White Paper

C and C++ Software Testing – Am I 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.

QA SYSTEMS The Software Quality Company

www.qa-systems.com

Contents

Copyright Notice

1 Introduction

This paper looks at coverage techniques that can be employed when testing C and/or C++ code. We look at the coverage requirements of applicable standards and how coverage metrics can be used effectively throughout your development lifecycle. The paper also explores some 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 coverage, it is assumed that you appreciate the importance of a functioning test suite. This paper specifically outlines the coverage considerations of a successful testing regime. Further information on software testing can be found in other QA Systems white papers and publications. These are 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 were 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 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

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 (Figure 1)

- **Requirements** – various levels of detail defining e.g. 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 which has been executed by tests. Test coverage measures the proportion of tests which have been run and passed.

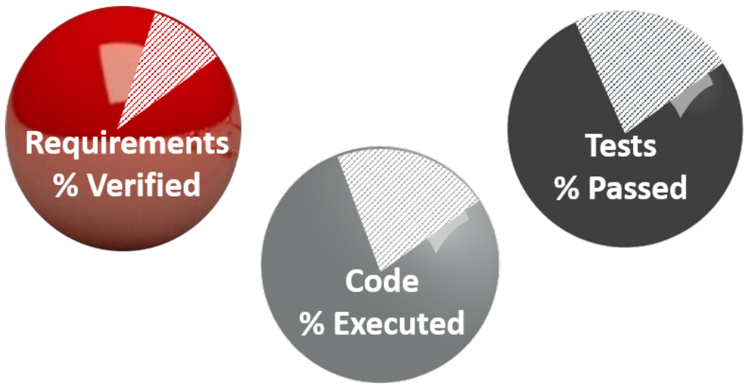


Figure 1 – 3 types of coverage in software testing measured as a percentage

Safety critical software standards, such as DO-178C and ISO26262, recommend use of **requirements coverage**, **structural code coverage** and **test coverage**. Correct use of these concepts can also help developers outside of the safety critical arena. Appreciation of the terms and their use will help deliver a more reliable and robust application. Each of these 3 types of coverage is explored further below.

3 Requirements Coverage

Development professionals consider it important in critical software to produce a complete and verifiable list of requirements against which you can develop your code and corresponding tests. This holds true regardless of whether you are developing using the traditional waterfall development method or an iterative model (maybe based around agile or DevOps).

Well-defined requirements at the start of the project, or at the start of an agile sprint, will make the technical teams' jobs easier and more measurable. It is unlikely that requirements supplied by the customer will be complete enough to ensure project success. Usually, there will be some degree of interpretation, and an upfront process of adding detail to initial high-level requirements, to help shape the architectural design and implementation of the requirements in code. Situations where requirements are not documented, or where initial requirements are not final, will mandate additional study phases with the team and agreement with the customer.

A link between an individual requirement and corresponding test at each level (figure 3) will help track and prove that software meets its requirements. The number of verified requirements compared to the total number of requirements, expressed as a percentage, is termed requirements coverage.

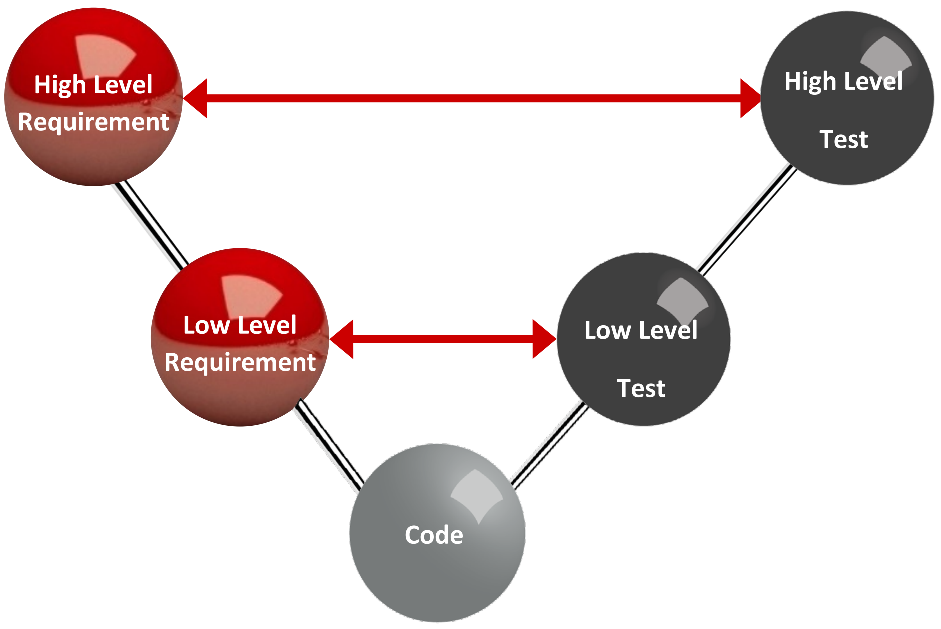


Figure 2 – Linking requirements to tests and each level

Linking individual tests to individual requirements provides the capability for bi-directional traceability between requirements and tests. Without these links it is not possible to know how thoroughly a set of tests verifies the correct code implementation of the requirement, i.e. requirements coverage in testing.

4 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.

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

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.

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

Reason for unexecuted code	Resulting Action
Code is 'Dead' (i.e. 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 but commented out to make it non-executable.

Reason for unexecuted code	Resulting Action
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 but commented out to make it un-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.

4.1 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)
- Decisions
- Conditions (Boolean operands)
- Relational Operators
- Loops
- MC/DC (Modified Condition / Decision Coverage), 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 code coverage metrics (depending on integrity level) are: Entry-point, Function Call, Statement, Decision and MC/DC coverage. These are explained in more detail below.

4.1.1 Entry-Point Coverage

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

4.1.2 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 commonly used metric to measure integration level testing.

4.1.3 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.

4.1.4 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.

4.1.5 MC/DC - Modified Condition / Decision Coverage

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% coverage in tests requiring the most test cases.

5 Test Coverage

Test coverage measures the proportion of your tests have run successfully (i.e. been executed and pass) on the code. While having tests which verify the requirements can be measured using requirements coverage, and the amount of the code executed by the test can be measured using structural code coverage, whether or not these tests are actually run and pass is the third critical thing to measure for software testing. Requirements coverage and code coverage only have meaning when the status of the tests which those metrics measure is itself measured. The simple expression of Test Coverage is as a percentage:

$$\%\ \textbf{Test Coverage} = \frac{\textbf{Number of Tests Passing}}{\textbf{Total number of Tests}}$$

However, there are test status questions to consider when looking at test coverage metrics:

- Are the executed tests for the correct version / variant for the requirements
- Are the tests which are run the correct version / build variant(s) for the code

Tests may be executed to verify that requirements have been correctly implemented in the code. Requirements will almost certainly change (new versions or variants) over time, and good engineering practice will version manage such changes. Changes to the version or variants of requirements may necessitate changes in the tests which verify them so ensuring the correct mapping of the version and/or variant of requirements to tests is essential if test coverage data is to be accurate.

Tests will be executed on a version or build variant of the source code. Source code will almost certainly change and/or be built as different variants. So as for requirements mapping, ensuring the correct build of the code is tested is essential for test coverage data to be accurate.

As tests are initially developed and re-run as changes occur, test coverage provides the third and final piece in the jigsaw of Coverage metrics.

6 Coverage by testing stage

Coverage of requirements, code and tests can be used at all stages of testing. Ideally coverage information of all types should be collected at each stage. Normally the first stage of testing will be at the unit level, at the bottom of the typical V model (figure 4 below). It is at this stage that obtaining high levels of coverage will be easiest, but it is the most likely stage to discover insufficiently defined requirements.

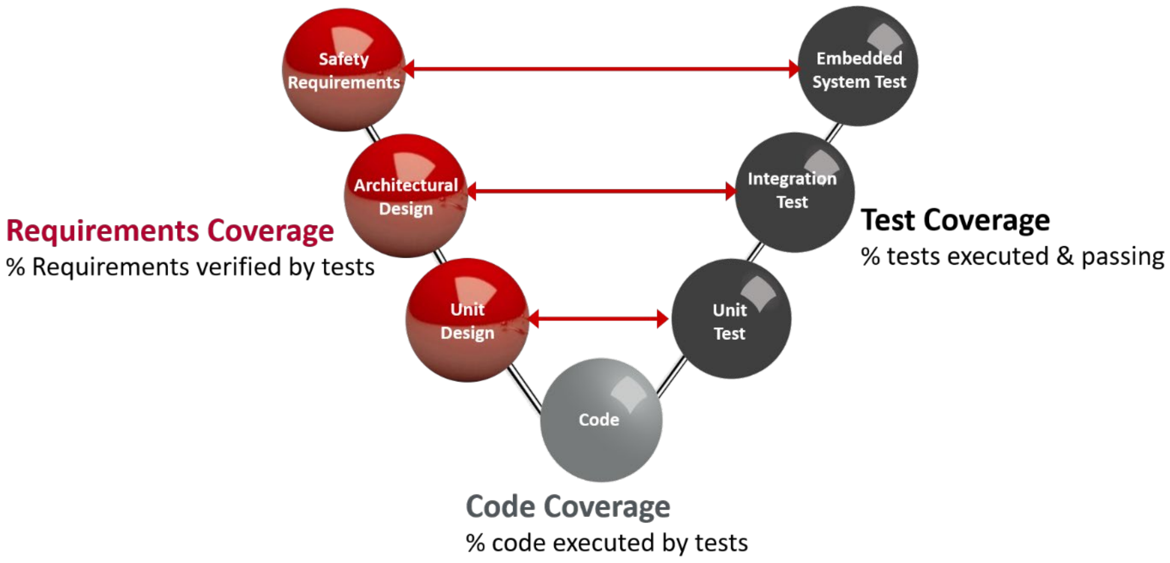


Figure 4 – A V model to illustrate stages of testing mapped to the corresponding requirements

One of the most common causes of applications being deployed with bugs is that the program experiences unpredictable, and therefore untested, input combinations when in the field. These types of errors can be discovered more readily as you build on your unit testing and move towards integration of the system. At the integration and system testing stages, code coverage becomes less of a driver towards completeness of testing and the focus moves more to requirements coverage.

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 these code units in test cases. Unit tests provide much greater control over the code being tested, and can be used to supplement integration and system tests enabling you to achieve 100% coverage overall. It is therefore very useful to be able to combine coverage data from different stages of testing.

It is almost impossible to obtain 100% code coverage during system level tests. Typically, during this stage of testing, you can reach 70% 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.

It is often more possible to obtain 100% requirements coverage during later stages of testing where the higher-level requirements are defined, than at unit or integration stages where detailed low-level requirements are often less defined.

It should always be possible to obtain 100% test coverage over all stages of testing. The more automated the tests at each stage the easier it will be to achieve, not only for the initial test run, but throughout subsequent regression test runs.

7 Why use coverage metrics

The 3 measures of coverage used in testing (requirements coverage, structural code coverage and test coverage) can not only monitor the thoroughness of testing, they can also guide test 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.

> 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 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

Setting coverage goals allows 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 coverage goal does not follow a linear progression. The graph in figure 5 illustrates this point. In the early stages of the project coverage metrics tend to increase in value quickly. As time progresses and you are left with more difficult to test scenarios, increasing coverage becomes harder.

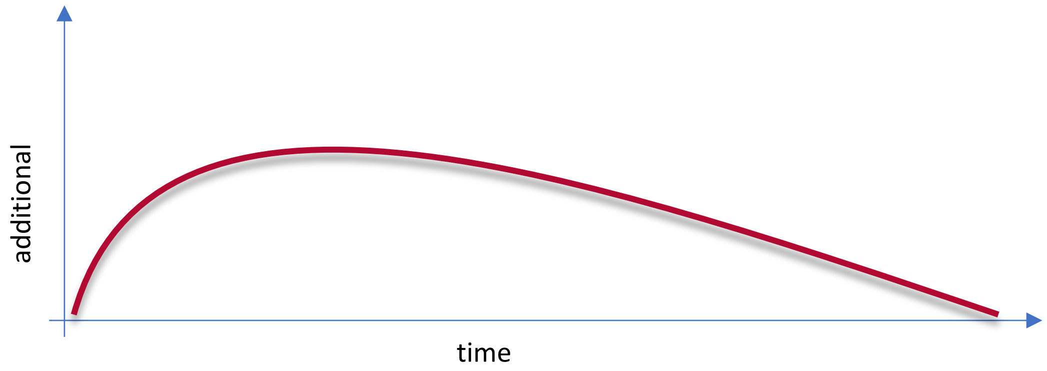


Figure 5 – An illustration of diminishing coverage returns

8 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 coverage metrics and target coverage values that you should achieve in your project.

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

Sector

Requirements Coverage	ISO 26262	DO-178B/C	IEC 62304	IEC 61508	EN 50128	IEC 60880
100% High-Level Requirements	A	D	A	1	1	A
100% Low-Level Requirements	A	C	C	1	1	A

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	-

Test Coverage	ISO 26262	DO-178B/C	IEC 62304	IEC 61508	EN 50128	IEC 60880
100% Tests Executed & Passing	A	D	A	1	1	A

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.

9 Using requirements coverage

9.1 Techniques

The main techniques adopted for requirements coverage are bi-directional traceability and the use of a Requirements Coverage Matrix. Bi-directional traceability between requirements and tests is needed to ensure all requirements have associated tests which verify they have been correctly implemented in code. It is also mandated for requirements-based testing by all the main software safety standards.

While requirements can of course be linked to (traced) with many software development artefacts, such as business objectives, contracts, review records and software implementation code items, the key relationship for software testing is that between individual requirements as individual tests or test cases as below.

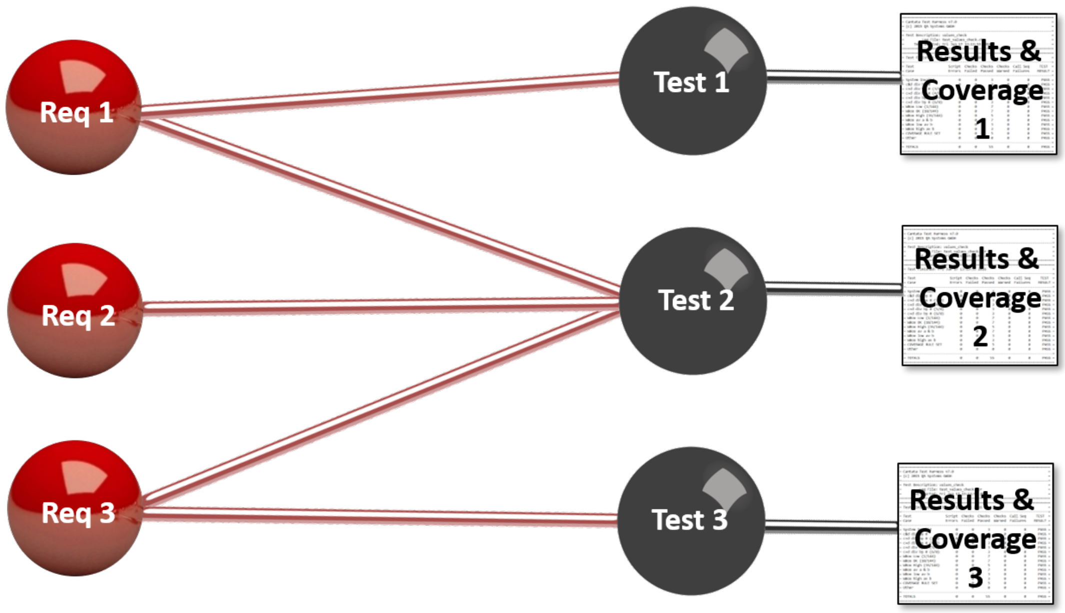


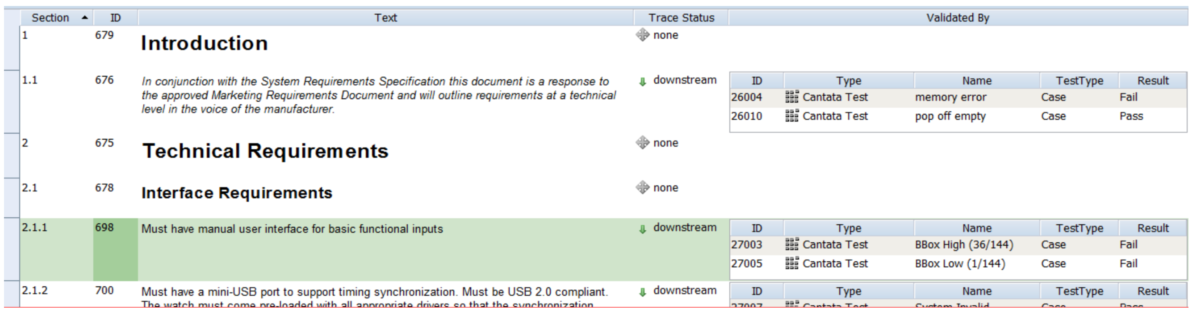
Figure 7 – Traceable link between requirements and tests

A Requirements Coverage Matrix is a table that contains a list of documented requirements for a product/task and links to the corresponding test scenarios. It is used for verification of requirements coverage against test cases. The matrix can also serve as a collection and storage place for the final approved requirements.

There are several considerations when putting together a suitable requirements matrix for a project:

- Does the project budget allow enough time for creating a Requirements Coverage Matrix (or matrices)?
- Any requirements to a product, or to a new task, will be substantial and suggest a number of changes.

- Requirements provided by the customer are usually not complete and do not take into account all the details of the product implementation, thus demanding further detailed analysis and involvement of a team on both the developers' and the customer's side.
- Initial requirements which are not documented and formulated clearly, will slow things down considerably as you capture these adequately.
- Do we require mandatory coverage of all requirements with test cases?



Section	ID	Text	Trace Status	Validated By				
				ID	Type	Name	TestType	Result
1	679	**Introduction**	none					
1.1	676	*In conjunction with the System Requirements Specification this document is a response to the approved Marketing Requirements Document and will outline requirements at a technical level in the voice of the manufacturer.*	downstream	26004	Cantata Test	memory error	Case	Fail
				26010	Cantata Test	pop off empty	Case	Pass
2	675	**Technical Requirements**	none					
2.1	678	**Interface Requirements**	none					
2.1.1	698	Must have manual user interface for basic functional inputs	downstream	27003	Cantata Test	BBox High (36/144)	Case	Fail
				27005	Cantata Test	BBox Low (1/144)	Case	Fail
2.1.2	700	Must have a mini-USB port to support timing synchronization. Must be USB 2.0 compliant. The watch must come pre-loaded with all appropriate drivers so that the synchronization	downstream	27007	Cantata Test	System Invalid	Case	Pass

Figure 8 – A typical requirements coverage matrix

Anyone, who has a completed Requirements Coverage Matrix, like the one in figure 8, is not left wondering where a certain requirement comes from, or whether it is approved with the team and the customer, or whether it is covered with test cases. The table provides all the necessary information.

9.2 Tools

Requirements management tools, often as part of ALM (Application Lifecyle Management) or PLM (Product Lifecycle Management) tool suites, offer the most full-featured capabilities for management of requirements, including bi-directional tracing of the requirements with tests. It is not however necessary to use a full featured requirements management tool for requirements coverage or as a requirements coverage matrix. The most commonly used tool for this remains Microsoft Excel® with simple pivot tables used for bi-directional tracing.

More importantly for tools is how the requirements information can be used by the tester (to ensure that the test verifies the requirement) in their test tool, and how requirements are linked to tests for traceability and requirements coverage.

It is very helpful for the tester to see the details of requirements when writing and reviewing tests. The best way of achieving this is by importing the requirements details information into the test tool. The greater the information (i.e. not just a requirement identifier, but the full description including any tables and images), the easier it is for the tester both to define the test and to link it to the correct requirement for traceability and requirements coverage.

Importing requirements information into the test tool is however only temporary and the first part of the process. Thereafter the requirements need to be linked to tests, the tests executed on the code and the tests results exported back to the requirements management tool along with the traceability link between them. As requirements are linked to many more items in a software project than just tests, it is very advisable to retain the requirements management tool as the definitive repository where bi-directional traceability, and requirements coverage can be analysed.

The main means by which such requirements can be passed between tools is by using CSV, Excel or XML formats. The most useful is a specialised form of XML called ReqIF^{TM} – an open standard Requirements Interchange Format.

The typical data flow between requirements management tools and test tools which makes requirements coverage most effective is outlined in the diagram below.

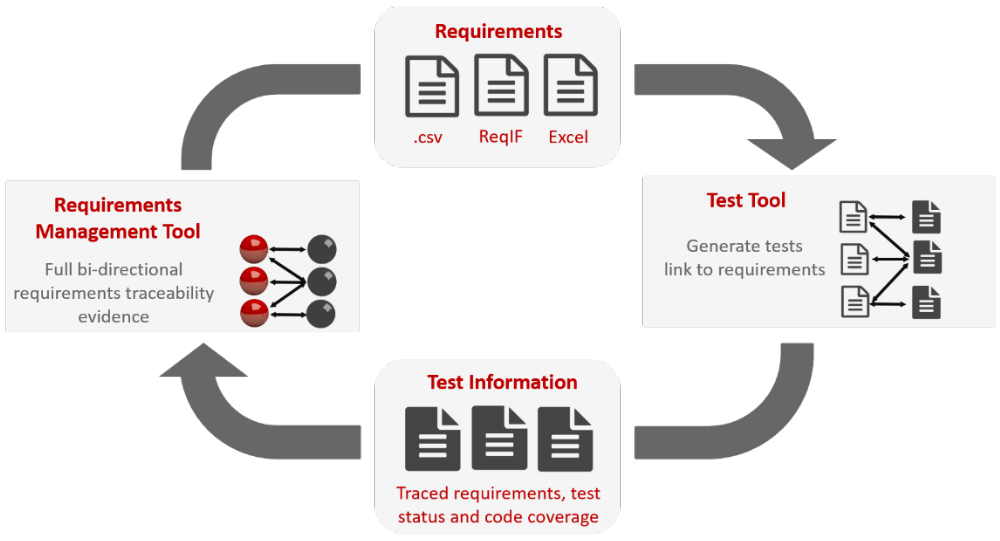

Requirements
.csv
ReqIF
Excel
Test Tool
Generate tests
link to requirements
Test Information
Traced requirements, test status and code coverage
Requirements Management Tool
Full bi-directional requirements traceability evidence

Figure 9 – Data flow between requirements management tools and test tools

10 Using code coverage

10.1 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.

10.2 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.

10.3 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 behaviour 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.

10.3.1 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.

10.3.2 Expected behaviour

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 behaviour 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 behaviour of the code under test is unaffected by measuring the code coverage.

10.4 Code coverage special considerations

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

10.4.1 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.

10.4.2 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.

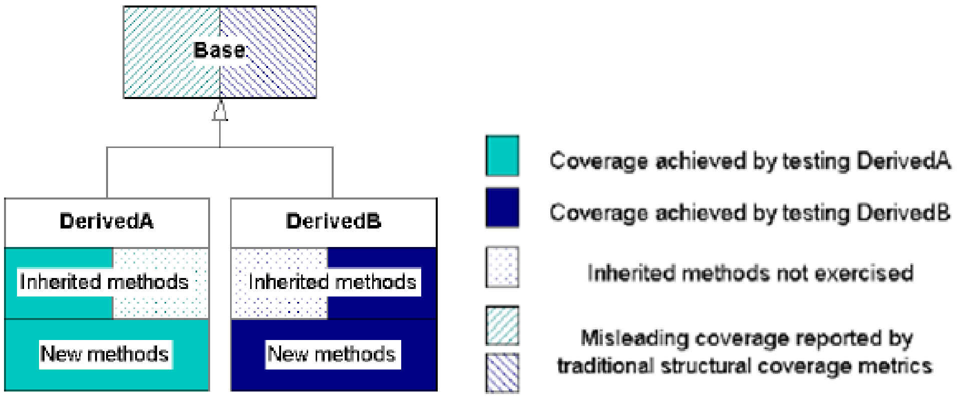

Base
DerivedA
Inherited methods
New methods
DerivedB
Inherited methods
New methods
Coverage achieved by testing DerivedA
Coverage achieved by testing DerivedB
Inherited methods not exercised
Misleading coverage reported by traditional structural coverage metrics

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.

10.4.3 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.

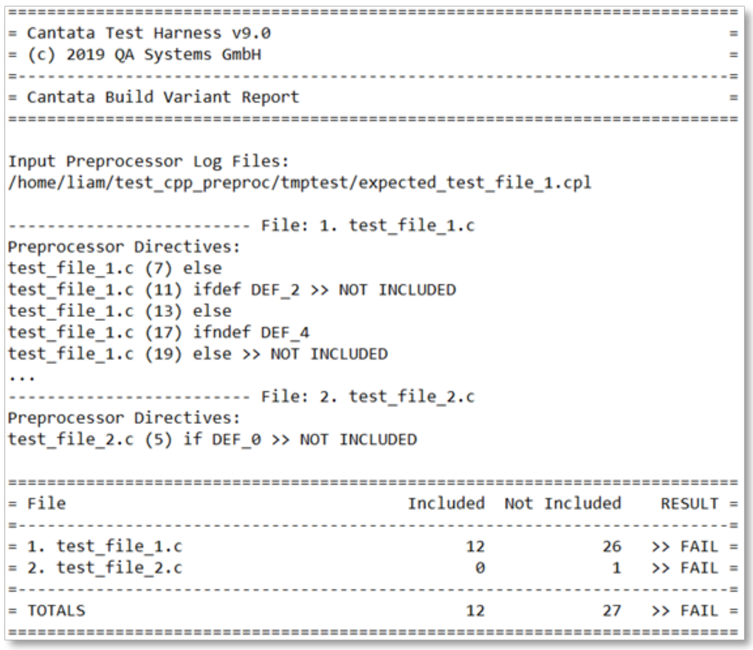
10.4.4 Thread context coverage

When testing multi-threaded code the behaviour 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.

10.4.5 Build variant coverage

When testing the same source code built with different variants using pre-compile defines (#defines) the behaviour 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.

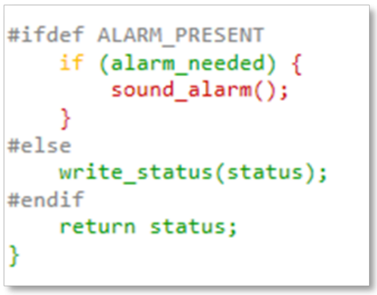


```
===========================================================================
= Cantata Test Harness v9.0                                               =
= (c) 2019 QA Systems GmbH                                                =
=-------------------------------------------------------------------------=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

10.5 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/)

11 Using test coverage

11.1 Techniques

As test coverage measures the proportion of your tests which have run successfully, the key technique is effective management of the code and tests as inputs and the results as outputs.

Configuration management of all inputs to and outputs from the testing process provides the necessary current test status, over-time trending information and certification ready test results evidence of compliance with the testing requirements of software safety standards.

Test management dashboards provide managers with actionable test coverage data to monitor testing progress. They can also provide the data to optimise the efficiency of a regression testing regime and enable “what-if” dependency analysis for dealing with changes in code, requirements or test environments.

11.2 Tools

The tools which can be used for gathering and recording test coverage need to provide the appropriate level of information for its intended use.

For managers that usually means dashboards showing for each part of the application:

> Source code items for which no tests have yet been created at all
> Test coverage percentage (% of tests passing) for each part of the application
> Whether the executed tests are the correct version / variant, for the requirements
> Whether the executed tests are the correct version / build variant(s), for the code

For engineers that usually means the detail records for each code item being tested:

> Test coverage percentage (% of tests passing)
> Pass or Fail status for each test / test case
> Test execution (whether & when the test ran to completion or did not)
> Test execution environment (platform & options used)
> Test execution verifying requirements (link to requirements)
> Test execution for code version / build variant

For quality assurance or standards compliance functions, that usually means:

> The detailed test execution records for engineers generated using appropriate tools suitable for use under the quality plan or in accordance with the report requirements of the software safety standard.

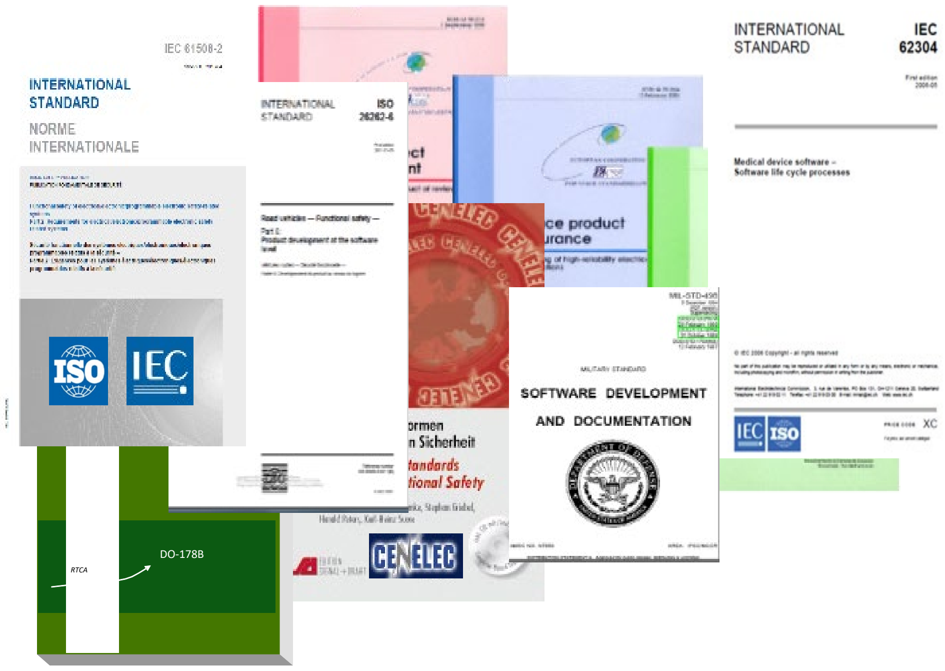
12 Coverage in the Cantata tool

12.1 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 requirements coverage, code coverage and test coverage (which it does), Cantata helps you achieve the desired levels of 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 code coverage, comprehensively analysed and results reported for certification compliance. Built on Eclipse, Cantata integrates easily with developer desktop compilers and embedded target platforms.

12.2 Certified coverage for software testing


IEC 61508-2
INTERNATIONAL STANDARD
NORME INTERNATIONALE
INTERNATIONAL STANDARD ISO 26262-6
Road vehicles — Functional safety —
Part 6: Product development at the software level
INTERNATIONAL STANDARD IEC 62304
Medical device software – Software life cycle processes
MIL-STD-498
MILITARY STANDARD
SOFTWARE DEVELOPMENT AND DOCUMENTATION
CENELEC
DO-178B
RTCA

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).

12.3 One tool that covers it all

In addition to providing comprehensive unit and integration testing for C and C++, Cantata implements the full range of coverage analysis set out above in this paper.

12.3.1 Requirements Coverage in Cantata

Cantata is integrated with the main industrial requirements management, ALM / PLM tools to provide an easy to use drag and drop means to obtain requirements coverage.

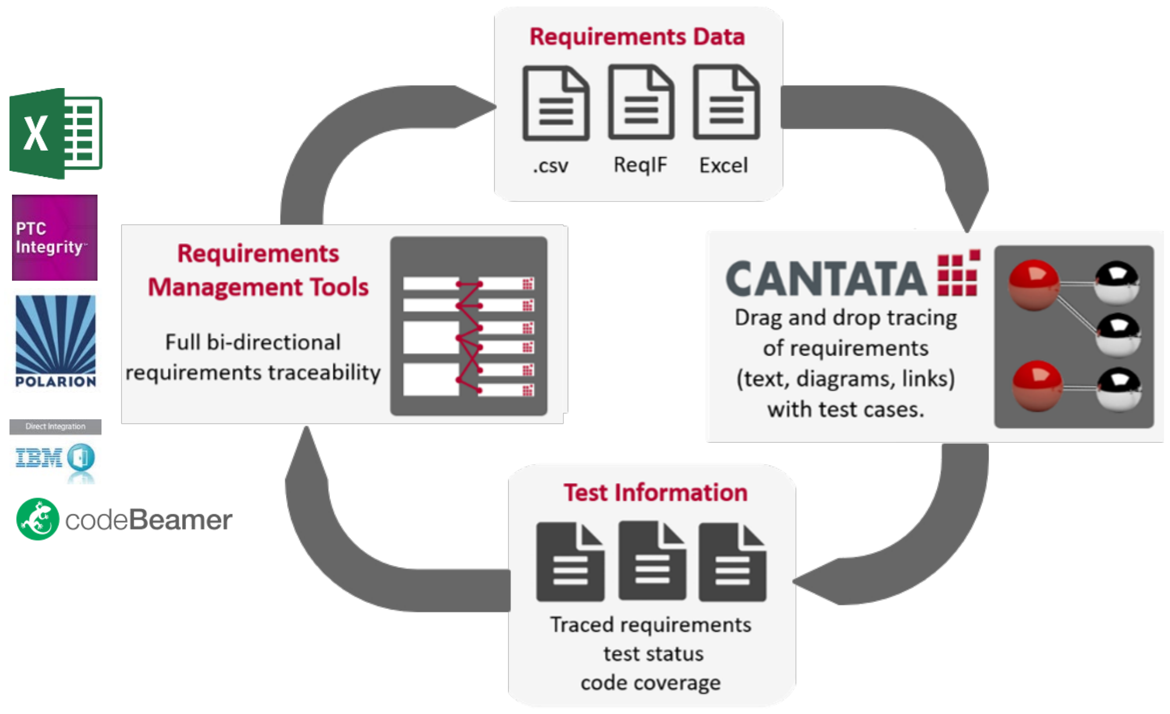
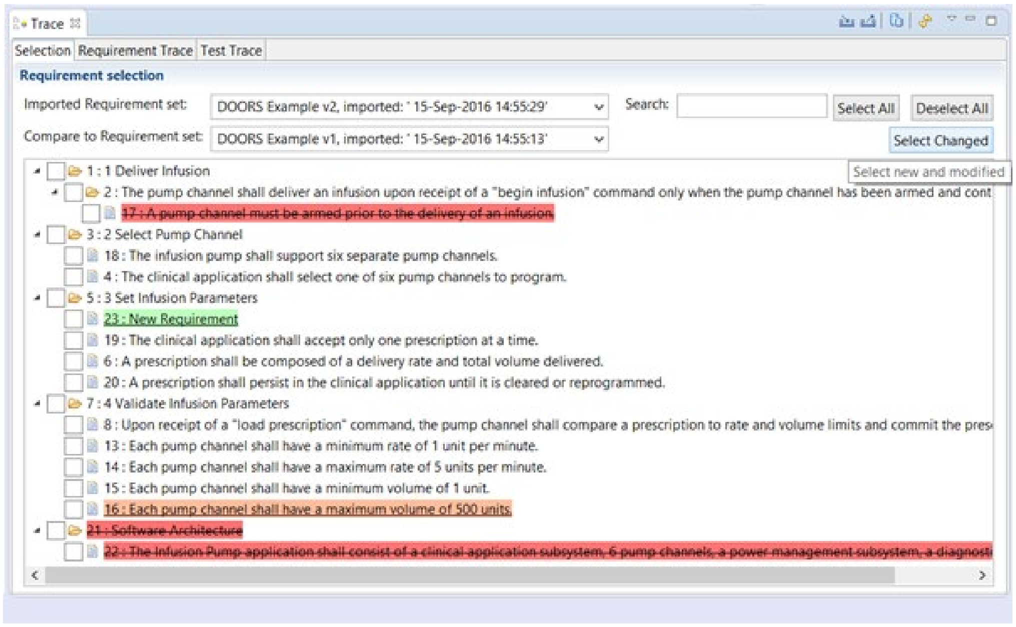

Requirements Data
.csv
ReqIF
Excel
PTC Integrity
POLARION
Direct Integration
IBM
codeBeamer
Requirements Management Tools
Full bi-directional requirements traceability
CANTATA
Drag and drop tracing of requirements (text, diagrams, links) with test cases.
Test Information
Traced requirements
test status
code coverage

Figure 12 – Requirements integration

Sets of requirements can be imported into the Cantata Trace view. Full visibility of the requirements (text, images and hyperlinks) in HTML allows the tester to easily see requirements alongside the test cases which verify the code implementation. Requirements sets can even be managed in Cantata to identify new, changed or deleted requirements making the change management of version and variants easier.



Requirement selection

Imported Requirement set: DOORS Example v2, imported: ' 15-Sep-2016 14:55:29'

Search:

Compare to Requirement set: DOORS Example v1, imported: ' 15-Sep-2016 14:55:13'

1 : 1 Deliver Infusion
2 : The pump channel shall deliver an infusion upon receipt of a "begin infusion" command only when the pump channel has been armed and cont
~~17 : A pump channel must be armed prior to the delivery of an infusion.~~
3 : 2 Select Pump Channel
18 : The infusion pump shall support six separate pump channels.
4 : The clinical application shall select one of six pump channels to program.
5 : 3 Set Infusion Parameters
23 : New Requirement
19 : The clinical application shall accept only one prescription at a time.
6 : A prescription shall be composed of a delivery rate and total volume delivered.
20 : A prescription shall persist in the clinical application until it is cleared or reprogrammed.
7 : 4 Validate Infusion Parameters
8 : Upon receipt of a "load prescription" command, the pump channel shall compare a prescription to rate and volume limits and commit the pres
13 : Each pump channel shall have a minimum rate of 1 unit per minute.
14 : Each pump channel shall have a maximum rate of 5 units per minute.
15 : Each pump channel shall have a minimum volume of 1 unit.
16 : Each pump channel shall have a maximum volume of 500 units.
~~21 : Software Architecture~~
~~22 : The Infusion Pump application shall consist of a clinical application subsystem, 6 pump channels, a power management subsystem, a diagnosti~~

Figure 13 – Importing requirements

For more information on the Cantata requirements coverage capability Cantata Trace, please see the Cantata requirements traceability webpage.

12.3.2 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:

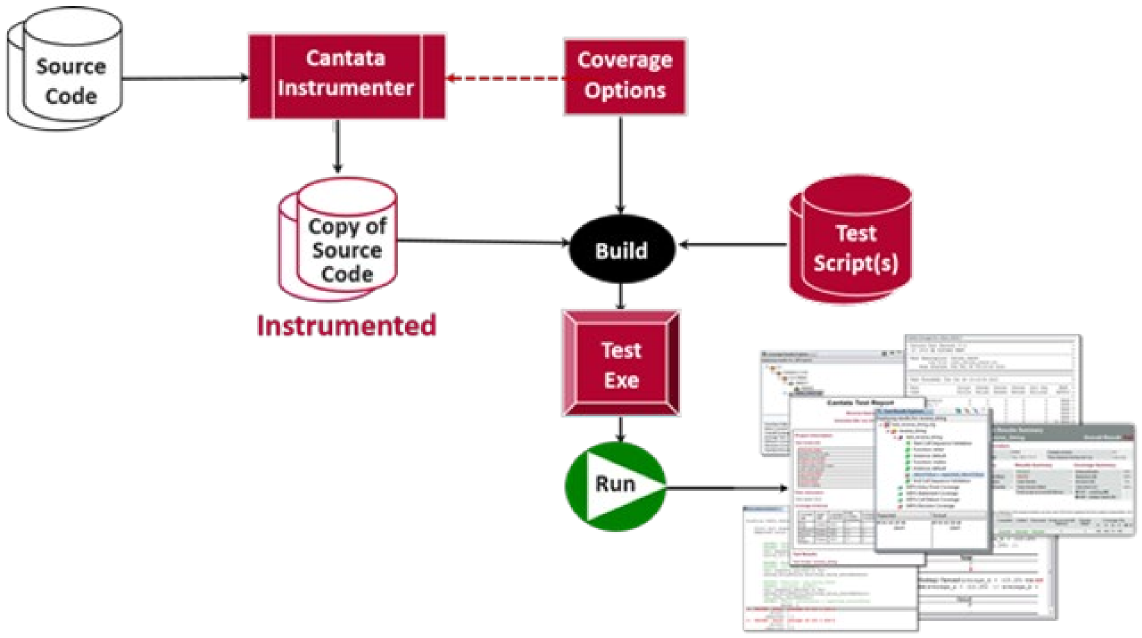
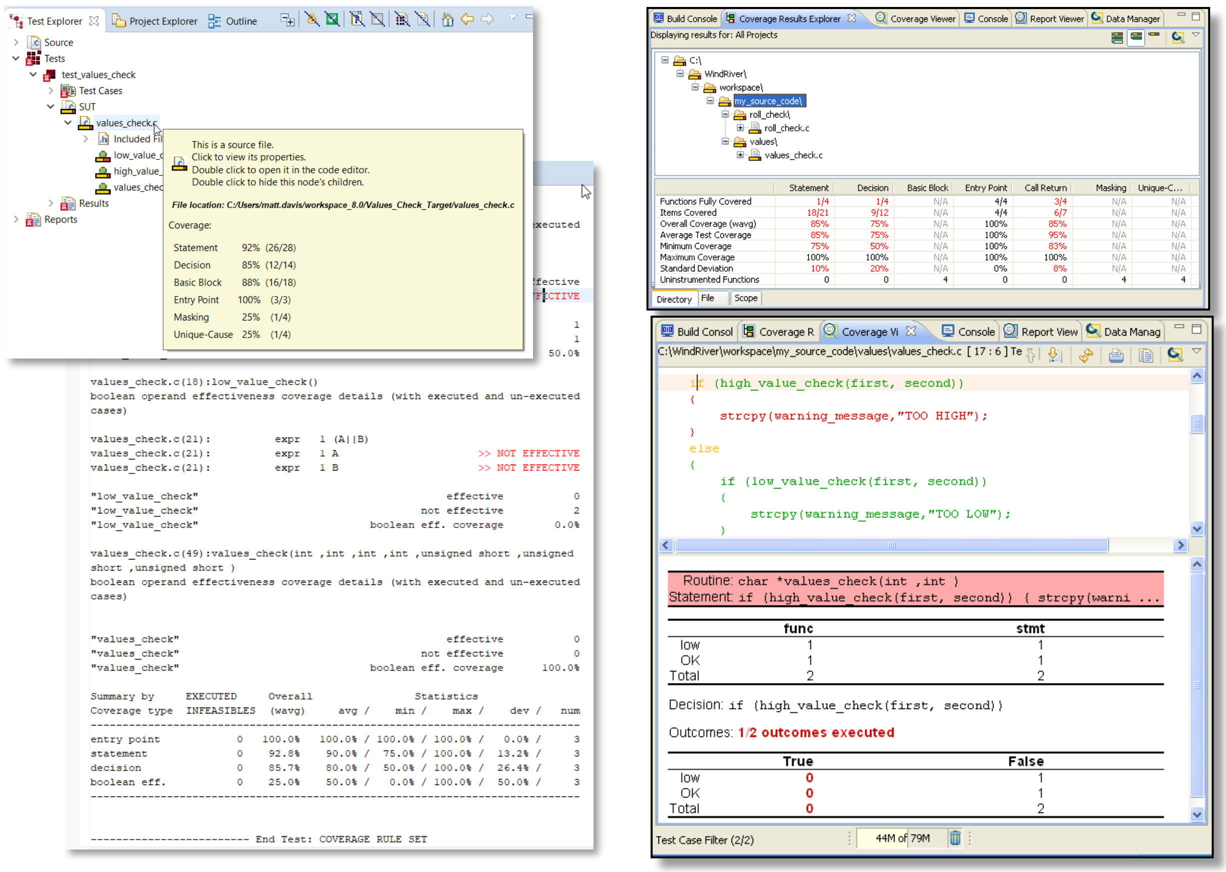


Figure 14 – Combined instrumentation and test process

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



This is a source file.
Click to view its properties.
Double click to open it in the code editor.
Double click to hide this node's children.

File location: C:/Users/matt.davis/workspace_8.0/Values_Check_Target/values_check.c

Coverage:

Statement	92%	(26/28)
Decision	85%	(12/14)
Basic Block	88%	(16/18)
Entry Point	100%	(3/3)
Masking	25%	(1/4)
Unique-Cause	25%	(1/4)

```
values_check.c(18):low_value_check()
boolean operand effectiveness coverage details (with executed and un-executed
cases)

values_check.c(21):          expr   1 (A||B)
values_check.c(21):          expr   1 A                      >> NOT EFFECTIVE
values_check.c(21):          expr   1 B                      >> NOT EFFECTIVE

"low_value_check"                                  effective          0
"low_value_check"                              not effective          2
"low_value_check"                       boolean eff. coverage      0.0%

values_check.c(49):values_check(int ,int ,int ,int ,unsigned short ,unsigned
short ,unsigned short )
boolean operand effectiveness coverage details (with executed and un-executed
cases)


"values_check"                                     effective          0
"values_check"                                 not effective          0
"values_check"                          boolean eff. coverage    100.0%

Summary by     EXECUTED     Overall                Statistics
Coverage type  INFEASIBLES  (wavg)      avg /    min /    max /    dev /   num
-----------------------------------------------------------------------------
entry point             0   100.0%   100.0% / 100.0% / 100.0% /   0.0% /     3
statement               0    92.8%    90.0% /  75.0% / 100.0% /  13.2% /     3
decision                0    85.7%    80.0% /  50.0% / 100.0% /  26.4% /     3
boolean eff.            0    25.0%    50.0% /   0.0% / 100.0% /  50.0% /     3
-----------------------------------------------------------------------------


------------------------ End Test: COVERAGE RULE SET
```

Displaying results for: All Projects

	Statement	Decision	Basic Block	Entry Point	Call Return	Masking	Unique-C...
Functions Fully Covered	1/4	1/4	N/A	4/4	3/4	N/A	N/A
Items Covered	18/21	9/12	N/A	4/4	6/7	N/A	N/A
Overall Coverage (wavg)	85%	75%	N/A	100%	85%	N/A	N/A
Average Test Coverage	85%	75%	N/A	100%	95%	N/A	N/A
Minimum Coverage	75%	50%	N/A	100%	83%	N/A	N/A
Maximum Coverage	100%	100%	N/A	100%	100%	N/A	N/A
Standard Deviation	10%	20%	N/A	0%	8%	N/A	N/A
Uninstrumented Functions	0	0	4	0	0	4	4

C:\WindRiver\workspace\my_source_code\values\values_check.c [17 : 6] Te

```
    if (high_value_check(first, second))
    {
        strcpy(warning_message,"TOO HIGH");
    }
    else
    {
        if (low_value_check(first, second))
        {
            strcpy(warning_message,"TOO LOW");
        }
```

Routine: `char *values_check(int ,int )`
Statement: `if (high_value_check(first, second)) { strcpy(warni ...`

	func	stmt
low	1	1
OK	1	1
Total	2	2

Decision: `if (high_value_check(first, second))`

Outcomes: **1/2 outcomes executed**

	True	False
low	0	1
OK	0	1
Total	0	2

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.

12.3.3 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:

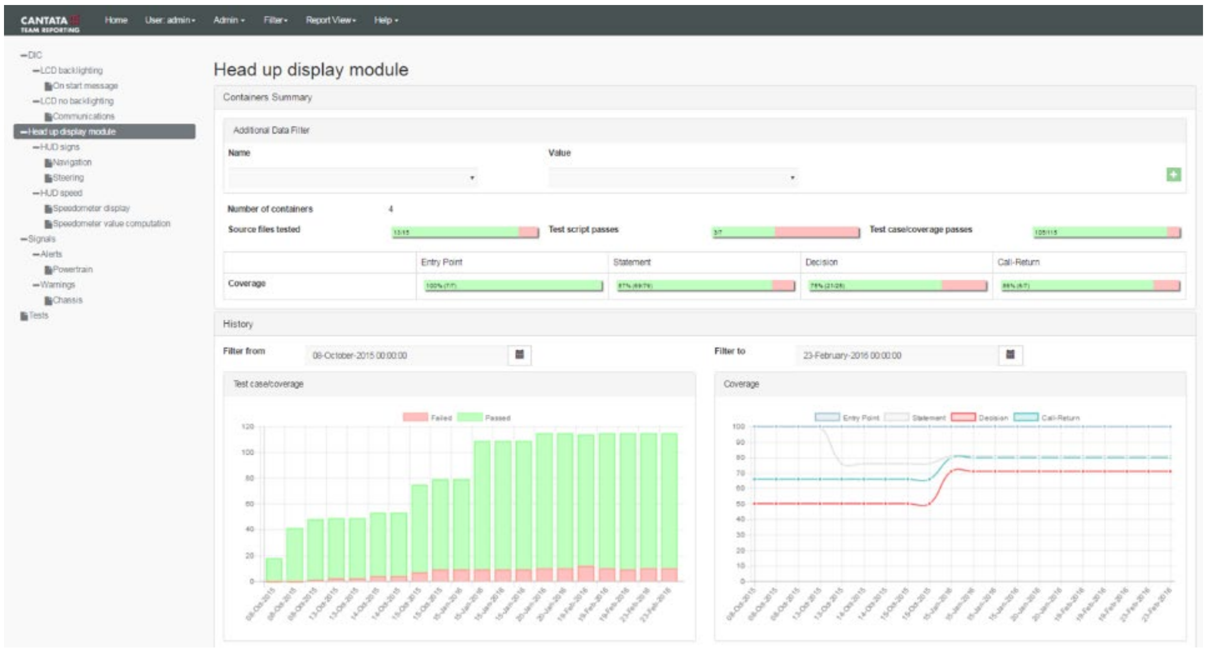
> Configure automatic test generation
> Identify code testability issues
> Generate tests with full code coverage
> Plug ‘edge case’ gaps in coverage from existing tests
> Create a thorough safety net of baseline regression tests
> Link generated test cases to requirements for requirements

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

12.3.4 Test Coverage

Cantata reports test 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 testing status, test coverage, historical data and trends over multiple codebases.



Head up display module

Containers Summary

Additional Data Filter

Name	Value

Number of containers 4

Source files tested 13/15 | Test script passes 3/7 | Test case/coverage passes 105/115

	Entry Point	Statement	Decision	Call-Return
Coverage	100% (7/7)	87% (69/79)	75% (21/28)	86% (6/7)

History

Filter from 08-October-2015 00:00:00 | Filter to 23-February-2016 00:00:00

Test case/coverage

Coverage

Figure 16 – An example management dashboard in Cantata

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

13 What next?

This paper has presented some arguments and explanations as to why and how the 3 measures of 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 be covered for your software testing, we look forward to hearing from you.

14 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/

[8] Automated Requirements-Based Testing for ISO 26262
A paper produced by QA Systems http://www.qa-systems.com/

[9] Automated Requirements-Based Testing for DO-178C
A paper produced by QA Systems http://www.qa-systems.com/

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