Eight top code coverage questions for DO-178B/C

White Paper

To meet DO-178B/C guidance, testing of airborne software should be supported with structural code coverage measurements. This paper sets out eight key code coverage questions for engineers working on embedded avionics systems. It then introduces RapiCover, which is optimized for on-target structural coverage analysis. RapiCover helps meet DO-178B/C guidelines, reduces verification effort and supports engineers working with C, C++ and Ada.

On-target software verification solutions

RAPITA SYSTEMS
A DANLAW COMPANY

Contents

1. Introduction

Supporting the test process with measurements of structural code coverage is a key activity for DO-178B/C compliance during the development of software for airborne systems. In this white paper we consider eight key questions:

- What is code coverage and how does it benefit my project?
- Should we do on-target or on-host coverage?
- What are the challenges to on-target code coverage and how can we overcome them?
- How can I use my code coverage results to support certification?
- What additional benefits are derived from measuring on-target?
- How do I combine results from multiple tests?
- How do I deal with missing code coverage?
- What should I look for in a code coverage tool?

After we have addressed these eight key questions, we introduce Rapi**Cover**, a software tool designed to efficiently perform structural coverage analysis on code running on an embedded target. The benefits of using Rapi**Cover** to conduct structural coverage analysis on an embedded target include:

- **Reduced timescales** by running fewer on-target tests. Very lightweight instrumentation means more coverage information per test cycle.
- **Reduced risk** through greater tool flexibility. Adapt Rapi**Cover** to work with your system, rather than adapting your system to work with another tool. Collect coverage information via a wide variety of mechanisms, making it easier to integrate Rapi**Cover** into your system.
- **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 quicker.

Keep up to date

The Rapita Systems blog addresses topics related to on-target verification, including code coverage and DO-178B/C.

www.rapitasystems.com/blog

2. Eight top code coverage questions

2.1 What is code coverage and how does it benefit my project?

Structural coverage analysis is an important verification tool for establishing the completeness of testing.

DO-178B/C emphasises the use of requirements-based testing as an important part of the software verification process. In requirements-based testing, the high and low-level requirements are used to derive source code and the tests for that source code. Traceability between the requirements, the test cases and the source code demonstrates:

- Every requirement has a test case.
- All source code is traceable to a requirement.

Measuring code coverage when the test cases are executed is essential for this process – where coverage is less than 100%, this points to code that is not traceable to requirements, tests or both.

Different coverage criteria (see table on p.5) allow the degree of rigor in measuring the coverage to reflect the Development Assurance Level (DAL) of the system.

What to look for in a code coverage tool:

Can it support all classes of code coverage? Can it support different variants such as masking v. non-masking MC/DC?

DO-178B/C and code coverage

RTCA DO-178B/C (also referred to as EUROCAE ED-12B) provides guidance for specific considerations for airborne software. It calls for demonstration of code coverage to a level determined by the criticality of the application under consideration. The table (right) lists a number of coverage criteria used to assess software testing effectiveness. The coverage criteria are defined in the Appendix (p.16).

Measurement	Description	Notes
Function coverage	Each function has been called at least once	Not required by DO-178B/C
Call coverage	Each function has been called at least once, and each different function call has been encountered at least once	Not required by DO-178B/C
Statement coverage	Each statement in the code has been encountered at least once	Required for DO-178B/C Level A, B, C
Decision coverage	Each decision (see box below) in the code has evaluated true at least once and evaluated false at least once, and each function entry and exit point has been encountered at least once	Required for DO-178B/C level A, B
Condition coverage	Each condition (see box below) in the code has evaluated true at least once and evaluated false at least once	Not required by DO-178B/C
Modified Condition/ Decision Coverage	Decision coverage plus each condition has been shown to independently affect the outcome of its enclosing decision	Required by DO-178B/C Level A

Where code is well-structured and derived directly from well-written requirements and architecture, then a full set of high- and low-level requirements-based tests is entirely capable of meeting many of the existing code coverage criteria.

What are conditions and decisions?

A condition is a Boolean expression that contains no Boolean operators. Examples of conditions are: "`true`", "`iterations > 5`" or the name of a Boolean variable, such as "`MaintenanceMode`". If the same Boolean expression is repeated several times, each specific instance is a different condition.

A decision is a combination of at least one condition with zero or more Boolean operators to create an overall Boolean expression.

To illustrate the differences, consider:

```
if (A) {                        // "A" is a condition and a
     ...                        // decision

} else if (B && x < 14) {       // "B" is a condition,
                                // "x < 14" is a condition,
                                // "B && x < 14" is a decision
     A = !(x > 14);             // "x > 14" is a condition
                                // "! (x > 14)" is a decision
```

2.2 Should we do on-target or on-host code coverage?

When developing software for an embedded application, such as an avionics system, verification activities can be performed **on-host** or **on-target**. On-target testing means the application is tested on the hardware to be deployed (the target). It may also be referred to as host-target testing or cross-testing. On-host testing means testing the application on a host computer (such as the development system used to build the application). This may also be referred to as host-host testing.

2.2.1 On-target testing

The key principle behind testing an application on-target is that code is executed in the environment for which it was designed, rather than in an environment where it was never intended to be executed. Test results are typically evaluated and analysed on a host. This has the following benefits:

» The "credibility gap", the possibility that some unanticipated difference exists between executing on-host in a harness and executing on-target, is minimized. This results in a lower likelihood of false-negative errors leading to errors not being detected, and faulty software being deployed, and of false-positive reports where time is wasted tracking down non-existent problems.

» The smaller "credibility gap" also makes it easier to provide an argument to certification authorities that testing achieves an appropriate level of rigor.

» Ability to execute all code. Some parts of your code might not be possible to run on host (for example, device specific code).

2.2.2 On-host testing

On-host testing involves compiling the application code to run on the host processor, rather than the target processor. Typically the application also requires a certain amount of adaptation to work in the host environment due to the following considerations:

» Running under a desktop OS rather than the target's RTOS may require different API calls.

» If the embedded application includes libraries, these may not be available on the host (or may be different, for example, in the case of graphics libraries).

» Alternative interpretations of ambiguous/undefined programming language features or compiler bugs may cause different behaviors between the host and the target.

» The embedded application may require access to specific hardware features that are not available on the host system.

However, there are benefits that can arise from on-host testing:

» The target may not be available, or there may be only limited access to it when testing needs to take place.

» The "build-deploy-analyze" cycle may be quicker than on-target testing.

» It is well suited to unit testing – a test harness can be used to achieve 100% coverage, even when defensive programming techniques are used.

What to look for in a code coverage tool:

Can it do on-host and on-target testing?

2.2.3 What to choose?

The choice between on-host and on-target testing is driven by a trade-off between cost/convenience and credibility of results.

In many cases, using a combination of both techniques offers dual benefits:

- Unit testing and test case development on-host gives the advantages of rapid turnaround;
- System/integration testing on-target provides the confidence that the code to be deployed has been tested in its intended environment.

2.3 What are the challenges to on-target code coverage, and how can we overcome them?

One of the biggest challenges to on-target code coverage is resource limitations in embedded systems. The standard approach to measuring coverage is to instrument source code to write tags into a memory buffer.

This approach evolved from host-based testing – it is clear that many commercially available code coverage solutions today begin with a host-based approach and attempt to transfer it to an embedded environment. This approach requires a large RAM buffer to store the data in, and each instrumentation point requires a large number of instructions (increasing execution time and increasing code size). On a resource-constrained platform this represents a difficulty.

The exact nature of resource constraints varies between systems. Data areas might be limited or code size constrained. On other systems high CPU utilization might limit what could be achieved. A code coverage solution has to recognize these limitations can exist, and to provide a viable route to dealing with them.

There are a number of ways to address resource constraints:

- Alternative data collection. In many cases, using an in-memory data structure to record coverage will be sufficient. However, when there is not enough room to store this data structure, or if the execution overhead of this approach is too high, alternative approaches need to be available. One such approach involves recording a *trace* of instrumentation points via an I/O port. This avoids the need for a large area of memory and simultaneously makes instrumentation overheads very low, typically 1-2 machine instructions. Advanced debuggers (e.g. Nexus or ARM ETM-based tracing debuggers) can also be used to collect data.

» Partial instrumentation. Rather than completely instrumenting an application, instrument specific parts of it, perform the tests and combine the results to provide an overall picture.

» Optimized instrumentation. Measuring certain types of coverage, for example MC/DC, can require significant memory overheads. Instrumenting an embedded system for coverage requires knowledge of how instrumentation is carried out. Once set up, there are opportunities to make trade-offs between exactly how the level of coverage is achieved, and the amount of instrumentation required.

What to look for in a code coverage tool:

Can it adapt to different embedded environments? Can it cope with low memory environments? Will it support partial instrumentation and provide the ability to combine results?

2.4 How can I use code coverage results to support certification?

If evidence of code coverage is mandatory for the project, for example because the customer requires strict adherence to DO-178B/C guidance, it's also important to be able to provide evidence that the process used to collect the data has worked correctly.

The evidence must show that the tool works correctly within the context of the development environment for which it is producing results. In the case of DO-178B/DO-330, the following items are recommended for tool qualification:

» PSAC (Plan for Software Aspects of Certification). This references the TQP and TAS (see below).

» TOR (Tool Operational Requirements). This describes what the tool does, how it is used and the environment in which it performs.

» TAS (Tool Accomplishment Summary). This is a summary of the data showing that all requirements in the TOR have been verified.

» TVR (Tool Verification Records). This comprises test cases, procedures and results.

» TQP (Tool Qualification Plan). This describes the process for qualifying the tool.

These items combine two main kinds of evidence:

» Generic evidence. This needs to be provided by the tool vendor to define the tool operational requirements, and verification evidence to demonstrate that the tool meets the requirements.

» Specific evidence. The tool user needs to demonstrate that the tool works correctly in a specific environment. Ideally the tool vendor should provide support to simplify this process as much as possible.

What to look for in a code coverage tool:

Is certification evidence available? Will the tool vendor support you in generating specific certification evidence?

2.5 What additional benefits come from measuring on-target?

When you instrument source code and run your application on target, you are opening the door to collecting other information besides simply code coverage. For example, if you collect a *trace* (i.e. recording the sequence of instrumentation points that are executed), it is possible to identify which test cases execute specific execution paths. Using the traces, it is possible to step through the code forwards and backwards.

If a trace also records the specific time at which instrumentation points are executed, it is possible to determine timing information. For example, RapiTime uses such information to provide a wide range of timing measurements that can be used for:

- execution time measurement;
- worst-case execution time (WCET) calculation;
- performance optimization.

What to look for in a code coverage tool:

Can it exploit the effort that you've put in to integrate it with your target to provide additional information?

2.6 How do I combine results from multiple tests?

Your approach to testing may rely upon combining coverage results from a variety of different tests. This could occur because:

- Your strategy includes upon a combination of on-target and on-host testing.
- You need multiple test cases reflecting different system modes.
- Possibly system constraints force you to instrument one only part of your system at a time (consider the advice in Section 2.3 to mitigate this issue).

It may be necessary to perform the coverage analysis for each of these tests individually, and to manually merge the results.

A better approach is to use a tool that supports the combination of multiple results into a single report.

What to look for in a code coverage tool:

Can it combine coverage data from multiple test scenarios into a single report?

2.7 How do I deal with missing code coverage?

In some situations, there may be legitimate reasons for not achieving 100% code coverage.

For example, it might not be possible to construct test cases to execute defensive programming constructs. In this case, alternative forms of verification of this code could be agreed upon as acceptable.

In such a situation, it is useful for any code coverage report to provide the ability to justify uncovered code. Summary reports could then show executed code, justified (but unexecuted code) and unjustified code. The objective should be for all code to be either justified or executed.

What to look for in a code coverage tool:

Is it possible to justify why some code is not executed, and to report the proportion of executed code, justified code and neither executed nor justified code?

2.8 What to look for in a code coverage tool

Summarizing the above, which represents knowledge collected from our work with avionics software teams in the aerospace industry, we see that a code coverage tool should:

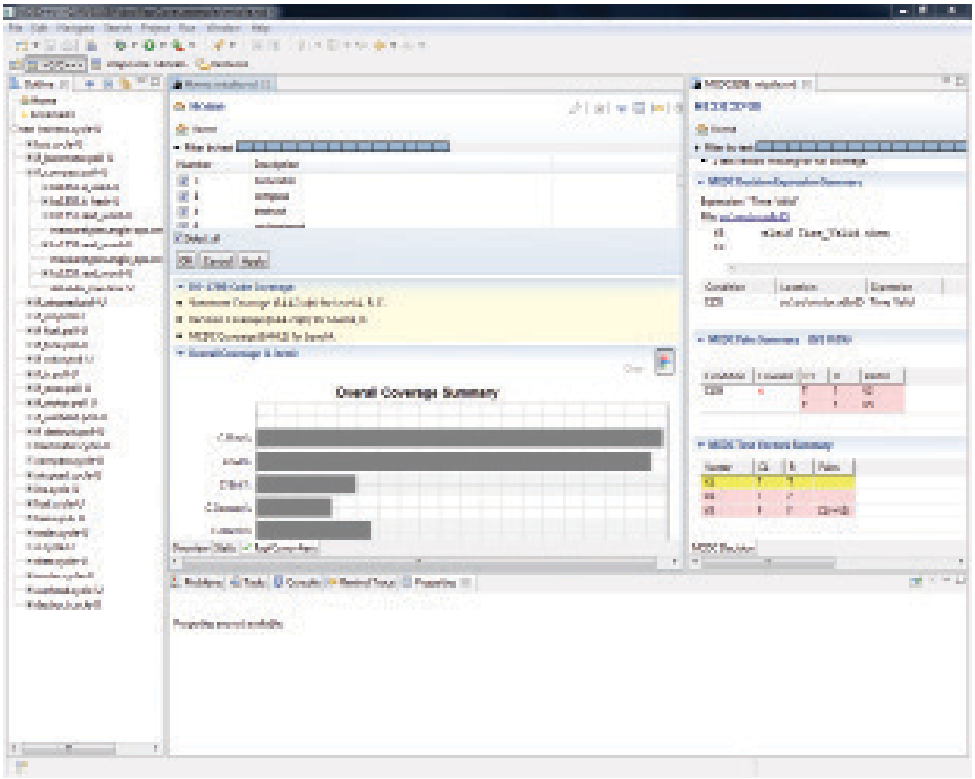
- support all classes of code coverage, including the specific interpretations used by your project;
- be capable of supporting on-host and on-target testing;
- be suitable for different embedded environments, including low memory environments;
- support partial instrumentation;
- have certification evidence available;
- be able to collect other classes of information;
- combine coverage reports from different tests;
- support justifications for code that has not been executed.

3. Product Summary: RapiCover

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

RapiCover is designed to deliver three key benefits:

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



RapiCover screen shot

3.1 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. If instrumentation overheads for code coverage are large, and system resources are limited, obtaining coverage can only be achieved through multiple test builds. This increases testing time, especially if additional time on test rigs needs to be negotiated.

RapiCover is designed specifically for use in resource-constrained, embedded applications. Because

What should I look for in a code coverage tool?	RapiCover
Support all classes of code coverage	✓
Including both masking and unique case MC/DC	✓
Support on-host and on-target testing	✓
Suitable for low memory embedded environments	✓
Support partial instrumentation	✓
Have certification evidence available	✓
Ability to collect other classes of information	✓
Combine multiple coverage reports	✓
Justify non-executed code	✓

there is considerable variation between embedded systems, both in their requirements and their underlying technology, RapiCover 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.

About instrumentation

Performing structural code analysis requires some way of identifying which parts of the code have been executed. One of the most widely-used approaches for this is source-code instrumentation. In this approach, instrumentation code is inserted into the source code during the build process. The instrumentation code is used to signal that a specific function, line, condition or decision (depending upon the coverage type required) has been executed. Done in a naïve way, this can negatively impact the executable code in two ways:

» Too many instrumentation points. Adding more instrumentation points than necessary doesn't improve the information generated, but does result in greater memory requirements and longer execution times.

» High overhead for each instrumentation point. If the implementation of an instrumentation point is inefficient, this has a multiplicative effect on the overheads of the system.

3.2 Reduced risk through greater tool flexibility

Rather than adapting your system to work with another tool, you can adapt how RapiCover works with your system. RapiCover also helps this flexibility by working with a wide variety of data capture mechanisms.

An early design objective for RapiCover 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.

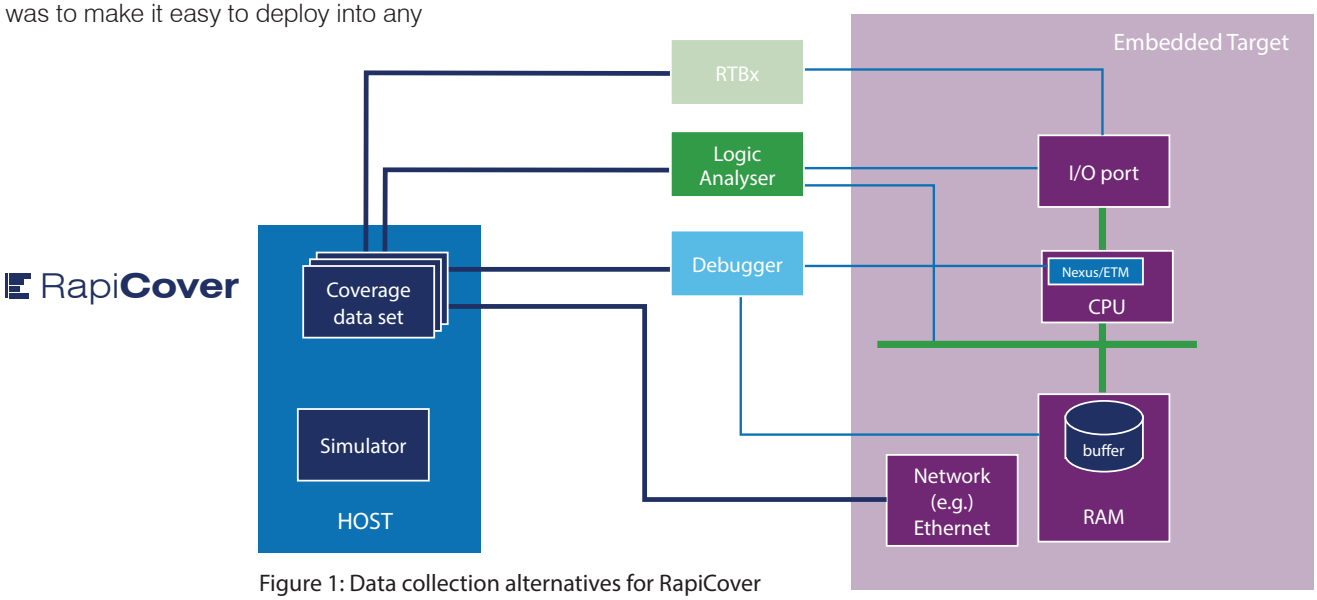


Figure 1: Data collection alternatives for RapiCover

» **Build System Integration.** RapiCover is designed to work with any combination of compiler (C, C++ or Ada), processor and real-time operating system (RTOS). Its use of command-line tools and the ability to choose between two alternative strategies for integrating RapiCover into pre-existing build systems ensures a seamless integration.

» **Coverage Data Collection** RapiCover 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 1 shows alternative data collection approaches.

To enable a rapid, high-impact integration into your development environment Rapita Systems provide 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 tool qualification process, ensuring that tool qualification runs smoothly.

A RapiCover 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 (see Section 4).

3.3 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, RapiCover 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.** RapiCover 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. RapiCover 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, RapiCover 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.

Text export of summary report

To facilitate your use of RapiCover 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 RapiCover.

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

» **Qualification Service.** Engineers from Rapita Systems work with you to apply the RapiCover tests to your system and to develop the necessary qualification arguments for your certification case.

3.4 Discover What RapiCover can do for you

» Contact us to find out more about RapiCover:
  » E: enquiries@rapitasystems.com

» Request a trial version to experience RapiCover for yourself:
  http://www.rapitasystems.com/trial

» To keep informed about RapiCover developments (and to receive other technical articles in the area of on-target verification), sign up for our monthly RapiTimes newsletter:
  http://www.rapitasystems.com/rapita/mailing_list

4. About Rapita Systems

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.

4.1 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").

4.2 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 our pre-production 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.

5. Appendix: overview of code coverage criteria

5.1 Function Coverage

Of the coverage levels discussed here, function coverage is the easiest to achieve. It demonstrates whether each function was called in some way during your tests. This level of coverage is a reasonable indicator that the tests have exercised a representative subset of the entire functionality of your system, without guaranteeing that every line of code has been executed during testing. Function coverage can reveal problems with dead code (which is an issue for DO-178B/C) or incomplete requirements-based testing.

It can be difficult to achieve full function coverage when working with generic or configurable components that may contain more functionality than that used by the specific application. When this happens, however, it is relatively easy to review the function behaviour and option selections to justify any omissions in function coverage.

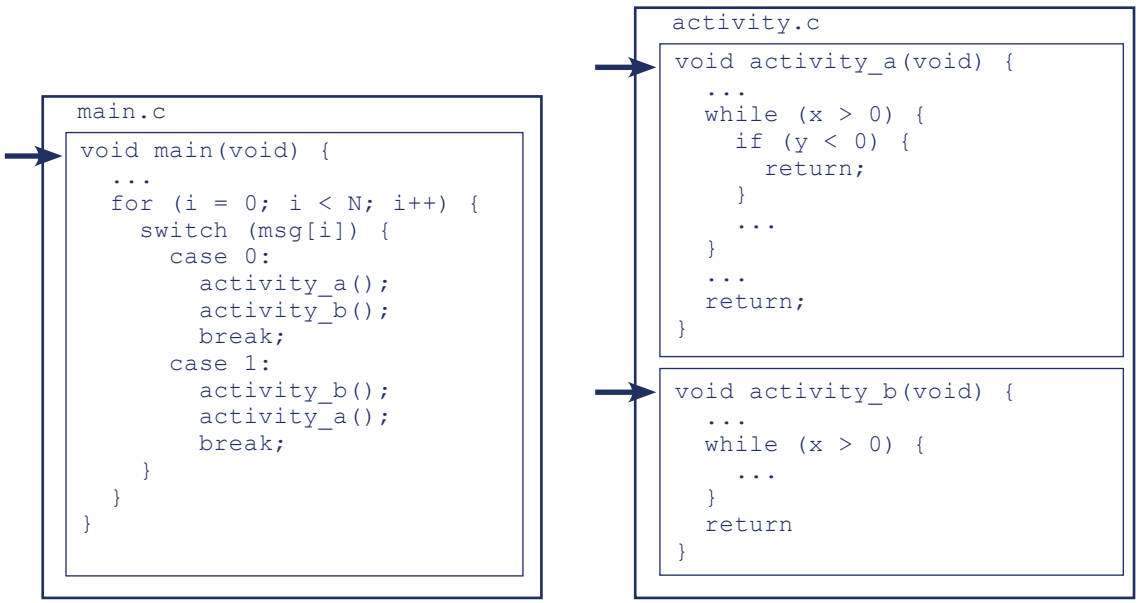
The example below contains three functions: `main`, `activity_a` and `activity_b`:

Function coverage of this program demonstrates:

- the program started;
- the for-loop executed at least once;
- at least one of the two switch-statement cases shown was taken.

Function coverage does not, however, reveal:

- whether both of the two switch-statement cases shown were taken;
- whether `activity_a` ever returned from within its own loop;
- whether `activity_b` ran any iterations of its loop.



5.2 Call Coverage

Call coverage represents a slight increase in complexity over function coverage. The term “call coverage” can actually be used to refer to two slightly different types of coverage:

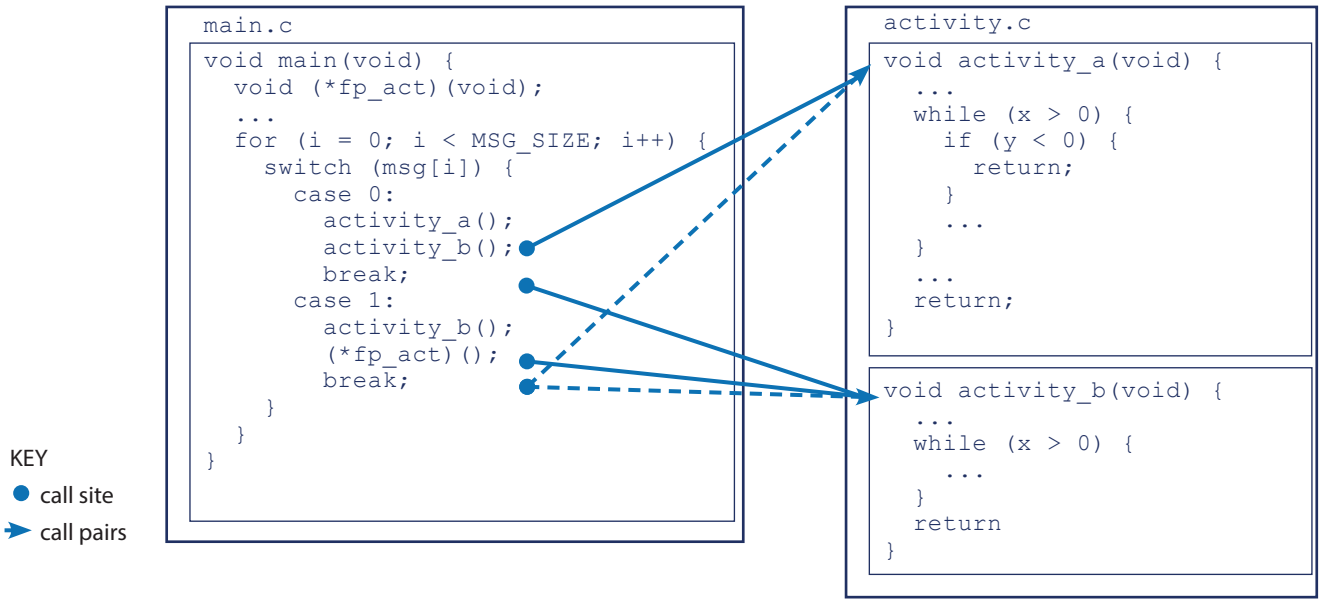
- **Call-pair coverage**. A call pair is the combination of a statement in one program unit (typically procedure, function or method) calling to another program unit (the callee). Call-pair coverage shows which of these pairs are exercised by a given set of tests.
- **Call site coverage**. A call site is the point in the program text from which the call is made. Call site coverage shows whether all such points have been exercised.

The two approaches are equivalent if each caller can only call one program unit, that is if no caller uses function pointers, dynamic dispatching or any similar method.

It is important to locate particular statements rather than performing the analysis at the level of entire program units, because there could be multiple calls made to a particular unit from another particular unit. The following figure shows an example of program structure and identified call pairs:

Support for call-pair coverage.

When function pointers are used, detecting which call sites are responsible for calling specific functions is difficult when using an “array of booleans”. A side effect of collecting a trace of instrumentation points is that call pair coverage between function pointers and functions can easily be detected.



5.3 Statement coverage

To achieve statement coverage, it is necessary for each statement in the source code to have been executed by at least one test in the test suite. If a particular statement cannot be covered, it is important to identify why. This may reveal dead code, for example, or it may be code that cannot be traced to a requirement or architectural structure.

Statement coverage is particularly useful when dealing with loops and returns. Consider our example code again:

Statement coverage reveals the answers to questions such as:

- Did every branch of the switch-statement get executed at least once?
- Did each while-loop run at least once?
- Did the code within each if-statement run at least once?

activity.c

```
void activity_a(void) {
  ...
  while (x > 0) {
    if (y < 0) {
      return;
    }
    ...
  }
  ...
  return;
}
```

```
void activity_b(void) {
  ...
}
```

In particular, for this program, statement coverage could determine whether testing was sufficient to show that the first return statement in `activity_a` was executed.

5.4 Decision coverage

Decision coverage criteria assess the ability of a set of tests to adequately exercise the routes through the logic of a program. They are derived solely from the structure of the code.

The code example contains two decisions. The first governs the while-loop at line four, and the second is the expression for the if-statement at line eleven.

For decision coverage, the typical criterion is that execution has reached every point of entry and exit in the code, and that for each decision in the source code that decision has resulted in each possible outcome (true, false) at least once. For the example code, this would mean:

- entry into function `sin_a_1000` reaches line 4;
- function `sin_a_1000` has exited on line 13 at least once;
- function `sin_a_1000` has exited on line 17 at least once;
- the expression governing the while-loop was true at least once, meaning that there was at least one non-sentinel entry in `Sin_Graph` and the loop body was executed;
- the expression governing the while-loop was false at least once, meaning that a sentinel entry was found in `Sin_Graph` and execution skipped to the bottom of the loop;

```
 1 int sin_a_1000 ( int v )
 2 {
 3    int Interpolate_Index = 0;
 4    while ( Sin_Graph[Interpolate_Index+1].x != Sin_Graph_Sentinel )
 5    {
 6       int x1 = Sin_Graph[Interpolate_Index].x;
 7       int y1 = Sin_Graph[Interpolate_Index].y;
 8       int x2 = Sin_Graph[Interpolate_Index+1].x;
 9       int y2 = Sin_Graph[Interpolate_Index+1].y;
10
11       if ( v >= x1 && v < x2 )
12       {
13          return (y1 + (v - x1) * (y2 - y1) / (x2 - x1));
14       }
15       Interpolate_Index++;
16    }
17    return Sin_Graph_Default;
18 }
```

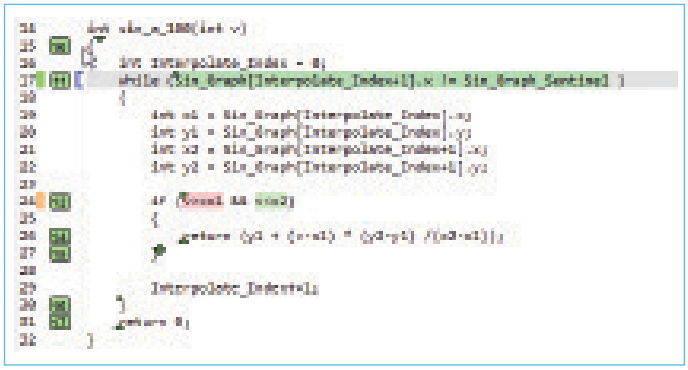
- the expression governing the if-statement was true at least once, meaning that the lookup value was located within one of the interpolation regions for at least one test;
- the expression governing the if-statement was false at least once, meaning that there is at least one run for which the lookup value was not in every region of `Sin_Graph`.

5.5 Modified condition/decision coverage (MC/DC)

Modified condition/decision coverage (MC/DC) extends decision coverage. Instead of just examining the outcome of each decision, the coverage check also shows that for each condition in the source code, that condition has resulted in each possible outcome (true, false) at least once, and also that each condition in a decision has been shown to independently affect that decision's outcome.

The additional checks for MC/DC for the example program show:

- The value of `v>=x1` has been true at least once.
- The value of `v>=x1` has been false at least once. In the below report, we see that this is not the case.
- The value of `v<x2` has been true at least once.
- The value of `v<x2` has been false at least once.
- The value of `v>=x1` independently affected the outcome of the whole expression, meaning that it has taken values of true and false while the value of `v<x2` was true.
- The value of `v<x2` independently affected the outcome of the whole expression, meaning that it has taken values of true and false while the value of `v>=x2` was true.



RAPITA SYSTEMS A DANLAW COMPANY

Rapita Systems Inc.
41131 Vincenti Ct.
Novi, MI 48375

Tel (USA):
+1 248-957-9801

Rapita Systems Ltd.
Atlas House, Osbaldwick Link Road
York , YO10 3JB
Registered in England & Wales: **5011090**

Tel (UK/International):
+44 (0)1904 413945

Email: **enquiries@rapitasystems.com** | Website: **www.rapitasystems.com**