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

White Paper

Structural coverage analysis (SCA – also referred to as code coverage) is an important component of critical systems development. Many standards/ guidelines including DO-178C (in aerospace) and ISO 26262 (in automotive) recommend or mandate the use of coverage analysis for measuring completeness of testing. In some situations, however, it is difficult to achieve complete (100%) coverage. What are these situations, and what can we do about them? In this white paper, we review the causes of being unable to attain 100% code coverage during testing, and will identify four strategies for handling code that has not achieved full coverage.

On-target software verification tools

RAPITA
SYSTEMS LTD

Contents

Introduction **3**

What is code coverage for? 3

What does it mean to get 100% code coverage? 4

Does it matter which code coverage criteria I'm using? 5

What are the barriers to 100% code coverage, and what can I do about it? **6**

Missing requirements 7

Missing or incorrect tests 7

Dead code 8

Deactivatived code 8

Defensive programming 9

Impossible combinations of events 10

Compiler introduced errors 11

Dealing with less than 100% coverage **12**

About Rapi**Cover** **14**

Reduced timescales by running fewer on-target tests 14

Reduced risk through greater tool flexibility 15

Reduced effort for certification activities 16

About Rapita Systems **17**

RVS 17

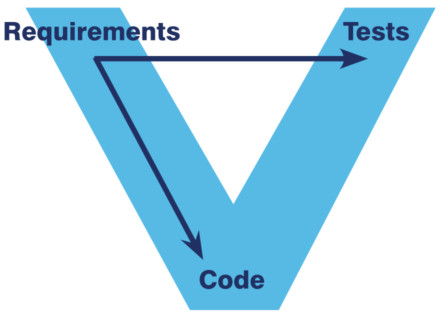
Early Access Program 17

Introduction

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.

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, below) results in the degree of rigor applied in testing the code to reflect the Development Assurance Level (DAL) of the system.

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.

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.

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 ISO26262) that you are using.

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**
Statement 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**

Table 1: Coverage requirements by standard/guideline

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, with one exception, which applies to MC/DC (see "Impossible combinations of events", below).

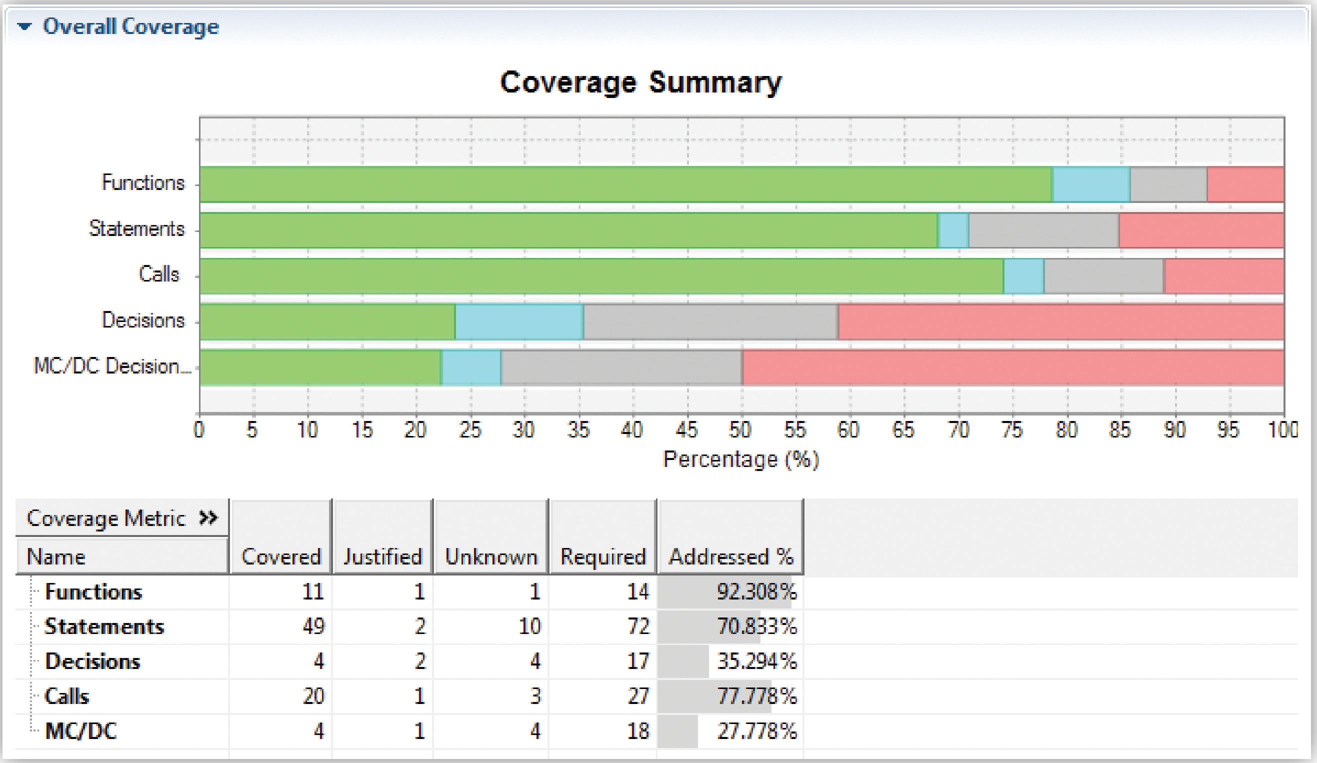
What are the barriers to 100% code coverage, and what can I do about it?

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 1: Rapi**Cover** report showing incomplete coverage

Missing requirements

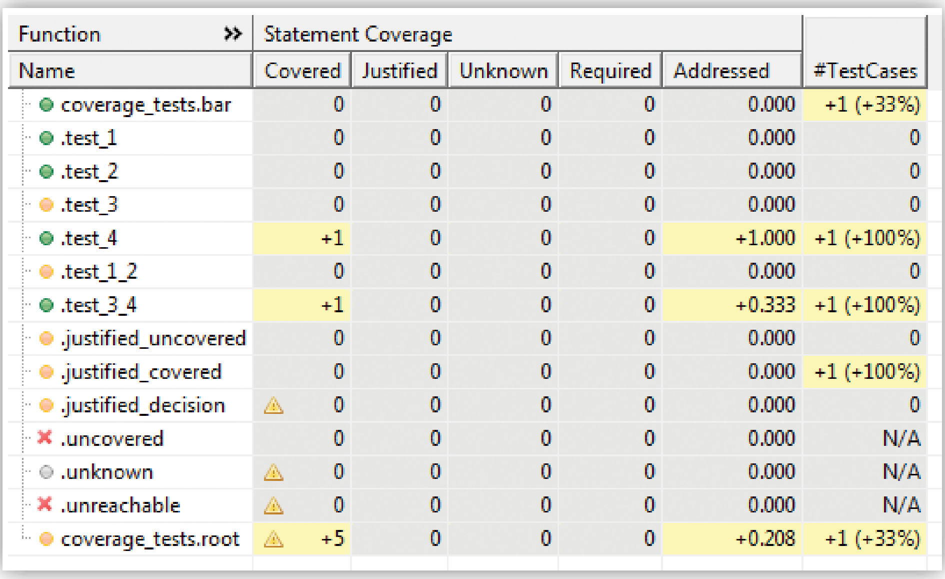
Identifying missing requirements is one of the primary reasons for performing code coverage.

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** includes a coverage report comparison feature (see for example, Figure 2), which will allow you to show that the new tests cover code that was not previously covered.



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 2: Rapi**Cover** coverage report comparison

Missing or incorrect tests

Your requirements may define specific functionality, which corresponds to a particular piece of code. However, if there are no tests which drive that particular piece of code, 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. Like missing requirements, the use of a coverage comparison feature is useful in this situation.

Dead code

According to DO-178C, **dead code** falls within the category of extraneous code, which is code that is not traceable to a system or software requirement.

The DO-178C definition of dead code (see sidebar) contrasts somewhat with other definitions of dead code, for example Wikipedia: "… dead code is a section in the source code of a program which is executed but whose result is never used in any other computation."

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]

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]

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?

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.

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.

"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 [ISO 26262-6, Table 1]

DO-178C: Defensive programming practices may be considered to improve robustness [DO-178C Section 4.5]

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, there may exist expressions where it simply isn't possible to achieve 100% MC/DC. A trivial 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 why 100% could not be achieved.

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.

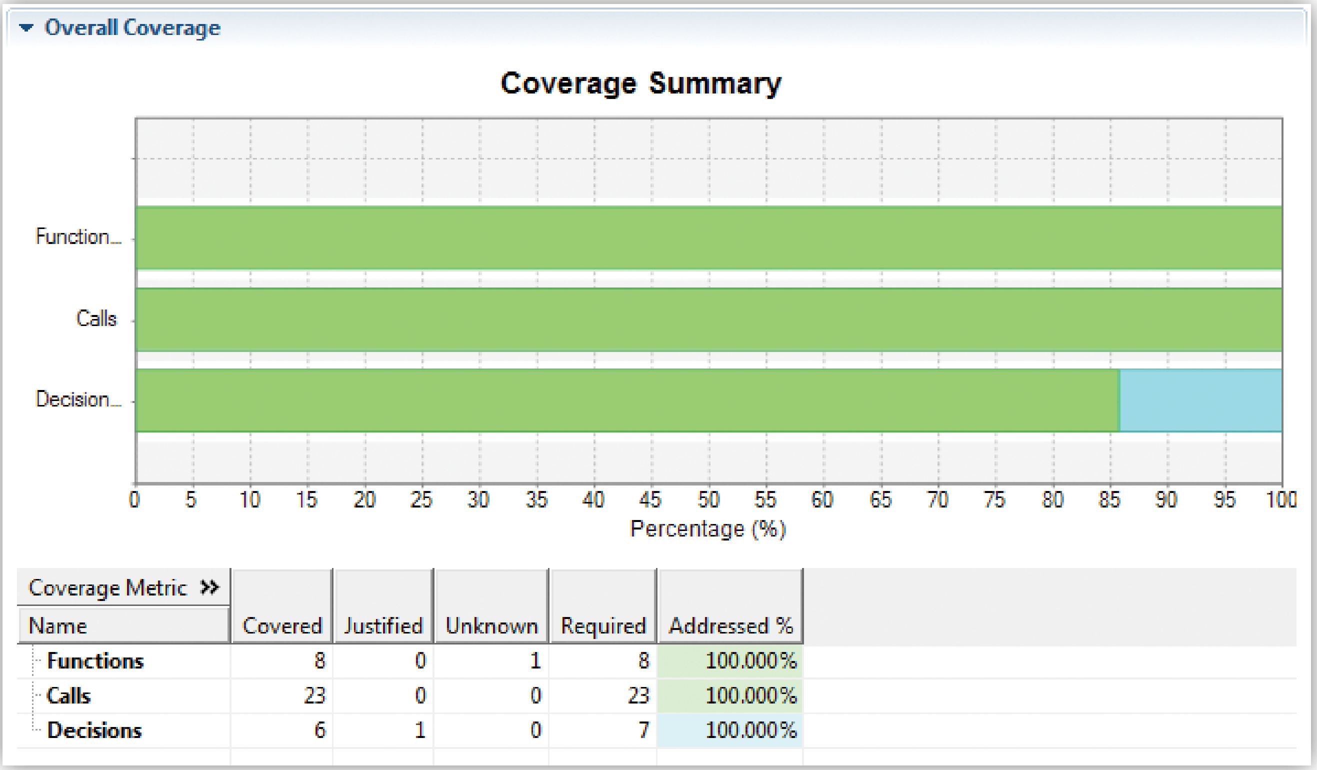
Dealing with less than 100% coverage

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.

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."



Name	Covered	Justified	Unknown	Required	Addressed %
Functions	8	0	1	8	100.000%
Calls	23	0	0	23	100.000%
Decisions	6	1	0	7	100.000%

"Figure 3: Rapi**Cover** report showing 100% addressed coverage using justifications

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 that easier?

Rapita Systems' 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.

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.

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.

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 Rapi**Cover** 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 RVS (Rapita Verification Suite) core toolflow. This makes it easy to extend the integration to support other RVS components such as RapiTime (measurement-based worst-case execution time analysis), RapiTask (visualization of scheduling behavior) or newer developments based upon Rapita Systems' Early Access Program.

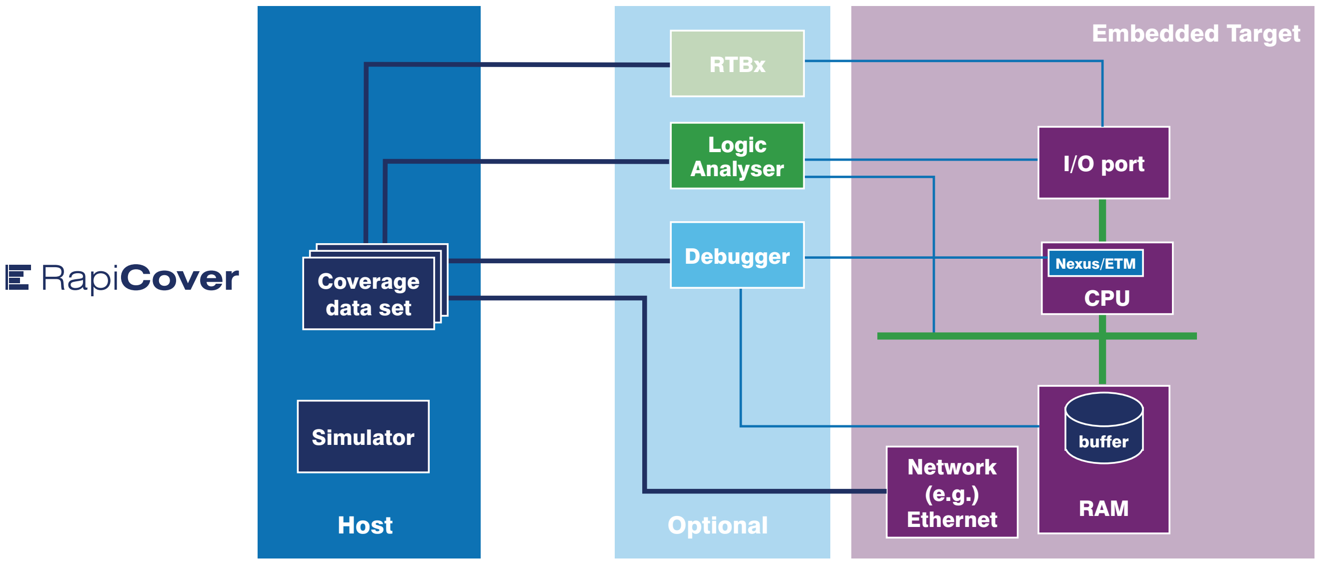


Figure 4: Alternative data collection approaches

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** provides you with the ability to browse coverage data using our eclipse-based viewer and to export the same information into CSV, text, XML or aligned with source code.

Combination of 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.

Justification of missing coverage.

Where legitimate reasons exist that specific parts of the code cannot be executed, Rapi**Cover** provides an automated way of justifying this. 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.

About Rapita Systems Ltd.

Founded in 2004, Rapita Systems develops on-target embedded software verification solutions for customers around the world in the avionics and automotive electronics industries. Our tools help to reduce the cost of measuring, optimizing and verifying the timing performance and test effectiveness of their critical real-time embedded systems.

RVS

RVS (Rapita Verification Suite) provides a framework for on-target verification for embedded, real-time software. It provides accurate and useful results by observing software running on its actual target hardware. By providing targeted services alongside RVS, Rapita Systems provides a complete solution to customers working in the aerospace and automotive industries.

RVS helps you to verify:

- Software timing performance (Rapi**Time**);
- Structural code coverage (Rapi**Cover**);
- Scheduling behavior (Rapi**Task**);
- Other properties (via Rapita Systems' "Early Access Program").

Early Access Program

We participate in many collaborative research programs, with a large variety of organizations. This results in our development of a wide range of advanced technologies in various pre-production stages.

Rapita Systems' customers have found access to this technology has been very useful.

Working with us in our Early Access Program gives you the ability to use the latest technology for your specific needs. Access to this technology is normally provided through defined engineering services and gives you the opportunity to influence the development of the technology into a product.

Early Access Program examples

Examples of technologies available in Rapita Systems' Early Access Program include:

- **ED4i.** Automatic generation of diverse code for reliability.
- **RapiCheck.** Constraint checking of code running on an embedded target.
- **Data dependency tool.** Supports the conversion of sequential code for multicore targets.

Atlas House
Osbaldwick Link Road
York
YO10 3JB
United Kingdom

Tel No: +44 1904 413945
Email: enquiries@rapitasystems.com
Website: **www.rapitasystems.com**

Registered in England & Wales
5011090