Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework

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

Abstract: This paper presents COUVERTURE, an open coverage analysis framework for safety-critical software development. COUVERTURE offers non-intrusive source and object coverage analysis on unmodified user code, using instrumentation of a virtual execution platform based on QEMU, a flexible and efficient open-source CPU emulator.

COUVERTURE focuses primarily on the DO-178 civilian avionics certification process: it supports the source coverage analysis activities required for each criticality level. It also provides instruction and branch coverage analysis of object code. We discuss the relationship between execution traces, object coverage metrics, and source coverage. In particular, we provide a characterization of source constructs for which object branch coverage implies modified condition/decision coverage, and we discuss the trace collection process required for those cases where there is no such equivalence.

Sources of the software components and qualification material for COUVERTURE are available under open source licenses from the OpenDO site (`http://www.open-do.org/`).

Keywords: Source coverage, object coverage, DO-178B, DO-178C, target emulation, open source.

1 Introduction

The development of a high-integrity application involves interaction between the design and testing phases. System requirements are decomposed into high-level requirements, which lead to the definition of a software architecture and lower level requirements. The latter are implemented, generally using a high-level programming language such as Ada, C, or C++. A set of corresponding verification activities often mirrors each decomposition step. For example, the testing part of the verification associated with system requirements is often referred to as *acceptance testing*, the one associated with software architecture and high-level requirements is referred to as *integration testing*, while *unit testing* is often used to verify low level requirements. The workflow implied by such a decomposition is the standard waterfall model but even when more agile workflows are used, with shorter iterations and earlier emphasis on integration issues, the same correspondance exist between a level of requirements and verification activities.

At any level of decomposition, the test cases to verify the implementation of a given requirement are to be built from the requirement itself: the idea is to have the *specifications*, rather than the *implementation*, drive the testing strategy. One essential point in this process is the evaluation of the quality of the testing: *Were enough test cases developed to verify that the software properly implements all of its requirements? Is there any functionality in software that does not stem from requirements?*. *Structural coverage analysis* offers a means to conduct such an evaluation.

Various coverage metrics are in industrial use [13]. They differ by how comprehensive a test suite should be to achieve coverage. Two common examples are *statement* and *decision coverage*; the former measures how source code statements are exercised, the latter measures how complex or flow-controlling Boolean expressions (decisions) are evaluated.

Achieving a certain coverage criterion means that the requirement-driven testing strategy includes a set of tests whose execution exercises all the coverable structures that the metric targets.

For example, achieving statement coverage means executing at least once all statements present in the source code of interest, and achieving decision coverage means that the test suite executes all decisions with both outcomes (True and False). In section 2.1, we discuss MC/DC (Modified Condition/Decision Coverage), an even more stringent criterion.

We will use the following small Ada program as an example throughout this paper:

```
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 on listing 1 contains two statements (lines 3 to 4) and a single decision (the Boolean expression at line 3). Statement coverage for procedure P can be achieved with just one test causing the decision to be evaluated True. Decision coverage is achieved with two tests, one evaluating the decision to True, the other to False.

Coverage evaluation is explicitly required by several industrial standards: for example, DO-178B (civil avionics) [14], EN 50128 (railroad systems) [7], ECSS-E-ST-40C (space systems) [11], or IEC 60880 (nuclear) [12].

The remainder of this paper focuses specifically on DO-178 (we use DO-178 to refer to both DO-178B, the current version of the standard, and DO-178C, its upcoming replacement). In section 2, we give an overview of the coverage analysis activities defined in DO-178, and introduce MC/DC analysis. We discuss the relationship between object and source coverage, as well as current assessment methodologies. Section 3 presents our innovative approach: achieving object and structural coverage analysis by means of control flow traces produced by an emulated execution platform.

In section 4, we provide an in-depth discussion of the applicability of our approach to MC/DC assessment. We introduce several theoretical results on the conditions in which MC/DC can be derived from synthetic branch coverage information, and those in which full execution history must be collected.

We present some practical results obtained with COUVERTURE in section 5, and we discuss future directions in section 6.

2 Structural coverage and DO-178

DO-178 [14] is the international standard providing guidance for the development of software to be deployed on airborne systems flying over civil ground. DO-178 describes a set of development and verification objectives which shall be met by a development team in order to gain certification credit for the developed software. Structural coverage analysis is an activity associated with the software verification process objectives (see [14], table 7).

Three different metrics are considered within DO-178: Statement Coverage (SC), Decision Coverage (DC) and Modified Condition/Decision Coverage (MC/DC). All these metrics express coverage in terms of source code elements: SC measures which source code statements are exercised, DC measures how boolean expressions (decisions) are evaluated, and MC/DC refines further with rules on atomic elements in boolean expressions (conditions). We give a brief description of MC/DC in section 2.1; additional information can be found in [9].

The metric required for structural coverage depends on the criticality of the application: the more safety-critical the application, the more stringent the metric (i.e. more extensive testing is required to achieve the coverage objective):

- Level A requires MC/DC. Applications whose criticality level is A are those whose failure would cause a catastrophic impact.
- Level B requires Decision Coverage. Applications whose criticality level is B are those whose failure would cause a severe impact such as a large reduction in safety.
- Level C requires Statement Coverage. Applications whose criticality level is C are those whose failure would cause major impact such as a significant reduction in safety margins.

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

2.1 Modified Condition/Decision Coverage

2.1.1 MC/DC definition

MC/DC distinguishes decisions and conditions. A condition is an atomic Boolean expression. A decision is composed of one or more conditions connected by boolean operators. `(C1 and then C2) or else C3`, for example, is one decision with three conditions `C1`, `C2`, and `C3`. 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, and each condition in a decision must be shown to independently affect the outcome of that decision [14]. To explain what "independently affect" actually means, we introduce here the notion of *decision evaluation vector*.

An evaluation vector for a decision is a vector of Boolean values where each element corresponds to the value of one condition in the decision. For example, (T,F,T) is an evaluation vector for `(C1 and then C2) or else C3` where `C1` evaluates to True, `C2` to False and `C3` to True. A condition C has an independent influence on a decision D if two evaluation vectors exist which evaluate D to True and False and are identical but for the value of C. Note that a given vector may participate in more than one pair, each pair showing independent influence of a different condition.

For example, to achieve MC/DC for the 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 influence 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.

2.1.2 Unique Cause + Short-Circuit MC/DC

MC/DC has been the source of confusion in the avionics certification community, stemming primarily from the meaning of a condition's "independently affecting the outcome" of a decision. The definition of MC/DC at the beginning of the previous section, based on the Glossary of DO-178B, has come to be known as *Unique Cause* MC/DC. This definition has several limitations:

- Unique Cause MC/DC can never be achieved for decisions with coupled conditions (because coupling means that the condition values change together);
- The set of possible independence pairs for a given condition is unnecessarily restrictive (because even inputs that cannot possibly affect the output are forced to be unchanged).

To address these issues, several variants of MC/DC have been proposed, which relax the definition of an independence pair, allowing some inputs (other than the condition whose independent influence is being demonstrated) to differ in the two test vectors. *Unique Cause+Masking MC/DC* and *Masking MC/DC* [9] are two examples of such relaxed variants which Certification Authorities have accepted as reasonable alternatives [4].

In COUVERTURE, we use an intermediate form, which introduces a limited use of condition hiding: *Unique Cause + Short-Circuit MC/DC*. In this variant, we consider the right-hand side operand of a short-circuit operator (`and then` or `or else`) to be hidden when the value of the left-hand side alone determines the outcome of the operator, and we ignore differences in hidden condition values for determination of independence pairs.

This is consistent with the code generation strategy for such operators: the right-hand side is evaluated only if the value of the left-hand side makes it necessary. In other words, a condition is hidden under Unique Cause + Short-Circuit MC/DC when the evaluation of the enclosing decision does not even evaluate that condition. In this case, no value change of that condition can possibly be the cause of a change in decision outcome. As for standard Unique Cause MC/DC, we retain the constraint that all *evaluated* conditions (other than the one being tested for independent influence) have fixed values. This definition of hiding is more conservative than the notion of *masking* proposed in [4].

2.2 Source Coverage versus Object Coverage

Applicants for DO-178 certification have sometimes proposed the use of object code coverage instead of source code coverage as a metric to satisfy the objectives of DO-178. The proposed approach involves measuring either instruction coverage or branch coverage. Object instruction coverage (OIC) requires assessing whether all object instructions are executed at least once; object branch coverage (OBC) in addition requires that all conditional branches 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-178 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 compiler generated machine code that impacts the execution control flow in a way not directly visible from source code. An example is the Ada `mod` operator, which requires an implicit conditional branch to distinguish between positive and negative moduli. Achieving a certain source coverage level is not indicative of coverage of untraceable code: even a comprehensive testing campaign (from a source coverage point of view) may not assure all object code is executed. The untraceable object code is nevertheless present in the application and may lead to unintended behaviour not verified during the testing process.

Measuring object code coverage may be a way to ensure even untraceable code is executed during the requirement-driven testing campaign. However, CAST paper 12 [5] 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.

The issue raised by section 6.4.4.2 of DO-178, and in general the equivalence of source and object coverage, is considered in both FAQ 42 of DO-248B [15] (*Can structural coverage be demonstrated by analyzing the object code instead of the source code?*) and CAST paper 17 (*Structural Coverage of Object Code*) issued by the Federal Aviation Administration [6].

Both documents assert that object code coverage can substitute for 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*. In section 4.3, we provide a precise analysis of the conditions under which the OBC and MC/DC properties imply each other, for a given set of test cases. It should be noted that such an equivalence applies only in the context of a given code generator and coding guidelines: different code generation algorithms may produce object code whose OBC status is different for the same set of inputs achieving a given source coverage objective.

2.3 Current approaches

Current industrial solutions to evaluate structural coverage fall into two main categories: tools measuring source code coverage and tools measuring object code coverage.

A common approach to source coverage, implemented in tools such as IBM Rational Test RealTime, LDRA Testbed, IPL AdaTest or Bullseye Coverage, depends on source code instrumentation. The coverage analysis tool modifies application code by inserting calls to logging facilities in appropriate locations. For example, to measure statement coverage, it is enough to log the entry in each basic block (IF statements, loops, CASE statements, etc.). If all basic blocks are entered and no exception has been raised, then all statements have been covered. More stringent coverage metrics require increasingly invasive instrumentation: instrumenting the source code for MC/DC requires logging the evaluation of each atomic boolean expression, in order 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 satisfies the requirements of DO-178. On the other hand, 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.

Another common approach to structural coverage targets object code, using hardware probes that log which instructions have been executed and allow step-by-step execution. This approach, implemented in tools such as VeroCel VeroCode or GreenHills GCover, has the advantage of being non intrusive: coverage is measured on the final, cross compiled application, and no additional verification is required. In addition, this approach is independent of the choice of a specific programming language, making it an excellent candidate for multi-language applications. However, this technique has several limitations. First of all, as we discuss in section 4.3, MC/DC and OBC are *not* equivalent in the general case. Furthermore, coverage may be measured only when the target hardware is available.

These limitations make it impractical to evaluate object code coverage continuously during the development process, potentially leading to late discovery of defects in the testing strategy, and costly remediation actions.

3 The COUVERTURE approach to coverage

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 control flow execution trace (i.e. a report of which basic blocks have been executed, and how control transferred from one to another).
- Measuring object code coverage through careful examination of execution traces.
- Measuring source code coverage as defined by DO-178 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, thus offering maximal flexibility. The use of a virtualized environment removes any dependency on target hardware and offers an efficient coverage analysis workflow. No source code instrumentation is required, which ensures that the coverage analysis activity does not interfere with application behaviour.

3.1 Virtualized 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 used 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 [1]. 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. This method is in general very efficient, and in the case of current generation PowerPC or ERC32/LEON2 targets, the simulator even proved to be faster than actual target boards in our environments. QEMU also provides an accurate model of the target Floating Point Unit (FPU), as well as an emulation of various I/O devices, allowing a significant amount of testing without any physical environment interaction.

An open source project, QEMU can be modified and extended by users. Our main contribution to QEMU is the support for the generation of execution traces. Two output modes are supported:

- synthetic, bounded-size traces indicating which range of code addresses have been executed, and, for conditional branches, whether they have been taken in one direction or the other, or both.
- full historical traces (for selected code addresses), indicating branch direction information for each separate evaluation of relevant conditional instructions.

These traces are the basis for COUVERTURE's coverage analyses, to date supported on LEON2/ERC32 (SPARC) bareboard platforms as well as on PowerPC, bareboard or with Wind River VxWorks 653.

3.2 Source coverage obligations

The traces produced by QEMU are enough to measure object code coverage. They are however not sufficient in themselves to produce source code coverage evidence as required by DO-178.

In order to produce these reports from object execution traces, COUVERTURE needs additional information about source code structure. This information comes in two parts.

The first one is the standard DWARF debugging information emitted by the compiler. Debugging information links every executable instruction to a location in source code (identified by file, line and column).

In order to further relate this information to coverable constructs (statements, decisions, and conditions), COUVERTURE augments the debugging information with *Source Coverage Obligations* (SCOs), which identify source constructs (including their source location) for which coverage artifacts need to be exhibited in order to satisfy some coverage objective.

SCOs are extracted from source code by the compiler. They contain an abbreviated representation of the logical control flow of the program, including the full logical structure of decisions. These obligations are then discharged by associating them with execution traces, using debug information and source locations as the connection.

COUVERTURE can thus infer all DO-178 source code coverage metrics from the execution traces produced by QEMU: Statement, Decision, and Modified Condition/Decision coverage.

4 MC/DC assessment from execution traces

In this section, we discuss how execution traces collected from an instrumented virtual execution environment, as described above, can be used to determine whether a given source coverage criterion (and specifically MC/DC) is achieved by a certain set of executions.

4.1 MC/DC assessment work flow

The general principle in this process is to infer the values of each condition for a given evaluation of a decision from the direction taken by execution at each conditional branch instruction in the object code. This assumes that the code generation process generates appropriate conditional branch instructions (as opposed e.g. to Boolean computations) for each condition, i.e. that the *object control flow graph* reflects the source structure of decisions. In GNAT, this is achieved using the `-fpreserve-control-flow` compiler switch, in addition to mandating the exclusive use of *short-circuit* boolean operators (the non-short-circuit Boolean operators `and`, `or`, and `xor`, as well as all relational operators on Boolean operands, are forbidden).

SCOs and DWARF debugging information are used as an aid to trace each edge of the control flow graph back to edges of the decision's Binary Decision Diagram (BDD), i.e. to the value of each condition: from a conditional branch instruction, debugging information yields a location in source code (file, line, and column), and SCOs then indicate what condition is tested at that location. This mapping is created during a preliminary static analysis phase, prior to code execution.

Determining whether MC/DC is reached for a given decision and a given set of tests requires the ability to reconstruct the complete set of evaluated test vectors, i.e. the combination of values of the relevant conditions encountered for each evaluation of a decision. In general, this requires full historical traces of the corresponding conditional branch instructions, and the amount of collected data can potentially grow to an unwieldy size.

However in some cases it is sufficient to keep synthetic information about each conditional branch instruction (indicating whether it has been taken in one direction, in the other, or both), as opposed to keeping state information about all evaluations of the instruction. In other words, there are cases where MC/DC can be derived from object branch coverage information and in such cases, there is no need to keep and process historical traces since the regular stateless execution traces are sufficient. Relying as much as possible on stateless traces makes it also easier to produce accurate coverage metrics in a multithreaded context. It is therefore important to be able to characterize very precisely and accurately the situations where MC/DC can be deduced from OBC.

4.2 Non equivalence between OBC and MC/DC

First let us have a look at how MC/DC and object coverage relate to each other on some simple cases. Cases of non-equivalence for decisions with up to 5 conditions have been studied in [10]: non-equivalence cases have been shown to occur in decisions with three or more conditions, and an illustration is provided with (A `and then` B) `or else` C, where A, B and C are three independent conditions. A representation of this decision's BDD is depicted on figure 1(a):

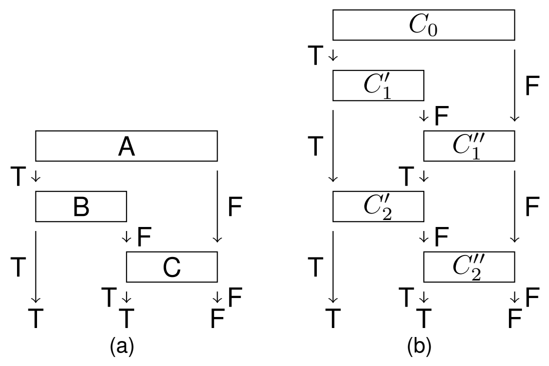

Figure 1: Example decision BDDs

From this representation, we can see that a set of three evaluations can achieve branch coverage of the whole BDD, corresponding to the three vertical paths in figure 1(a). These evaluations are:

A	B	C	(A `and then` B) `or else` C
T	T	x	T
T	F	T	T
F	x	F	F

where “x” means not evaluated and thus can be indifferently True or False. Now, indeed, even though all the BDD edges are covered, MC/DC is not met. In particular, the independent effect of conditions B and C on the decision is not shown. Since $n + 1$ tests are needed to cover a decision with n condition with respect to MC/DC, 3 evaluations cannot cover a three-condition decision.

It turns out that this particular case can be generalized in a quite spectacular counterexample: there exists classes of decisions with an arbitrary high number of conditions that can be branch covered by just three evaluations; the previous example was one element of this class with 3 conditions.

Consider the following set $\{D_n\}_{n\in\mathbb{N}}$ of decisions:

- let D_0 be a simple condition decision; by convention, we will call C_0 its condition;
- let us define D_n, for any $n > 0$, as follows:
  $D_n = (D_{n-1}$ `and then` $C'_n)$ `or else` C''_n
  C'_n and C''_n being independent from each other and from any condition in D_{n-1}.

In other words:

- $D_0 = C_0$
- $D_1 = (C_0$ `and then` $C'_1)$ `or else` C''_1
- $D_2 = (((C_0$ `and then` $C'_1)$ `or else` $C''_1)$
  `and then` $C'_2)$ `or else` C''_2
- $D_3 = (((((C_0$ `and then` $C'_1)$ `or else` $C''_1)$
  `and then` $C'_2)$ `or else` $C''_2)$
  `and then` $C'_3)$ `or else` C''_3
- ...

Figure 1(b) shows the BDD for D_2, where it is visible that all the edges can be covered by three evaluation paths which only demonstrate the independent effect of C_0:

C_0	C'_1	C''_1	C'_2	C''_2	D_2
T	T	x	T	x	T
T	F	T	F	T	T
F	x	F	x	F	F

We can thus build a decision D_n with an arbitrary number of conditions, that can be BDD branch covered by just three evaluation paths. As MC/DC can only be achieved with a minimal number of $n + 1$ evaluations, this is a striking case where BDD branch coverage (and consequently OBC) is far from being equivalent to MC/DC.

4.3 Equivalence cases between OBC and MC/DC

Having introduced two example situations where OBC and MC/DC are not equivalent, we now generalize this discussion using formal reasoning on properties of the canonical reduced ordered binary decision diagram associated with each decision, considering the variables ordered as they appear in source code. The construction of this canonical diagram is given in [3]. In the remainder of this paper, we will refer to this canonical diagram informally as the BDD.

We rely on the straightforward correspondence between nodes of the object control flow graph and nodes of the BDD: it is assumed — and this property is guaranteed by GNAT when using the `-fpreserve-control-flow` compiler switch — that the evaluation of a condition corresponds to a conditional branch instruction (that is, an alteration of the execution control graph depending on the condition), and that a mapping can be established between edges of the control flow graph and edges of the BDD.

Under this representation, evaluating the decision using the Ada semantics of short-circuit operators (evaluating the right-hand side only if necessary to determine the outcome) is exactly equivalent to following a path through the BDD.

The following property then holds:

Theorem 1 *Given a decision D, branch coverage of the BDD implies MC/DC if, and only if, there is no diamond in the BDD (i.e. no node of the BDD is reachable through more that one path from the root).*

A complete formal proof is given in the COUVERTURE documentation [2].

As a consequence of this theorem, when the BDD has no diamond, it is possible to establish MC/DC without resorting to full historical traces. When there is exactly one conditional branch instruction for each condition, this is also exactly equivalent to OBC, assuming a code generation process where the object control flow is isomorphic to the BDD (which is the case in the context of COUVERTURE).

When some conditions involve multiple conditional branch instructions, OBC still implies MC/DC, but becomes in effect an even stronger property: MC/DC could potentially be established by a test set that does not achieve OBC. However, as long as the conditions of theorem 1 hold, it is still the case that MC/DC can be determined from stateless execution traces (i.e. from the OBC information for each conditional branch instruction) even if full OBC is not achieved (because some edge of the control flow graph that remains within a condition might not be taken, even when all those edges that do correspond to BDD edges are).

4.4 Alternative characterization

In this section we provide an alternative (equivalent) property that characterizes cases where BDD branch coverage implies MC/DC:

Theorem 2 *Given a decision D, BDD branch coverage implies MC/DC if, and only if, when considering the negation normal form D' of D, for every sub-decision E of D', all binary operators in the left-hand-side operand of E, if any, are of the same kind as E's operator.*

The negation normal form is obtained by rewriting the expression using De Morgan's laws so that negations apply only to atomic conditions (and not to more complex subexpressions).

More informally, this corresponds to expressions where there is no `or else` in the left-hand-side of a `and then` and conversely.

The proof of this theorem first shows that this alternative characterization is equivalent to having no diamond in the associated BDD. Theorem 2 then follows by application of Theorem 1.

5 Results

In this section, we give examples of the output of the COUVERTURE analysis tool for simple test cases, and we discuss interesting MC/DC analysis results obtained on real-life industrial code.

5.1 Object and Source coverage ouput examples

We will consider the results obtained exercizing the Ada procedure in listing 1, which evaluates `(A and then B) or else C` where A, B and C are incoming arguments. We are using a version of the GNAT compiler which ensures that each condition eventually materializes as a conditional branch instruction.

Let us look at object branch coverage results first. Amongst several possible output formats, we will observe an “annotated source” report, where each source line is annotated with a synthetic sign to summarize the coverage information for all the instructions associated with that line.

Out of two calls with (T, T, F) and (T, F, T) for A, B and C respectively, we obtain partial branch coverage for the code associated with the evaluation line. This translates into a '`!`' sign next to the line number, and we can request the expansion of all the associated machine instructions to confirm:

```
4 !:    if (A and then B) or else C then
...
fffc019c v:  beq cr7, <_ada_p+00000044>
...
fffc01ac +:  bne cr7, <_ada_p+00000054>
...
fffc01bc v:  beq cr7, <_ada_p+00000060>
...
```

The '`v`' sign next to address for the first conditional branch instruction indicates that the branch was only taken one way. This is as expected since the branch is decided on the value of A, and A was True in both evaluations. The '+' on the second branch indicates that it was taken both ways, as expected since B was evaluated twice, once True and once False. The '`v`' sign on the third branch indicates partial coverage of the branch for C, as for A. Even though the C argument was passed both True and False in the program, partial coverage of this branch was actually expected because C is not evaluated on the first call.

As suggested earlier in this paper, full object branch coverage for this expression is achieved by adding no more than a third call with (F, F, F), which indeed translates as '+' annotations everywhere.

If we now consider source coverage criteria, the 3-calls testset we have so far achieves decision coverage as well: there is one decision in the code and the tests have exercized it both True and False. We still don't have MC/DC, however, and the reason is reported if we re-analyze for this criterion:

```
"B" : failed to show independent influence
"C" : failed to show independent influence, MC/DC not achieved
```

5.2 Evaluation on actual industrial code

Our approach to DC and MC/DC assessment through execution traces rely on good properties of the code generator used, preserving source decision structure and reflecting it, to some extent, in the object code control flow graph. Improvements have been made to the GNAT compiler enabling this mapping to be performed completely on an actual industrial application, with no optimization of the generated code. Adjusting the code generator to preserve sufficient decision structure in the generated code at higher optimization levels is still work in progress.

We have evaluated COUVERTURE against two industrial applications. Reporting from the analysis phase of COUVERTURE provided interesting insight on the frequency of those constructs that are specifically difficult to check for MC/DC coverage (decisions involving diamond paths in their BDD). Some of these results are summarized below.

Measure	App. 1	App. 2
Ada units	351	8 356
Ada SLOCs	67 344	1 166 183
Decisions	869	37 324
Max. conds per decision	7	78
Decisions with diamond	7 (0.8 %)	141 (0.4 %)

For the first application, the breakdown of decisions according to condition count is as follows:

# conditions	1	2	3	4	5	6	7
decisions	597	194	49	13	5	10	1

Of these 869 decisions, only 7 (or 0.8 %) have a node in their BDD that is reachable through multiple paths. Additionally, all of these complex decisions are in preconditions, postconditions or assertions: none is in actual functional code.

The second application similarly has relatively few decisions with multiple paths (0.4 % of all decisions). It has a few pathological very large decisions (up to 78 conditions), but 35 844 decisions (or 96 %) have one, two or three conditions.

This means that in practice, even though COUVERTURE is prepared to handle the worst case scenario where full historical traces is required to establish MC/DC coverage, in practice the overhead will be marginal compared to plain OBC assessment (which can be performed using synthetic traces of bounded size), because in industrial code those worst cases are very infrequent.

6 Future directions

The current results of COUVERTURE have been obtained with all compiler optimizations disabled (`gcc -O0`). Higher optimization levels hinder coverage assessment because of code reorganizations that break the relationship between object code and source coverable constructs. Work is in progress to improve COUVERTURE so that most of `-O1` can be retained while preserving the ability to compute source coverage from execution traces.

As discussed in section 3.2, this computation relies on SCOs: artefacts produced by the compiler that describe source constructs that are the targets of coverage assessment. SCO generation is currently implemented only for Ada language programs. We have plans to extend GCC to also generate SCOs for C source code. Ongoing research work of academic partners in the COUVERTURE project also aims at applying it to Ocaml programs [8].

7 Conclusion

We have presented the COUVERTURE approach to object and structural coverage analysis for certified safety-critical applications, in particular in the context of DO-178. Our methodology is non-intrusive, assessing coverage on unmodified user code, and collecting execution traces from an instrumented emulated and very efficient execution platform. We have used object coverage techniques in order to generate precise source coverage metrics. In particular, we have formally characterized the cases where synthetic object branch coverage information is sufficient to establish MC/DC coverage and the cases where it is not.

This approach has been implemented in a set of open source tools that have been successfully evaluated against both extensive unit tests aiming at tool qualification and real-life industrial application code. These tools and methods will constitute an important building block within the OpenDO initiative.

We also hope that the availability of such efficient and accurate coverage techniques on the embedded code itself will be useful to the growing number of critical software developers adopting agile techniques such as "continuous integration". As a matter of fact, providing daily coverage metrics on the system being continuously integrated provides an inestimable too for evaluating project advancement.

References

[1] QEMU, a generic and open source machine emulator and virtualizer.

[2] AdaCore. Couverture project documentation, 2010.

[3] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[4] CAST, Certification Authorities Software Team. Rationale for accepting Masking MCDC in certification projects. Position Paper 6, August 2001.

[5] CAST, Certification Authorities Software Team. Guidelines for Approving Source Code to Object Code Traceability. Position Paper 12, December 2002.

[6] CAST, Certification Authorities Software Team. Structural Coverage of Object Code. Position Paper 17, June 2003.

[7] CENELEC. Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. European standard EN 50128:2001, Brussels, Belgium, Mar 2001.

[8] Emmanuel Chailloux, Adrien Joncquet, and Philippe Wang. Non Intrusive Structural Coverage for Objective Caml. *BYTECODE 2010, 5th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, March 2010.

[9] John J. Chilenski. An Investigation of Three Forms of the Modified Condition/Decision Coverage (MCDC) Criterion. Technical Report DOT/FAA/AR-01/18, April 2001.

[10] FAA, Federal Aviation Administration. Object Oriented Technology Verification Phase 3 Report - Structural Coverage at the Source Code and Object Code Levels. Technical Report DOT/FAA/AR-07/20, June 2007.

[11] European Cooperation for Space Standardization (ECSS). Space engineering — software. ECSS standard ECSS-E-ST-40C, ESA-ESTEC, Requirements & Standards Division, Mar 2009.

[12] IEC. Nuclear power plants — Instrumentation and control systems important to safety — Software aspects for computer-based systems performing category A functions. IEC standard 60880:2006 (2nd edition), Geneva, Switzerland, May 1986.

[13] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, 1988.

[14] RTCA. Software considerations in airborne systems and equipment certification. Document RTCA DO-178B, 1992.

[15] RTCA. Final annual report for clarification of do-178b ”software considerations in airborne systems and equipment certification”. Document RTCA DO-248B, 2001.