO 26262, IEC 61508, IEC 62304, EN 50128, and IEC 60880), and qualifiable for standards such as DO-178B/C.

Founded in 1996 by CEO and racing driver, Andreas Sczepansky, QA Systems operates across Europe and through a global reseller network. QA Systems has over 350 blue-chip customers, across all safety related and business critical industries. In addition to our tools, the QA Systems Academy shares our know-how and expertise with engineers from around the world.

12 QA Systems Tools

12.1 Cantata

Cantata is a unit and integration software testing tool, enabling developers to verify standard compliant or business critical C/C++ code on embedded target and host native platforms. Cantata is integrated with an extensive set of embedded development toolchains, from cross-compilers to requirements management and continuous integration tools. The Eclipse GUI, tight tool integrations, highly automated C/C++ test cases generation, all make Cantata easy to use. Cantata has been independently certified by SGS-TÜV SAAR GmbH for use at the highest integrity levels for safety-related standards including ISO 26262, IEC 61508, IEC 62304, EN 50128, and IEC 60880. It is also end user qualifiable for standards such as DO-178B/C.

12.2 Cantata Team Reporting

Cantata Team Reporting is an optional add-on to Cantata which provides a web-based management dashboard showing current testing status, historical data and trends over time. All test data is stored on a centralised server enabling teams to work more effectively together and managers to monitor test status and progress. Test and code coverage results are aggregated, and additional data can differentiate tests across multiple system or product variants for the same Cantata tests. Cantata Team Reporting is integrated with continuous integration and other ALM tools.

12.3 QA-MISRA

QA-MISRA is a static analysis tool, enabling developers to comply with C/C++ coding standards for functional safety (MISRA, AUTOSAR etc.) and security (CERT and CWE etc.). It also provides insights through metrics and visualisations into source code quality. QA-MISRA has an interactive GUI, full Command Line Interface and integrations with IDEs and CI frameworks, a very fast analysis speed and open format reports. QA-MISRA has been independently certified by SGS-TÜV SAAR GmbH for use at the highest integrity levels for safety-related standards including ISO 26262, IEC 61508, IEC 62304, EN 50128, EN 50657, and IEC 60880. It is also end user qualifiable for standards such as DO-178C with a Qualification Support Kit that automatically generates the necessary reports for tool qualification.

Cantata is a registered trademark of QA Systems GmbH ©. The Cantata logo, trade names and this document are trademarks and property of QA Systems GmbH ©.

QA Systems
With offices in Waiblingen, Germany | Bath, UK | Boston, USA | Paris, France | Milan, Italy
www.qa-systems.com | www.qa-systems.de