RAPITA SYSTEMS
A DANLAW COMPANY

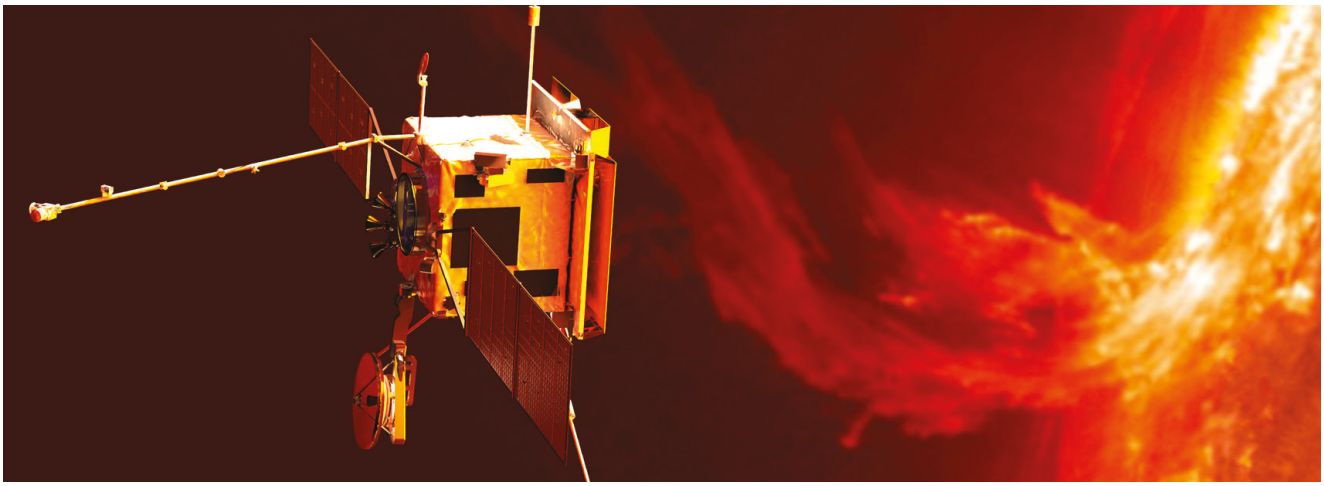
Safety through quality

WHITE PAPER

Seven Roadblocks to 100% Structural Coverage (and how to avoid them)

Contents

1. Introduction



Correct test results are essential

Throughout this white paper, we assume that in addition to monitoring the code coverage of tests, you are also checking the tests result in "correct" behavior, whether that is a specific result, responses within a particular time frame or some other criteria.

Unless test results are 100% correct, code coverage measurements are meaningless.

1.1 What is code coverage for?

Code coverage (also referred to as structural coverage analysis) is an important verification tool for establishing the completeness and adequacy of testing.

DO-178B/C and ISO 26262 both emphasise the use of requirements-based testing as an important part of the software verification process. In requirements-based testing, source code and tests are derived from high and low level requirements. Checking traceability between the requirements, the test cases and the source code demonstrates:

- Every requirement has one or more test cases, which verify that it has been correctly implemented
- All source code is traceable to a requirement

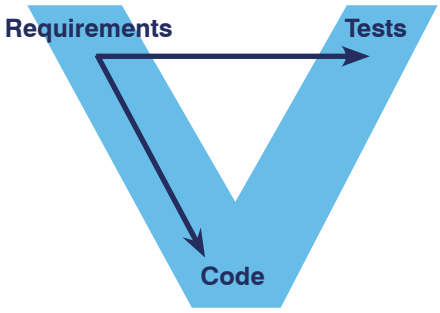


Figure 1 – Requirements traceability

Traceability between code, requirements and tests is complemented by measuring structural coverage of the code when the tests are executed. Where coverage is less than 100%, this points to:

- Code that is not traceable to requirements
- Inadequate tests
- Incomplete requirements
- A combination of the above

Different coverage criteria (see Table 1, on page 3) results in the degree of rigor applied in testing the code to reflect the Development Assurance Level (DAL) of the system.

1.2 What does it mean to get 100% code coverage?

When you use requirements-based testing, 100% code coverage means that, subject to the coverage criteria used, no code exists which cannot be traced to a requirement. For example, using function coverage, every function is traceable to a requirement (but individual statements within the coverage may not be).

What 100% code coverage **does not mean** is:

- Your code is correct. You've got test cases which, when aggregated, exercise every line of code. This is not sufficient to show there are no bugs. As long ago as 1969 Edsger Dijkstra noted "testing shows the presence of bugs, not their absence" – in other words, just because testing doesn't show any errors, it doesn't mean they are not present.
- Your software requirements are correct. This is determined through validation of the requirements with the customer.
- You've tested 100% of your requirements. Merely achieving 100% code coverage isn't enough. This is only true if you achieve 100% code coverage AND you have a test for 100% of your requirements AND every test passes.
- Your compiler translated your code correctly. You might discover the compiler is inserting errors which cause incorrect results in some situations (ones you haven't tested for).
- You have covered 100% of your object code. Even when all statements and conditions of the source code are being executed, the compiler can introduce additional structures into the object code.

1.3 Does it matter which code coverage criteria I am using?

There are a number of different code coverage criteria, which affect how thoroughly test completeness is assessed. The code coverage criteria you use are typically driven by the integrity level of the software (DAL for DO-178B/C or ASIL for ISO 26262) that you are using.

Table 1. Coverage requirements by standard/guideline

Coverage type	DO-178B/C	ISO 26262 (software architecture)	ISO 26262 (unit test)
Function Coverage	Used with MC/DC	ASIL C,D Highly Recommended ASIL A,B Recommended	-
Call Coverage	Not Required	ASIL C,D Highly Recommended ASIL A,B Recommended	Not Required
Statment Coverage	Level A, B, C Required	Not Required	ASIL A, B Highly Recommended ASIL C, D Recommended
Decision Coverage	Level A, B Required	Not Required	Not Required
Branch Coverage	Not Required	Not Required	ASIL B, C, D Highly Recommended ASIL A Recommended
MC/DC	Level A Required	Not Required	ASIL D Highly Recommended ASIL A, B, C Recommended

Irrespective of which criteria you are using in your testing, if you cannot achieve 100% code coverage, the strategies you need to apply are the same – so for the purposes this paper it doesn't matter which coverage criteria you are using. There is one exception to this, which applies to MC/DC (see *Impossible combinations of events* on *page 8*).

2. What are the barriers to 100% code coverage

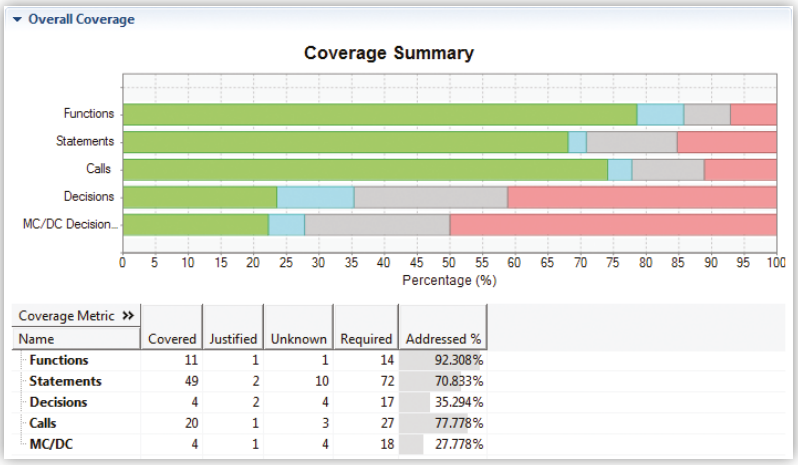
```
131     if Sign_Word < 0 then
132        Sign_Word := -Sign_Word;
133        Neg_Sign := True;
134     end if;
135     Word := Systemtypes.Word(Sign_Word);
136     if Neg_Sign then
137        Word := Word + 2**15;
38     end if;
39     return word;
40  end Millirad_To_word;

1   function Int_To_Millirad(Count : in Systemtypes.Integer32)
2                            return Measuretypes.Millirad
3   is
       Neg_Sign : Boolean;
       Tmp_Count : Systemtypes.Unsigned32;
       Answer : Millirad;
    begin
```

There are a number of reasons why it may not be possible to achieve 100% code coverage. In practice these usually conspire to make it very rare to achieve full coverage. In this paper, we've identified seven of the most frequently occurring reasons:

- Missing requirements
- Missing or incorrect tests
- Dead code
- Deactivated code
- Defensive programming
- Impossible combinations of events
- Compiler-introduced errors

When structural coverage analysis shows less than 100% coverage, one reasonable response would be to follow a process similar to this:

1. Review code to establish where coverage is missing.
2. Identify the cause of the missing code. The list (see left) gives a checklist for likely causes of this.
3. Identify what action needs to be undertaken to remedy the missing coverage.



Coverage Metric

Name	Covered	Justified	Unknown	Required	Addressed %
Functions	11	1	1	14	92.308%
Statements	49	2	10	72	70.833%
Decisions	4	2	4	17	35.294%
Calls	20	1	3	27	77.778%
MC/DC	4	1	4	18	27.778%

Figure 2 – Rapi**Cover** report showing incomplete coverage

2.1 Missing requirements

Identifying missing requirements is one of the primary reasons for performing code coverage.

If code exists for which there is no requirement, there will be no tests for that requirement, and consequently that code would not be covered during testing. It may be the case that the code in question is implicitly related to an existing requirement, which needs to be refined to cover additional cases, or to be treated in more detail (a derived requirement in DO-178C).

What to do about it?

If the missing coverage is due to missing requirements, you will need to add new requirements (or refine existing ones), develop tests for the new/modified requirements, and then run the tests.

Rapi**Cover** lets you compare your reports so you can see how new tests you've added have contributed to your total coverage.

Function	Statement Coverage					
Name	Covered	Justified	Unknown	Required	Addressed	#TestCases
coverage_tests.bar	0	0	0	0	0.000	+1 (+33%)
.test_1	0	0	0	0	0.000	0
.test_2	0	0	0	0	0.000	0
.test_3	0	0	0	0	0.000	0
.test_4	+1	0	0	0	+1.000	+1 (+100%)
.test_1_2	0	0	0	0	0.000	0
.test_3_4	+1	0	0	0	+0.333	+1 (+100%)
.justified_uncovered	0	0	0	0	0.000	0
.justified_covered	0	0	0	0	0.000	+1 (+100%)
.justified_decision	0	0	0	0	0.000	0
.uncovered	0	0	0	0	0.000	N/A
.unknown	0	0	0	0	0.000	N/A
.unreachable	0	0	0	0	0.000	N/A
coverage_tests.root	+5	0	0	0	+0.208	+1 (+33%)

Figure 3 – Rapi**Cover** coverage report comparison

2.2 Missing or incorrest tests

Your requirements may define functionality that applies to particular piece of code. However, if there are no tests that exercise that section of code, your coverage will be incomplete.

What to do about it?

If you have identified that:

- there is a requirement for the code identified by the missing coverage; but
- there is no relevant test traceable to that requirement

you will need to implement additional tests, ensure that they trace to the correct requirement(s), and run the new tests. As when dealing with missing requirements, it is useful to be able to compare your reports in this situation.

2.3 Dead code

Identifying missing requirements is one of the primary reasons for performing code coverage.

If code exists for which there is no requirement, there will be no tests for that requirement, and consequently that code would not be covered during testing. It may be the case that the code in question is implicitly related to an existing requirement, which needs to be refined to cover additional cases, or to be treated in more detail (a derived requirement in DO-178C).

What to do about it?

DO-178C recommends "The [dead] code should be removed and an analysis performed to assess the effect and the need for reverification."

For ISO 26262, the general guidance for "insufficient" coverage applies, namely to provide a rationale for why the dead code is acceptable [6:9.4.5].

If it is not possible to remove dead code, it may be possible to provide some form of justification to demonstrate why it cannot be executed.

What the standards say

DO-178C: Dead code is "Executable Object Code (or data) which exists as a result of a software development error but cannot be executed (code) or used (data) in any operational configuration of the target computer environment." [DO-178C]

2.4 Deactivated code

Like dead code, deactivated code is also classed as extraneous code by DO-178C. Unlike dead code, deactivated code is deliberately included in the target system. Examples of deactivated code include: unused legacy code, unused library functions, or code that is only executed in certain hardware configurations (for example when particular mode is selected by adjusting jump leads on hardware pins).

What to do about it?

DO-178C recommends "The [dead] code should be removed and an analysis performed to assess the effect and the need for reverification." [6.4.4.3c].

If specific sections of code are to be treated as deactivated code, a reasonable approach is to provide some form of justification as to why the code:

- will never be executed during normal operation; or
- will not affect the execution of the system.

[1]RTCA DO-178C 6.4.4.3c

[2]ISO 26262 6:9.4.5

What the standards say

ISO 26262: Defensive programming is highly recommended by ISO 26262 for ASIL C and D, and is recommended for ASIL B.[3]

DO-178C: Defensive programming practices may be considered to improve robustness.[4]

2.5 Defensive programming

You might find 100% code coverage cannot be achieved if your application includes some form of defensive programming. Common examples of this might include:

- A "default" clause in a switch statement in C (equivalently, a "when others" clause in an Ada case statement), where all of the cases represent the complete range of possible values. This might be required by a coding standard.
- Out of bounds checking where it can be formally proved that the bounds are never exceeded.
- Exception handlers automatically generated by the compiler.
- Built-in self-test operations, such as read-write memory tests. These are very difficult to test, as triggering the condition requires injecting the error exactly at the right time.

When developing software which features some form of defensive programming, you should be clear what you are defending against. In general, the purpose of defensive programming is to ensure that a piece of software degrades gracefully under unforeseen circumstances. These could include:

- Compiler-introduced errors.
- Hardware errors.
- Design errors.
- Incorrectly used interfaces (for example, passing a negative value in an integer parameter, where only positive values are expected).

One of the characteristics of defensive programming is that it is extremely difficult to set up test cases to trigger it. Consequently, defensive programming can result in incomplete code coverage.

What to do about it?

When defensive programming has led to incomplete code coverage, you will need to provide some form of justification why the defensive programming cannot be executed during tests.

You will also need to provide evidence (from reviews or special testing) that if it is ever triggered, the defensive programming works correctly.

[3]ISO 26262 Table 1

[4]RTCA DO-178C 4:5

2.6 Impossible combinations of events

In some places in your code there may exist situations where doing one thing automatically precludes doing another. For statement, decision, branch coverage, this will likely mean that there will be pairs of code blocks that cannot both be executed in a single run. A trivial example of this is an if-then-else structure. It's not possible to run both the `then` and the `else` in a single execution. Clearly, all that needs to happen here is to run multiple test cases and aggregate the results together.

In the case of MC/DC, code may include expressions where it simply isn't possible to achieve 100% MC/DC. The following example shows this:

```
-- "enabled" might already be set at this point
if speed < 1700 then
    enabled := true;
end if;
if speed < 1500 and enabled then
    ...
end if;
```

It isn't possible for `enabled` to independently affect the outcome of this final expression, because for `enabled` to be false, the first condition (speed < 1500) can never be true.

What to do about it?

Failing to achieve 100% MC/DC coverage in this situation usually indicates that your conditions could be simplified.

In some cases, simplification of the condition may not be possible, for example, because the code is automatically generated. If it is not possible to simplify the condition, you will need to provide some level of justification as to why 100% coverage could not be achieved.

2.7 Compiler introduced errors

It is possible that the compiler introduces an error into the generated object code in such a way that the test cases being executed pass, but do not achieve 100% coverage.

For example:

```
if a < 10 {
        result = x / 5;
} else {
        result = x / 10;
}
```

If the compiler incorrectly translates the conditional statement as `a <= 10`, test cases where `a` takes the values `3` and `10` and `x` takes the value `4` will pass correctly, but will not generate 100% statement coverage.

What to do about it?

In this kind of situation, the remedial action is likely to be much more wide-reaching than other actions we've discussed. You will likely need to do some of the following:

- Discuss the issue with the compiler vendor, and if feasible, change compiler to a version that does not contain this error.
- If it is not possible to change compiler version, you may need to conduct a review of your existing code base to ensure that the same error isn't triggered by other parts of your source code.
- Add to your coding guidelines/code review guidelines to ensure that the same error isn't added in subsequent developments.

3. Dealing with less than 100% coverage

As we have seen, if structural code analysis demonstrates less than 100% coverage, there are four main responses you can make, depending on the reason why coverage is incomplete:

- **Remove code**. Review shows that this code is unnecessary to the system, and you can remove it with no negative effects.
- **Add (or fix) tests.** Where tests (and possibly requirements) are incomplete, you need to develop new tests, which will improve the overall coverage level.
- **Justify why code will never be executed.** For deactivated code, you need to provide some form of justification why this code cannot be executed.
- **Justify why it is not possible to test code.** Through manual testing/review you should demonstrate that the code works correctly. For example, for defensive programming structures, this could involve setting a breakpoint just before the test for the defensive code, then manually forcing the test to fail, and executing the defensive code.

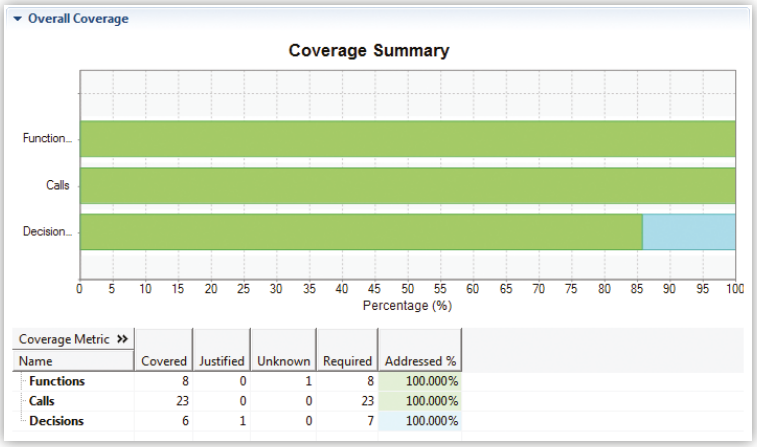


Figure 4 – Rapi**Cover** report showing 100% addressed coverage using justifications

What the standards say

ISO 26262 "If the achieved structural coverage is considered insufficient, either additional test cases shall be specified or a rationale shall be provided."

When code is justified, it is important to demonstrate that it has not been executed during testing. For example, deactivated code should never be executed, nor should code that is only accessible when the system is in a different mode. If we have said "this code will never be executed because XYZ" and the code is then executed, that represents an error in our justifications.

- When presenting your code coverage results, you should aim to present:
- The parts of the code you have executed through testing.
- The parts of the code you have not executed, but have provided some form of justification for.
- The detailed justification for each piece of code that was not executed.

How can I make it easier?

Rapita's code coverage tool, Rapi**Cover**, offers the ability to add a justification indicating the reason why a section of code was not executed during the system testing. Sections of code to which justifications have been applied are identified in the GUI and in exported text reports (which are typically used as part of the certification case). A justification can be set up once and reused for every test – which is a valuable reduction in effort when preparing coverage reports for each release.

There are two ways in which justifications can be applied to source code:

1. They can be added into the source code at the location of the code they refer to. This makes it straightforward to see which sections of the source code are expected to remain uncovered during testing.
2. A separate file containing justifications for specific sections of code can be supplied. This is useful if different tests exercise the code in different ways. For example, a set of justifications could be applied to tests on a per-mode basis.

To ensure that justifications are not incorrectly used when a software version changes, Rapi**Cover** also allows the justifications to be tagged with a build or version identifier.

Rapi**Cover** will issue a warning when it encounters any code sections that are both covered and justified, because this may indicate a flaw in the test or any assumptions made about the operation of the software.

4. About Rapi**Cover**

RapiCover** is a structural coverage analysis tool designed specifically to work with embedded targets.**

Rapi**Cover** delivers three key benefits:

- Reduced timescales by running fewer on-target tests.
- Reduced risk through greater tool flexibility.
- Reduced effort for certification activities.

4.1 Reduced timescales by running fewer on-target tests

Running system and integration tests can be time-consuming and runs the risk of introducing schedule delays, especially if the availability of test rigs is limited. Most commercially available coverage solutions have large instrumentation overheads. As system resources are typically limited, obtaining coverage with these older coverage solutions typically requires multiple test builds. This takes longer to complete the testing program, especially if you need to negotiate additional time on test rigs to perform the extra tests.

Rapi**Cover** is designed specifically for use in resource-constrained, embedded applications. Because there is considerable variation between embedded systems, both in their requirements and their underlying technology, Rapi**Cover** provides a range of highly-optimized solutions for the instrumentation code it generates. This flexibility allows you to make the best use of the resources available on your platform.

This results in best-in-class instrumentation overheads for an on-target code coverage tool, and consequently fewer test builds.

4.2 Reduced risk through greater tool flexibility

An early design objective for Rapi**Cover** was to make it easy to deploy into any development environment, whether they be highly customized, extremely complex or legacy systems.

The two key factors to consider in a deployment of a coverage tool are: build system integration and coverage data collection.

Build System Integration

Rapi**Cover** is designed to work with any combination of compiler (C, C++ or Ada), processor and real-time operating system (RTOS). Access to command-line tools and the ability to choose between recommended strategies for integrating RapiCover into pre-existing build systems ensures a seamless integration.

Coverage Data Collection

Rapi**Cover** is designed with the flexibility to handle data from a wide variety of possible sources. This flexibility means that when creating an integration with a specific target, you can select the most convenient collection mechanism, including legacy approaches such as CodeTEST® probes. Figure 4 shows alternative data collection approaches.

To enable a rapid, high-impact integration into your development environment Rapita Systems provides the option of a target integration service. In this service, Rapita Systems' engineers will work with your team to establish an optimal integration into your development environment. This integration will be consistent with Rapita Systems' DO-178B/C and ISO 26262 tool qualification process, ensuring that tool qualification runs smoothly.

A Rapi**Cover** integration is based upon the R**VS** (Rapita **Verification Suite**) core toolflow. This makes it easy to extend the integration to support other R**VS** components such as Rapi**Time** (measurement-based worst-case execution time analysis) or Rapi**Task** (visualization of scheduling behavior).

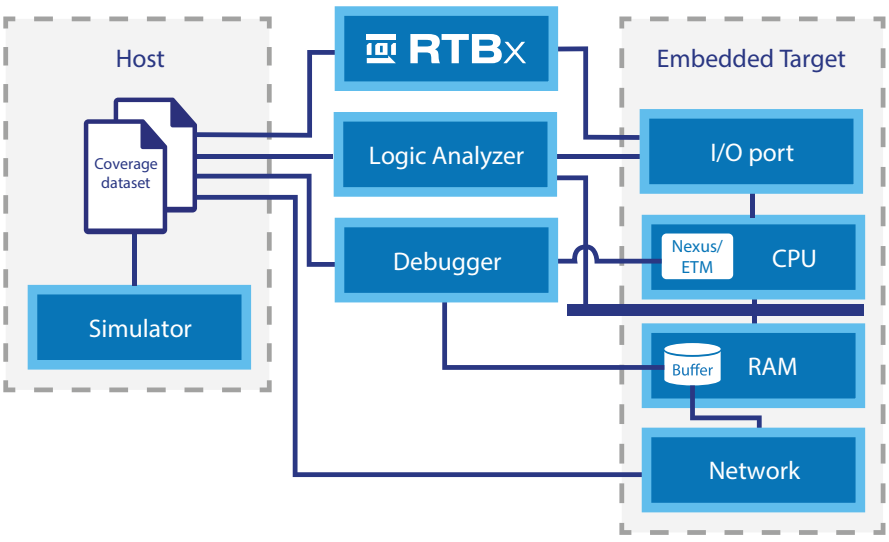


Figure 5 – Example Rapi**Cover** data collection approaches

4.3 Reduced effort for certification activities

Automatic combination of results from multiple test runs and the ability to justify missing coverage makes the preparation of coverage software verification results quicker.

A major driver for the use of code coverage is the need to meet DO-178B/C objectives. In addition to providing options for achieving DO-178B/DO-330 tool qualification, Rapi**Cover** also aims to make the process of gathering and presenting code coverage results easier. This is achieved in the following ways:

Multiple format report export

Rapi**Cover** lets you view results in our using our Eclipse®-based viewer and and export your results into various formats including plain text, HTML, CSV or XML.

Combine reports from multiple sources

Coverage data is often generated at multiple phases of the test program, for example: unit test, integration test and system test. Rapi**Cover** supports the consolidation of this data into a single report.

Justify missing coverage

Where legitimate reasons exist that specific parts of the code cannot be executed, Rapi**Cover** lets you justify them. The summary report shows code that is executed, code that is justified and code that is neither executed nor justified.

To facilitate your use of Rapi**Cover** within a DO-178B/C project, we provide several options for tool qualification:

Qualification Data

This gives you access to documents necessary to support tool qualification of Rapi**Cover**.

Qualification Kit

In addition to the qualification data, this provides test code and supporting framework that enables you to generate evidence that Rapi**Cover** works correctly on your own system.

Qualification Service

Engineers from Rapita Systems work with you to apply the Rapi**Cover** tests to your system and to develop the necessary qualification arguments for your certification case.

5. Want to learn more?

If you want to learn more about code coverage, visit our website where you gain access to a wide range of white papers and videos about the topic.

www.rapitasystems.com/code-coverage-ada-c-cplusplus

Rapita Systems regularly releases new material and runs training courses on multicore timing analysis worldwide. To make sure you're notified, sign up to our mailing list.

www.rapitasystems.com/newsletter

RAPITA Systems
A DANLAW Company

About Rapita

Rapita Systems provides on-target software verification tools and services globally to the embedded aerospace and automotive electronics industries.

Our solutions help to increase software quality, deliver evidence to meet safety and certification objectives and reduce costs.

Find out more

A range of free high-quality materials are available at:
rapitasystems.com/downloads

SUPPORTING CUSTOMERS WITH:

Tools	Engineering Services	Multicore verification
Rapita **Verification Suite**:	V&V Services	**MACH**[178]
Rapi**Test**	Integration Services	Multicore Timing Solution
Rapi**Cover**	Qualification	
Rapi**Time**	SW/HW Engineering	
Rapi**Task**	Compiler Verification	

Contact

Rapita Systems Ltd.
Atlas House
York, YO10 3JB
UK
+44 (0)1904 413945

Rapita Systems, Inc.
41131 Vincenti Ct.
Novi, Mi, 48375
USA
+1 248-957-9801

Rapita Systems S.L.
Parc UPC, Edificio K2M
c/ Jordi Girona, 1-3
Barcelona 08034
Spain
+34 93 351 02 05

rapitasystems.com

linkedin.com/company/rapita-systems

info@rapitasystems.com