Couverture: an Innovative Open Framework for Coverage Analysis of Safety Critical Applications

Matteo Bordin, Cyrille Comar, Tristan Gingold, Jérôme Guitton, Olivier Hainque, Thomas Quinot
AdaCore, 46 rue d'Amsterdam, 75009 Paris (France).
{bordin, comar, gingold, guitton, hainque, quinot}@adacore.com

Julien Delange, Jérôme Hugues, Laurent Pautet
Télécom ParisTech, 46 rue Barrault, 75634 Paris Cedex 13 (France)
{julien.delange, laurent.pautet, jerome.hugues}@telecom-paristech.fr

Abstract. *One key step in the development of safety-critical applications is the assessment of the quality of the verification strategy. In practice, structural coverage is the methodology used to ascertain the testing campaign well satisfy a given quality criteria. In this paper, we describe the possible strategies to measure structural coverage in a DO-178B context. After evaluating state-of-the-art approaches to measure structural coverage, we introduce Couverture, an innovative framework for coverage analysis exploiting a virtualized execution environment.*

Keywords: *structural coverage, DO-178B, MC/DC virtualization, Ada.*

1 Introduction

The development of a high-integrity application usually requires a close interaction between the design and testing phases. System requirements are decomposed into high-level architecture, then into atomic modules; the latter are then finally implemented, possibly using a third generation programming language such as Ada or C/C++. A set of corresponding verification activities mirrors each decomposition step in the design phase: for example, acceptance testing corresponds to system design, integration testing to architectural design and unit testing to module design. The *test cases* are derived from the *requirements* themselves: the idea is to have the specifications, rather than the implementation, drive the testing strategy. This is the V process model applied to software engineering (see fig. 1).

One key step in the V development process is the assessment of the quality of the testing strategy. For software systems, *structural coverage* [8] is the canonical approach in practice. Structural coverage is the analysis of how an application is exercised by a testing campaign. The basic idea behind coverage analysis is to gain confidence in the testing process by checking that the testing suite exercises all *meaningful* constructs present in the application in a *sufficiently extensive* way.

Several different coverage metrics exist [7], usually differing on the minimal number of tests necessary to reach full coverage. The most common are statement and decision coverage: the first measures which source code statements are exercised, while the second measures how Boolean expressions (decisions) are evaluated. Reaching a certain degree of coverage means that the requirement-driven testing strategy involves a set of tests whose execution exercises all the structures the metric targets. For example, reaching full statement coverage means executing at least once all statements present in the application source code; reaching full decision coverage

means that the tests execution caused all decisions (typically in if, case or while statements) to be evaluated to *both* outcomes (True and False). To provide a practical evidence of the difference between different testing metrics, we introduce here a very simple example which we will use extensively in the remainder of this paper. The source code of the example is the following:

```
1. procedure P (A, B, C : Boolean) is
2. begin
3.    if (A and then B) or else C then
4.       Do_Something;
5.    end if;
6. end P;
```

Listing 1 – Example Source Code

The example comprises two statements (lines 3 to 4) and a single decision (the Boolean expression at line 3). Statement coverage for procedure P can be reached with just one test causing the Boolean expression "`(A and then B) or else C`" to be evaluated to True. Decision coverage can be reached with two tests, one evaluating the decision at line 3 to True, the other to False.

Coverage analysis is explicitly required by several industrial standards: for example, DO-178B (civil avionics), ECSS-40 (space systems) or IEC-880 (nuclear). In this paper we concentrate on the DO-178B standard because it requires the most stringent coverage metrics to be applied.

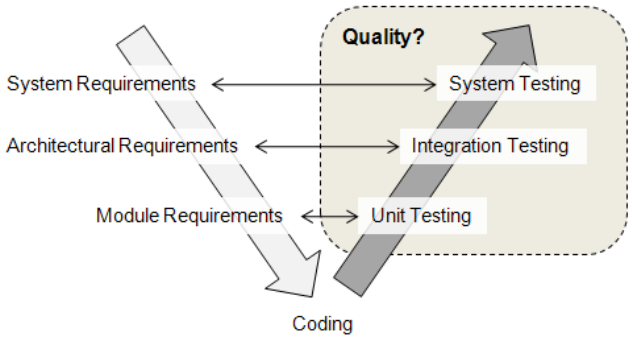

Figure 1 – The V process model

2 Structural Coverage and DO-178B

DO-178 [1] is an international standard providing guidance for the development of software to be deployed on airborne systems flying over civil ground. The standard applies to both Europe and North and South America. Developers of avionics software are expected to follow the guidelines contained in DO-178. They shall then submit the required evidence to certification authorities to gain certification credit for the software and allow its deployment on airborne systems. The DO-

178 standard has been reviewed in 1992, giving birth to DO-178B. The next revision of the standard, DO-178C, is expected by the end of 2010: the concepts contained in this paper apply to both DO-178B and DO-178C. DO-178B is complemented by DO-248B [2], which provides clarifications on the use of the DO-178B standard.

DO-178B describes a set of development and verification activities which shall be performed by a development team in order to gain certification credit for the developed software. Among these activities, structural coverage is part of the software verification process (see [1], table 7). Within DO-178B, structural coverage serves both to assess the quality of the verification strategy and to demonstrate the absence of *unintended functions*: if a part of the application is not covered by a requirement-driven testing strategy, it is likely that the implementation did not proceed from the requirements. Unintended functions shall not be deployed on the final applications, as they are not justified by any requirements, nor they are likely to have been thoughtfully tested.

Three different metrics are considered within DO-178B: Statement Coverage (SC), Decision Coverage (DC) and Modified Condition/Decision Coverage (MC/DC). All of these metrics express coverage in terms of source code elements: notions such as statement, decision, condition are defined only in relation to source code. We have already introduced SC and DC. We briefly illustrate MC/DC here: more information can be found on [5,6]. MC/DC distinguishes between decisions and conditions; a decision is, exactly like in DC, a Boolean expression. a condition is an atomic Boolean expression: conditions are grouped in decisions . In the example in listing 1, "`A`", "`B`" and "`C`" are conditions grouped in the decision "`(A and then B) or else C`". To achieve MC/DC, every point of entry and exit in the program must be invoked at least once, every decision must take all possible outcomes at least once, every condition in a decision must take all possible outcomes at least once, and each condition in a decision should be shown to independently affect the outcome of that decision [1]. Two evaluation vectors are required to demonstrate independent effect of a condition C on a decision D. The vectors shall be identical but for the value of C and one vector must cause D to be evaluated to True, the other to False. For example, to reach MC/DC for the source code in listing 1, we can use the following evaluation vectors for the single decision: (T,T,F), (T,F,T), (T,F,F), (F,T,F). Independent effect of "A" is demonstrated by (T,T,F) and (F,T,F): the two vectors are identical but for the value of "A" and the decision is evaluated first to True, then to False. Independent influence of "B" is demonstrated by (T,T,F) and (T,F,F); independent influence of "C" is demonstrated by (T,F,T) and (T,F,F). MC/DC requires at least n+1 tests to cover a decision composed by n independent conditions [5].

Within DO-178B, the metric chosen for structural coverage depends on the criticality of the application: the more safety-critical the application, the more stringent the metric (i.e. more tests are required to achieve the coverage objective). The aim is to require a more extensive testing campaign for the most safety-critical software applications. The effort required to reach a given coverage shall however be reasonable. For example, path coverage (the execution of all possible execution paths) exercises the application more than MC/DC, but it also requires an unreasonable amount of tests. The correspondence between coverage metrics and criticality level is as follows:

- Level A: MC/DC. Applications whose criticality level is A are those whose failure would cause a catastrophic impact such as to *"prevent continued safe flight and landing"*.

- Level B: Decision Coverage. Applications whose criticality level is B are those whose failure would cause a severe impact such as *"a large reduction in safety margins or functional capabilities, or physical distress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely, or adverse effects on occupants including serious or potentially fatal injuries to a small number of those occupants."* .
- Level C: Statement Coverage. Applications whose criticality level is C are those whose failure would cause major impact such as *"a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injuries."*.

Level D (minor impact) and E (no impact) do not require any measure of coverage.

2.1 Source Coverage versus Object Coverage

Applicants to DO-178B certification have proposed the use of object code coverage *instead* of source code coverage as a metric to satisfy the objectives of DO-178B. The proposed approach consisted in measuring either instruction coverage or branch coverage. Object instruction coverage (OIC) requires assessing all object instructions are executed at least once; branch coverage (OBC) in addition requires all conditional branches to be exercised for both directions (branch and fall through). The use of object code coverage has also been proposed as a way to cope with *untraceable object code*. Section 6.4.4.2 of DO-178B indeed states: *"The structural coverage analysis may be performed on the Source Code, unless the software level is A and the compiler generates object code that is not directly traceable to Source Code statements. Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences"*. Untraceable object code is object code generated by the compiler and impacting the control flow in a way not directly visible from source code. A typical example is array bounds check which may raise an exception. Reaching a given level of coverage for source elements may not be representative of the untraceable object code: even an extremely complete testing campaign may not assure all object code is executed. The untraceable object code is nevertheless present in the application, it may be executed during operation and may thus lead to unintended behaviour not verified during the testing process. Measuring object code coverage may thus be a way to ensure even untraceable code is executed during the requirement-driven testing campaign.

The issue raised by section 6.4.4.2 of DO-178B, and in general the equivalence of source and object coverage, is considered in both FAQ 42 of DO-248B (*"Can structural coverage be demonstrated by analyzing the object code instead of the source code?"*) and CAST paper 17 (*Structural Coverage of Object Code*, [3]) issued by the FAA (Federal Aviation Administration). Both documents assert that object code coverage can substitute source code coverage *"as long as analysis can be provided which demonstrates that the coverage analysis conducted at the object code will be equivalent to the same coverage analysis at the source code level"* [2]. In this context, equivalence means that object code coverage should require the same number of test cases as those needed to reach source code coverage for the appropriate metric (CAST paper 17 at [3]). In practice, object code coverage may be equivalent to source code coverage only on limited cases, for example when [4,12]: (i) the compiler generates a branch instructions for each condition and (ii)

only short circuit operators (and then, or else) are used and (iii) short circuit Boolean operators are always right-associated, like "`A and then (B or else C)`". When Boolean operators are right associated, the nodes of the resulting evaluation graph always have a single incoming edge (coming from the previous condition) and two outgoing edges: one exits the evaluation, the other proceeds to the next condition. The evaluation graph is thus a tree with n+1 leaves: there are n+1 paths from the root to the leaves, meaning that we need n+1 test vectors to fully cover the tree and achieve full OBC, exactly like for MC/DC (fig. 2a). When Boolean operators are left-associated (like in listing 1), the resulting evaluation graph is not a tree and can be potentially covered by less than n+1 tests (see fig. 2b). In the general case, it is rarely the case for object code coverage metrics to be in any direct relation to source code coverage metrics. For example, branch coverage is not equivalent to MC/DC, and instruction coverage is not equivalent to DC or SC. FAA reports such as DOT/FAA/AR-07/17 [4] provide evidence of this absence of relationship. It is also pretty simple to provide an example using the code in Listing 1. When compiling the code in listing 1 with GNAT Pro 6.2.1 for a bareboard PowerPC platform and with -O1 optimization, the generated object code is (branch instructions in bold):

```
_ada_p:
        stwu 1,-16(1)
        mflr 0
        stw 0,20(1)
        cmpwi 7,3,0 # compare r3 (A) with 0 and put result in cr7
        beq- 7,.L2  # if equal (A=False) branch to L2
        cmpwi 7,4,0 # compare r4 (B) with 0 and put result in cr7
        bne- 7,.L3  # if not equal (B=True) branch to L3
.L2:
        cmpwi 7,5,0 # compare r5 (C) with 0 and put result in cr7
        beq- 7,.L5  # if equal (C=False) branch to L5 (end)
.L3:
        bl do_something_pkg__do_something
```

The resulting execution graph is shown in figure 2b. Note that the same graph would be produced even with no optimization (-O0). Full branch coverage can be reached with just three tests (using test vectors (T,T,F), (T,F,T), (F,T,F)), while MC/DC requires at least four tests. The basic reason for such a difference is MC/DC begin a stateful property over the evaluation of a decision, whereas OBC is a local property of a single instruction.

Object code coverage is also just a partial solution to cope with the additional verification activities on untraceable object code for level A software (DO-178B, section 6.4.4.2). Producing additional tests to cover all untraceable object code has little sense because the requirements that caused such code to be produced cannot be found in the specifications of the system under development, but rather in the way the compiler implements a part of the programming language standard. Let's consider again array bounds check: they are generated because this is how the compiler implements the language standard, not because of some requirement specific to the application being developed. Moreover, covering all untraceable object code may require a significant effort which would be better spent in performing more meaningful verification activities. This is why CAST

paper 12 [3] suggests the use of a traceability study to satisfy the additional verification activities on untraceable object code for level A software: a traceability study provides evidence that, in a given context (coding standard, compiler, compilation switches), the compiler either generates traceable code or the untraceable code is correct – i.e. it correctly implements the requirements expressed by the specifications of the chosen programming language. A traceability study is thus a sort of black box qualification of the compiler: the requirements are the parts of the language standard used in the application context and tests compare the object code generated by the compiler with the legitimate expectations.

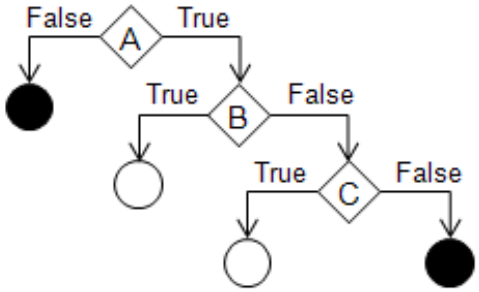

Figure 2a – Evaluation graph for "A and then (B or else C)":
Four tests are necessary to cover the whole graph.

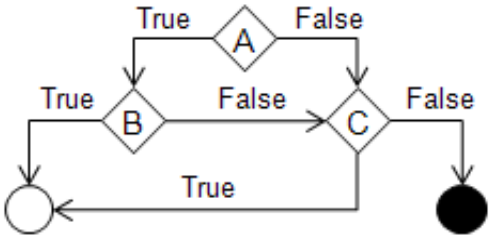

Figure 2b – Evaluation graph for "(A and then B) or else C":
Branch coverage achieved with (T,T,F), (T,F,T), (F,T,F)

Regardless of guidance and empirical evidence, the equivalence of object code coverage and source code coverage is however still a controversial topic and subject of debate: applicants managed to gain certification credit using both source code coverage and object code coverage.

3 Current approaches to structural coverage

Current industrial solutions to evaluate structural coverage are usually divided into two main categories: tools measuring source code coverage and tools measuring object code coverage. Given the domain of interest (aerospace), it is important to remind that the source code is usually cross compiled on host workstations and deployed on less powerful target platforms.

Tools measuring source code coverage like IBM Rational Test RealTime, LDRA Testbed, IPL AdaTest or Bullseye Coverage, usually depend on source code instrumentation to evaluate the coverage. Source code instrumentation requires the coverage tool to modify the application source code by adding calls to logging facilities in appropriate locations. For example, to measure statement coverage it is enough to log the entry in each block of execution (if statements, loops, switch statements, etc.): if all blocks are entered and no exception has been raised, then all statements have been covered. Of course, the more stringent the coverage metrics, the more invasive the instrumentation: instrumenting the source code for MC/DC requires to log the evaluation of each atomic Boolean expression, as well as to record all Boolean evaluation vectors to demonstrate independent effect of conditions on decisions. Coverage of instrumented code has the advantage of providing a straightforward mapping between the source code and the logged coverage information: it perfectly fits the requirements of DO-178B. On the other side, developers may be asked to demonstrate that the instrumentation did not modify the application behaviour and that the coverage of the instrumented application is representative of the coverage of the final application. It is usually necessary to compare the tests results for the two versions of the application (instrumented and not instrumented) to provide such evidence.

The other approach to structural coverage targets object code coverage. Tools measuring object code coverage usually log which instructions have been executed by exploiting hardware probes connected to the target via a JTAG/BDM, debug information and instruction-by-instruction execution. This approach has the major advantage of being non intrusive: since no instrumentation is added, and since coverage is measured on the final, cross compiled application, no additional verification is required. In addition, this approach depends on the target platform rather than on the programming language, making it an excellent candidate for applications written with several programming languages. The main limitation consists in the dependence upon the target hardware (and a physical connection) to execute the tests and measure coverage: this means that coverage may be measurable only when the target hardware is available; furthermore, such a process suffers from the slowness of the hardware connection to the target hardware required by the technology. Regardless of the limitations described in section 2.1, several successful tools measure object code coverage, for example VeroCel VeroCode or GreenHills GCover.

In this section, we identified the following major limitations in current state-of-the-art coverage approaches:

For approaches based on source coverage via instrumentation: it is necessary to demonstrate that instrumentation did not modify the behaviour and coverage of the final, non-instrumented application.

For approaches based on object code coverage: coverage results cannot be directly mapped onto source coverage metrics and they usually require the final hardware to be available and connected to the host via a probe.

4 Couverture

Couverture is a research project founded by French institutions within the System@tic framework. The project consortium comprises AdaCore, OpenWide, Telecom ParisTech and LIP6 (Pierre et Marie Curie University, Paris).

Couverture innovates on all aspects of current technology for structural coverage by:

- Providing *a virtualized execution platform* for cross-compiled application on the host machine. The virtual machine is able to produce a detailed execution trace.
- Measuring object code coverage through careful examination of *execution traces*.
- Measuring source code coverage as defined by DO-178B by relating elements of the execution trace (instructions, branches) to source-level structures (statements, decisions, conditions).

The Couverture technology is thus able to measure both source and object coverage for the cross compiled application. Supporting both object and source code coverage guarantees the maximum flexibility in terms of user needs. The use of a virtualized environment guarantees an extremely efficient process because it does not require the hardware to be available nor to be physically connected to the target board. Finally, since source code coverage is inferred from execution traces, no instrumentation of source code is required, making it unnecessary to provide evidence that the natively compiled, instrumented application is equivalent to the cross compiled one.

In the following sections we examine in details the main point of interests of Couverture.

4.1 A virtualized, instrumented execution environment

The core of the Couverture technology is a virtualized execution environment playing a dual role: it permits the execution of cross compiled applications on the host workstation without requiring the final hardware to be available and it gathers execution traces exploited by Couverture to measure object and source code coverage. The basic idea behind Couverture is to virtualize the approach commonly used to measure object code coverage: while traditional object coverage tools require a physical connection to the target hardware to measure coverage, Couverture uses a virtualized environment producing a rich execution trace containing the address of executed object instructions and the outcomes of conditional branches. Couverture is then able to determine actual object code coverage by processing several execution traces corresponding to the executions of different tests.

The technology at the heart of Couverture is QEMU (http://www.qemu.org). QEMU is a processor emulator employing dynamic binary translation. QEMU takes as input a cross-compiled application and translates basic blocks into executable code for the host processor. Thanks to binary translation, and to the fact that the target platform is usually much slower than the host workstation, performances are more than adequate: in the case of current generation PowerPC or ERC32/LEON2 targets, the simulator proved to be faster than the target boards. QEMU has an accurate model of the target Floating Point Unit (FPU). Although using an emulated FPU is slower than directly accessing the host FPU, the emulator assures a numerical precision identical to the target: from a numerical computations point of view, QEMU is representative of the target platform. QEMU also emulates different I/O chips which can be used to simulate the interaction

between the application and the environment. Thanks to the emulation of I/O, QEMU permits to run also those testing campaigns which would require a physical interaction with the external environment. QEMU is released as free software: the availability of the source code and the presence of an active user community guarantee the tool can be extended to support additional hardware and logging techniques. Our main extension to QEMU is the support for the generation of execution traces containing the list of executed object instructions: our addition required minimizing the amount of logged instructions to avoid the explosion of the size of execution traces. By comparing this list with the dump of the executable object, it is possible to measure object code coverage. In addition, the virtualized environment is able to keep track of the history of the evaluations of branch instructions: each single evaluation of a branch instructions is tracked along with the value to which the branch instruction was evaluated (branch or fallthrough). This additional information is necessary to measure branch coverage because passing through a branch instruction just once is not enough to achieve its full coverage.

Currently, the following platforms are supported via a QEMU-based emulation: PowerPC bareboard, LEON2 and ERC32 (SPARC) bareboard and WindRiver VxWorks 653 on PowerPC.

4.2 Introducing Source Coverage Obligations

The augmented execution traces produced by QEMU are enough to measure object code coverage. They are however not sufficient to infer source code coverage: without additional data, Couverture would suffer the same limitations plaguing other coverage tools targeting object code coverage (see section 2.2). Couverture needs to relate source code structures to object code elements and in particular to determine:

1. Which object code instructions are generated from each statement;
2. Which statement (and thus which instructions, see point 1) is executed if a decision is evaluated to True and which is executed if it is evaluated to False.
3. Which branch instruction corresponds to which condition;
4. How conditions (and thus branch instructions, see point 3) are grouped together to form a decision:

Standard debug data does not provide all required information. Debug formats such as DWARF 2/3 can just link object instructions to a location in the source code identified by a line and a column, but are not able to map them to source-level structures such as conditions or decisions. This is why we had to complement the debug information with Source Coverage Obligations (SCOs). SCOs are extracted from source code by the compiler: they identify source constructs for which coverage artifacts need to be exhibited in order to satisfy some coverage objective. SCOs contain a compact representation of the control flow which group conditions within the enclosing decision and specify which statements are executed when a decision is evaluated to True and False. By weaving SCOs with debug information, it is possible to represent the source-level control flow and its structures in terms of object instructions. Couverture can then infer all DO-178B source code coverage metrics from the execution traces produced by QEMU:

- **Statement coverage** is achieved if any instruction generated by a given statement are executed (see point 1 above).
- **Decision coverage** is achieved if the Boolean expression composed by all object branch instructions associated to a given condition is evaluated to both True and False. Calculating decision coverage requires a sort of abstract interpretation of execution traces to determine the evaluation of a decision starting from the evaluation of the associated object branches.
- **MC/DC** is achieved if all branch instructions have demonstrated to have an independent impact on the evaluation of the decision they belong to (see point 4 above). Of course, this includes each branch instructions to be evaluated for both directions (branch and fallthrough, see point 3 above)

As explained, SCOs and debug information contain the information to detect whether the conditions above apply. An example shall help us clarifying this crucial point. Starting from the decision at line 3 of the procedure P in listing 1, the GNAT Pro compiler is able to produce the data depicted in fig. 3: it associates conditions to branches and it groups branches corresponding to the evaluation of a given decision. Thanks to this information, Couverture is able to detect that the three tests sufficient to achieve branch coverage (see fig. 2) are not enough for MC/DC: since the SCO states that all three branch instructions are logically owned by the same condition, Couverture can easily verify that the independent impact of decision "C" on the whole decision is not demonstrated (test vector (T, F, F) is missing). This example shows that the coverage technology employed by Couverture does not suffer from the same limitations of other object code coverage tools.

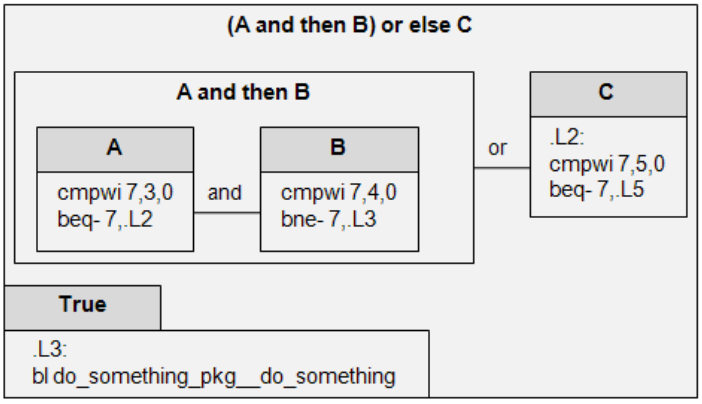

Fig. 3 – Simplified graphical representation of a SCO plus debug information produced by GNAT Pro

4.3 Preserving execution flow

The mechanism employed by Couverture to measure code coverage consists in inferring source code execution flows by analyzing object code execution traces with the support of SCOs: for this to be possible, the control flow across object instructions must be representative of the control flow across source code elements. In particular, each condition must be mapped onto a branch

instruction. It is thus extremely important to preserve the execution flow from source code to object code: if control structures are not preserved during compilation, it may be impossible to infer paths across source code elements from object code execution traces. A simple example will help clarifying this idea. Consider the following function:

```
1. function F (A : Boolean) return Boolean is
2. begin
3.    if A then
4.       return True;
5.    else
6.       return False;
7.    end if;
8. end F;
```

Listing 2 – Example Source Code

When compiled with minimal optimization (-O1), the resulting object code is:

```
_ada_f:
        blr    # return the value of the parameter
```

The generated object code does not contain any branch instruction: the execution flow implemented by the object code is not representative of the execution flow in the source code. As a result, even the Couverture toolset would assert that any form of DO-178B coverage can be achieved with just one test, instead of the two actually necessary: in fact, a simple execution of function F will cause the complete coverage of all generated object code. In the general case, even disabling optimization cannot guarantee the source code execution flow is preserved down to object code. To address this issue, the compilation technology shall guarantee the execution flow is preserved, even when using optimizations. In GNAT Pro (AdaCore flagship Ada compiler), this is achieved by the –fpreserve-control-flow switch. –fpreserve-control-flow disables all requested optimizations (possibly via the –O option) which may influence the preservation of the control flow from source code to object code. The result of using this compilation switch with the source code in listing 2 is the following:

```
_ada_f:
        cmpwi 7,3,0
        beqlr- 7    # branch instruction for A
        li 3,1
        blr
```

In the example, a branch instruction is generated for the condition "A" in the source code: this ensures the control flow described at source level is preserved in object code instructions.

5 Current Results

The Couverture technology is currently used both as an internal tool and within an industrial context. In both cases, Couverture has been coupled with a GNAT Pro compilation chain.

5.1 Industrial Test Cases

Couverture is currently used at AdaCore to measure the source code coverage of the internal test suites used to qualify the Ravenscar run-time for the ERC32 and LEON2 processor in a ECSS-40 context. QEMU has been extended to support these processors.

In addition, the development team responsible for the Air Data Inertial Reference Unit (ADIRU) for the Airbus A350 XWB at Thales Aerospace uses Couverture to measure object and source code coverage at DO-178B level A [10]: in this context, QEMU emulates a bare-board PowerPC target.

5.2 R&D Test Cases

Couverture has also been evaluated using applications developed within two R&D projects. The first test case is represented by an Ada 2005 application built in the context of the IST-ASSERT project. The application is a reduced subset of the guidance and navigation system of a satellite. Coverage analysis with Couverture uses the support for LEON2 in QEMU. We exercised the application in nominal mode, and processing the output with the Couverture toolchain. From the analysis, we found out that 71% percent of the code was fully covered, whereas 25% was partially covered. Only 4% was not covered. The coverage analysis revealed that the partially covered code corresponded to error conditions not being exercised, and thus showed that our test suites didn't completely cover our requirements.

The second test case resolves around POK, a real-time embedded kernel for safety-critical systems. POK implements several industrial standards, including ARINC653 used in the avionic domain. POK comprises two main layers: the kernel and the partition runtime (called libpok). The kernel provides partition support, inter-partition communication and scheduling; it is particularly compact in order to (i) ease its verification/certification and (ii) minimize the amount of potentially unintended functions. The libpok layer provides kernel-interfacing functions and abstraction layers, for example, POSIX or ARINC653. Partitions are executed on top of the POK kernel to ensure partitions isolation. Each partition can include one or more compatibility layers to execute their application code. Each partition is also isolated in space (a memory segment is dedicated to each partition) and time (each partition has at least one time slot to execute its threads). POK is written in C for x86 and PowerPC architectures. Ongoing work includes a LEON port and an Ada API for the ARINC653 layer of libpok. POK is available as open-source software under the BSD licence at http://pok.gunnm.org.

Since the libpok layer contains several heterogeneous components, its full inclusion would likely lead to deploy code not derived from requirements in the final application. To avoid the deployment of unintended functions, kernel and libpok can be configured to explicitly select the required services using an automatic code generator from AADL specifications [13]. The code generator configures each layer according to system requirements and thus ensures configuration correctness and absence of unintended functions.

We have carried out experiments to analyze kernel code coverage using Couverture (statements coverage). We studied two examples of POK using different kernel services:

- "Partition-threads": one partition that contains two tasks. It only uses the partitioning services.
- "Middleware-queuing": two partitions with one task per partition. Inter-partition communication occurs between the two partitions so the kernel provides partitioning and inter-partition communication services.

The measured code coverage for x86 targets ranges from 65% (middleware queuing) to 91% (partition threads). This is mostly expected: more complex setup in the second example leads to potentially more dead code. Thanks to Couverture we were thus able to identify which services of the POK kernel should be moved to libpok in order to include them only when strictly necessary: after these changes, POK and its test cases shall comply with the coverage requirements for DO-178B level C. In addition, we managed to improve our code generation strategies to decrease the amount of dead code coming from non-necessary libpok services. Both case studies demonstrate that Couverture is ready to address the challenges of certification of systems deployed on the LEON2 processors and of applications executed on partitioned kernels.

6 Conclusions

In this paper, we have illustrated an innovative approach to structural coverage based on a virtualized, instrumented, execution environment. Couverture is able to measure structural coverage of object and source code without requiring any form of application instrumentation with a single execution of the cross-compiled application and test suites. To measure source code coverage, the Couverture technology requires the compiler to identify source constructs for which coverage artifacts need to be exhibited and to link them to object instructions. This is achieved by the generation of Source Coverage Obligations and debug information. The compiler must also assure the execution flow is preserved from source code to object code. The GNAT Pro compiler has been extended to implement such feature via the –fpreserve-control-flow switch. The technology described in this paper is currently used both as an internal tool and in a major project in a DO-178B level A context.

In addition to its technical contributions, Couverture is the first industrial project which leverages on the Open-DO vision [9]. Open-DO is an initiative to promote an open and collaborative approach to the development of high integrity systems, in particular within a DO-178 context. Couverture completely embraces the Open-DO vision, providing an open repository [11] where development artifacts are freely available: potentially, the whole user community can access and

contribute to the development and qualification of the technology. This is the first step towards the cross fertilization between open and high-assurance development promoted by Open-DO.

References

[1] EUROCAE: "Software Considerations in Airborne Systems and Equipment Certification" - DO-178B, 1992, 1999.

[2] EUROCAE: "Final Report of Clarification of DO-178B" – DO-248B, 2001.

[3] FAA CAST papers, http://www.faa.gov/aircraft/ air_cert/design_approvals/air_software/cast/cast_papers/

[4] FAA: "Object-Oriented Technology Verification Phase 3 Report - Structural Coverage at the Source-Code and Object-Code - DOT/FAA/AR-07/20", 2007

[5] FAA: "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion - DOT/FAA/AR-01/18", 2001

[6] NASA: "A Practical Tutorial on Modified Condition/ Decision Coverage - NASA/TM-2001-210876", 2001

[7] Ntafos, S.C: "A comparison of some structural testing strategies", IEEE Transactions on Software Engineering, Issue 6, 1988

[8] Beizer, B., "Software Testing Techniques", 2nd edition

[9] Open-DO: www.open-do.org

[10] Thales Aerospace Division Selects GNAT Pro for Airbus A350 XWB (Xtra Wide-Body): http://www.adacore.com/2009/06/01/a350/

[11] Couverture Open Repository at Open-DO.org: http://forge.open-do.org/projects/couverture

[12] Romanski G.: "MCDC coverage using short0circuit conditions", available at verocel.com.

[13] Delange J., Pautet L., and Kordon F.: "Code Generation Strategies for Partitioned Systems", 29th IEEE Real-Time Systems Symposium (RTSS'08), work in progress proceedings, December 2008