LDRA
Software Technology

Technical Briefing

The significance of Object Code Verification: An overview

www.ldra.com

Introduction

Verification and validation practices championed by functional safety, security and coding standards (including IEC 61508 [1], ISO 26262 [2], IEC 62304 [3], MISRA C [4] and C++ [5], CWE [6]...) place considerable emphasis on showing how much of an application under test has been exercised. Experience has shown us that if all code has been shown to perform correctly, then the probability of failure in the field is considerably lower. And yet the focus is on the high level source code – whether that is written in C, C++, or whatever. Such an approach assumes that the compiler will create object code that faithfully reproduces what the developers intended.

It is inevitable that the control and data flow of object code will not be an exact mirror of the source code from which it was derived, and so proving that all source code paths can be exercised reliably does not prove the same thing of the object code. Worse, and despite their undeniable value, source code unit tests can also be misleading because the object code derived from the compilation of a function wrapped in a unit test harness can differ markedly from that generated in the context of a complete system.

Only DO-178C [4], used in the aerospace industry, places any focus on the potential for dangerous inconsistencies between developer intent and executable behaviour – and yet the fact remains that the differences between source and object code can have devastating consequences in ANY critical application.

Object code verification: Tool qualification example

It is useful to understand how and why the control flow structure of compiler-generated object code differs from that of the application source code from which it was derived by reference to some very simple C source code from an LDRA tool qualification pack [8].

```
void f_while4( int f_while4_input1, int f_while4_input2 )
{
  int f_while4_local1, f_while4_local2 ;

  f_while4_local1 = f_while4_input1 ;
  f_while4_local2 = f_while4_input2 ;

   while( f_while4_local1 < 1 ||  f_while4_local2 > 1 )
     {
        f_while4_local1 ++ ;
        f_while4_local2 -- ;
     }
}
```

This C code can be demonstrated to achieve 100% source code coverage by means of a single call thus:

```
f_while4(0,3);
```

The code can be reformatted to a single operation per line and represented on a flowgraph as a collection of “basic block” nodes, each of which is a straight line code sequence (Figure 1). “Directed edges” (lines) between the nodes illustrate the relationships between them.

The “as-if” rule

When source code is compiled, the flowgraph for the resulting assembler code is likely to be quite different to that for the source. The rules followed by C or C++ compilers permit them to modify the code in any way they like, provided the binary behaves as if it were the same.

From the C++ standard §1.9/1 [9]:
“This provision is sometimes called the “as-if” rule, because an implementation is free to disregard any requirement of this document as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.”

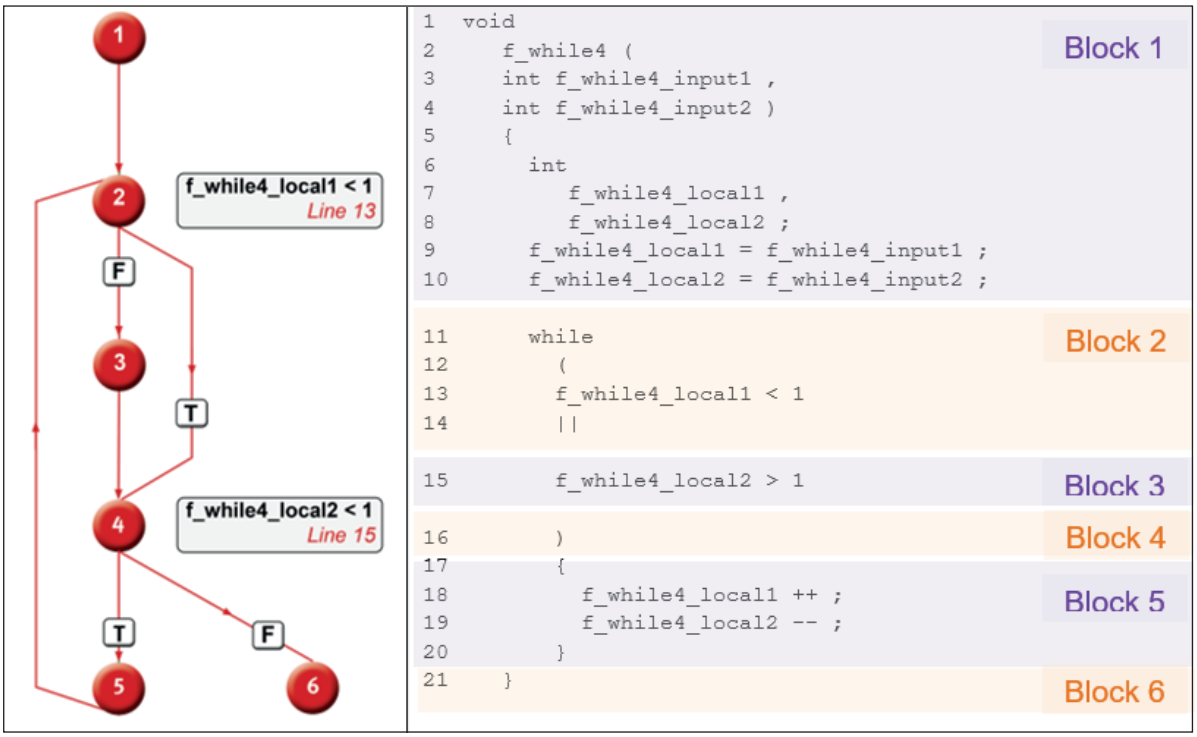

Figure 1: Original source code reformatted to a single operation per line and represented on a flowgraph as a collection of “basic block” nodes, each of which is a sequence of straight line code

When the code is compiled using any widely used commercially available compiler, the flowgraph is likely to look different to that of the source code (Figure 2). In this example, using a TI compiler with optimization disabled, the result is as shown below. The red elements of the flow graph below represent code that has not been exercised by the call

```
f_while4(0,3);
```

By leveraging the one-to-one relationship between object code and assembler code, this mechanism exposes which parts of the object code are unexercised, prompting the tester to devise additional tests and achieve complete assembler (and hence object code) coverage.

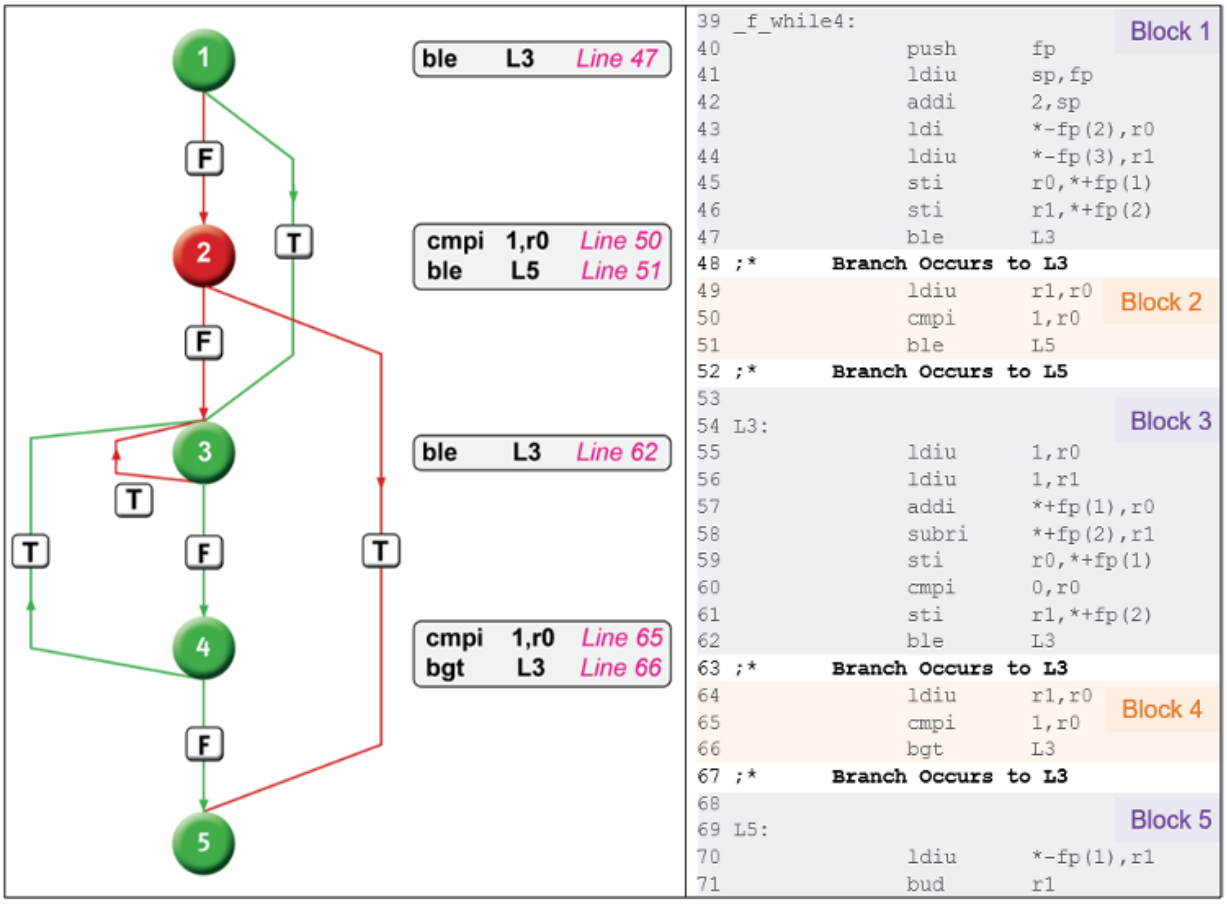

Figure 2: Despite the full coverage of the source code, code coverage analysis of the object code reveals several unexercised paths

Devising additional tests

Additional tests can be devised at source code level by assessing the logic of the object code. For example, the ble branch (line 62, end of block 3, branch to L3) always evaluates to false with the existing test data because it only exercises the loop once, and so only one of the two possible outcomes results from the test to see whether to continue. Adding a new source code test case to ensures a second pass to exercise the true case too (Figure 3).

```
f_while4(-1,3);
```

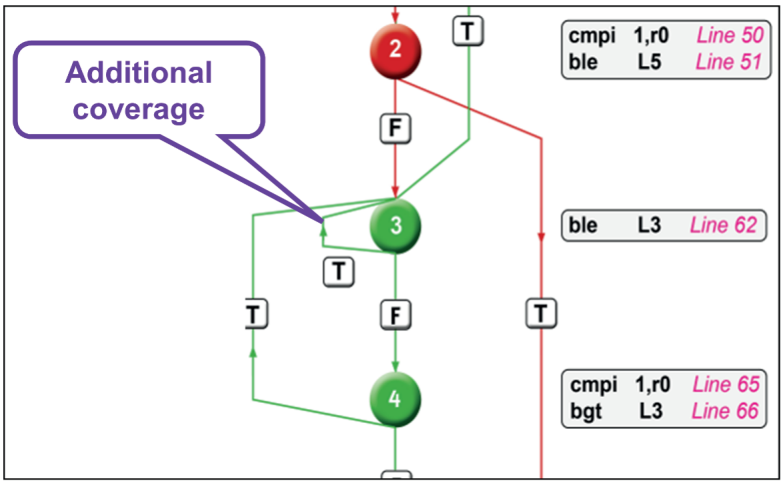

Figure 3: Additional object code coverage results from the application of an additional test case f while(-1,3)

Achieving 100% object code coverage

A similar approach to assessing the logic of the object code for the branches at lines 47 and 52 results in two further test cases. A main function to exercise all object code in the example therefore looks like this:

```
int main()
{
  int a = 0;
  int b = 1;
  int c = 0;

  a = b || c;

  /* WHILE_WITH_OR.C */

  f_while4(0,3); //Achieves 100% source code coverage
  f_while4(-1,3);
  f_while4(3,3);
  f_while4(3,0);

  return(0);
}
```

DO-178C and Object Code Verification

With that premise established it is useful to return to DO-178C and what is required in the aerospace domain and particularly paragraph 6.4.4.2, entitled "Structural Coverage Analysis". The standard reasons that structural coverage is required because it is the best way to ensure that requirements-based tests have completely exercised the application:

"The objective of this analysis is to determine which code structure was not exercised by the requirements-based test procedures. The requirements-based test cases may not have completely exercised the code structure, so structural coverage analysis is performed and additional verification produced to provide structural coverage."

Paragraph 6.4.4.2b describes the additional requirements when the application is Level A:

"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…"

In other words, if the application is Level is A and the compiler is generating object code that is not traceable to the source code, then additional verification of the object code must be performed. However, there is no obligation in DO-178C to automate this exercise at all. Underlining that point, the (now defunct) 2002 FAA paper CAST-12, "Guidelines for Approving Source Code to Object Code Traceability" [10] outlined not only how a manual review might be achieved, but also how a sample application might be devised using typical constructs to optimize the time spent performing such a manual analysis.

The alien world of functional safety

The problem with such an approach lies with the specifications that compiler developers are obliged to fulfil, coupled with the flexibility afforded to them on the basis of the "as-if" principle. Functionally safe code is required to include code to defend against unexpected events which can result from a disparate variety of causes. For example, memory corruption through coding errors or cosmic ray events can lead to the execution of code paths that are "impossible" according to the logic of the code.

High level languages, particularly C and C++, also include a surprising number of features whose behaviour is not prescribed by the language specification to which the code adheres. This undefined behaviour can clearly lead to unexcepted and potentially disastrous outcomes that must also be defended against in a functionally safe application. Code is also required to lend itself to the achievement of high levels of code coverage, and in some sectors – particularly automotive - it is quite common to have requirements for the design to accommodate sophisticated external diagnostic, calibration and development tools.

But practices such as defensive coding and external data access are not part of a world the compiler recognizes. For example, neither C nor C++ make any allowances for memory corruption, so that unless code designed to protect against it is accessible when there is no such corruption, it may be ignored when the code is optimized. Defensive code must be syntactically and semantically reachable if it is not to be "optimized away".

Instances of undefined behaviour can also cause surprises. It is easy to suggest that they should be simply avoided, but it is often quite difficult to identify them. Where they exist, there can be no guarantee that the behaviour of the compiled executable code matches the developers' intentions. The "back door" access to data used by debug tools represents yet another situation that the language makes no allowance for, and so can have unexpected consequences.

Compiler optimization can have a major impact on all of these areas, because none of them is part of the remit for compiler vendors. Optimization can result in apparently sound defensive code being eliminated where it is associated with "infeasibility" – that is, where it exists on paths which can't be tested and verified by any set of possible input values. Even more alarmingly, defensive code shown to be present during unit testing may well be eliminated when the system executable is constructed. Just because coverage of defensive code has been achieved during unit test therefore doesn't guarantee that it is present in the completed system. Detailed worked examples illustrating the potential impact of this problem can be found in the LDRA white paper, "Object Code Verification: Why it matters" [11].

How much OCV is sufficient?

With the focus on the DO-178C requirements, despite the additional clarification as compared with DO-178B the standard remains vague at best in this regard. The onus is on the development team to choose a path that can be justified in the context of the work they are doing, and the risks involved should that work introduce errors.

In each of the following three cases, instances of different coverage outcomes achieved at the object level are indications of differing code structure and therefore provide the opportunity to review and document these differences. It should be stressed it is quite common for additional branch structure in the object code to be infeasible, making it impossible to achieve 100% coverage at the object level. That is not the point of the exercise, and so failing to do so does not imply a failure to satisfy the DAL-A requirements of the standard.

Tool qualification pack OCV

The example function `f_while4` was taken from an LDRA tool qualification support pack. That support pack consists of a set of constructs encompassing all of those likely to be used within a safety critical system, to allow each to be individually proven. This supports the notion that if the compiler is proven to be capable of correctly interpreting every construct, then when those constructs are used as part of a complete application they will remain trustworthy. The LDRA tool suite is equipped to support such an approach (Figure 8).

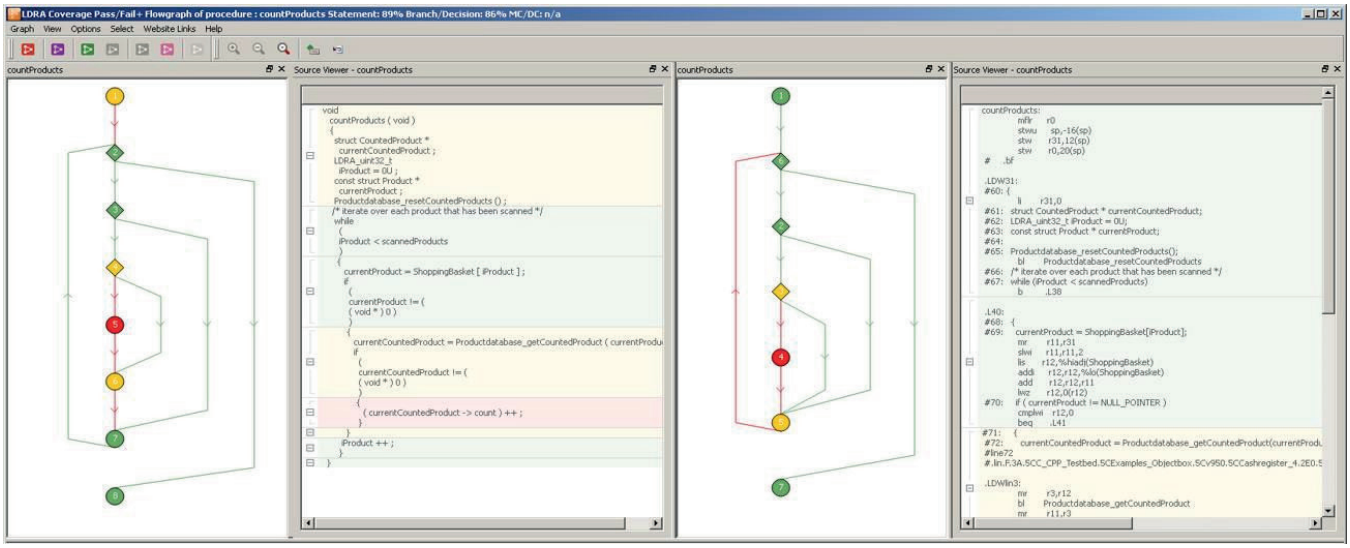

Figure 8: Using the LDRA tool suite to compare coverage of source and object code

Unit test OCV on application code

A more commonplace compromise is to apply OCV to the same unit tests leveraged to complete the source code analysis, compare coverage results, and investigate any results that differ between the object code and the source code.

Clearly this represents a better solution. In this scenario the code under test is part of the application, unlike the tool qualification pack examples. The scope for mismatch between the object code created for a function as part of a test harness and that generated for the same function as part of the complete application code base remains, as the result from the “as if” rule. But application code code provides a quick and easy way of identifying differences in structure between the high-level and object-level code – a clearly stated requirement of the DO-178C standard.

OCV on complete application code

That said, the scope for mismatch between compiled code and the developer’s intentions exacerbated by the implications of the “as if” rule is not easy to ignore. It makes it clear that the most accurate and comprehensive testing can only be assured by undertaking OCV of a complete application. The facilities provided by the LDRA tool suite make that possible.

Is it necessary? Certainly not from the perspective of any standard, including DO-178C. Is it best practice? It’s hard to argue against that position. Is it commercially justifiable? That is a judgement to be made on the part of the development team.

Conclusions

The discussion in this paper centres on the C high level language. However, the challenges highlighted apply to a greater or lesser extent irrespective of the high level language selected. For example, the nature of C++ in general, and in particular extensions introduced by later variants such as C++14 and C++17, means that more elements in the executable (and hence object code) are untraceable to the source.

The example code is also very simple. Clearly in a live project, there is much more code to consider, often including third party code such as libraries which also need to be subjected to the same level of rigorous analysis as the rest of the code base. Best practice in such cases is to deploy to certified libraries, or to subject the third party to code to the same validation and verification activities as code developed internally.

DO-178C level (or DAL) A applications are representative of the critical applications at risk. Object Code Verification (OCV) provides a tool that is very useful in ensuring that the demands of the standard are met. But it is a fact that those demands are vague and leave it to the judgement of the development team to decide what is appropriate and necessary, which may not be the ultimate in best practice but some practical compromise.

The differing approaches to the application of OCV discussed in this paper are best considered as part of an overall strategy to inspect and justify inconsistencies between source and object code. They may be applied in combination to different parts of the system, perhaps in tandem with other techniques including manual code inspection, and the use of debugging tools to trace execution paths.

OCV is not obligatory for any standard, in any sector. There are ways around it even in civil aviation where there have been CAST and other papers that explain other approaches. But to bypass OCV altogether represents much more of a compromise than anything discussed in this document.

Issues resulting from the mismatch between the remit of complier developers and the demands of functional safety might well go undetected without the use of OCV. Compilers used for highly critical software applications can generally be assumed to fulfil their design criteria in all but very unusual circumstances, but those criteria do not include functional safety considerations. Object code verification currently represents the most assured approach to bringing those considerations within that circle of trust.

Works cited

1. International Electrotechnical Commission (IEC), IEC 61508:2010 “Functional safety of electrical/electronic/programmable electronic safety-related systems” (all parts), IEC, 2010.
2. International Organization for Standardization, ISO 26262:2018 “Road vehicles - Functional safety”, International Organization for Standardization, 2018.
3. International Electrotechnical Commission, IEC 62304:2006+AMD1 Medical device software - Software life cycle processes, IEC, 2015.
4. MISRA, “MISRA C:2012 Third Edition, First Revision,” The MISRA Consortium Limited, February 2019. [Online]. Available: https://www.misra.org.uk/product/misra-c2012-third-edition-first-revision/. [Accessed 24 September 2021].
5. MISRA, “MISRA C++:2008,” The MISRA Consortium Limited, June 2008. [Online]. Available: https://www.misra.org.uk/product/misra-c2008/. [Accessed 24 September 2021].
6. Homeland Security Systems Engineering and Development Institute, “CWE Commone Weakness Enumeration,” The MITRE corporation, August 2021. [Online]. Available: https://cwe.mitre.org/. [Accessed 24 September 2021].
7. RTCC, DO-178C “Software Considerations in Airbourne Systems and Equipment Certification”, RTCA, 2011.
8. LDRA, “LDRA Tool Qualification Support Packs (TQSP),” 2021. [Online]. Available: https://ldra.com/maccordion/tool-qualification-support-pack-tqsp/. [Accessed 24 September 2021].
9. ISO, “ISO/IEC 14882:2020 Programming languages — C++,” December 2020. [Online]. Available: https://www.iso.org/standard/79358.html. [Accessed 24 September 2021].
10. Certification Authorities Software Team, “Position Paper CAST-12 - Guidelines for Approving Source Code to Object Code Traceability,” CAST, 2002.
11. LDRA, “Object Code Verification: Why it matters,” LDRA, Monks Ferry, 2021.

LDRA UK & Worldwide
Portside, Monks Ferry,
Wirral, CH41 5LH
Tel: +44 (0)151 649 9300
e-mail: info@ldra.com

LDRA Technology Inc.
2540 King Arthur Blvd, 3rd Floor, 12th Main, Lewisville, Texas 75056
Tel: +1 (855) 855 5372
e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
Unit B-3, Third floor Tower B, Golden Enclave
HAL Airport Road Bengaluru 560017
Tel: +91 80 4080 8707
e-mail: india@ldra.com