LDRA
Software Technology

Technical White Paper

Object Code Verification: Why it Matters

www.ldra.com

Contents

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 code has been shown to perform correctly, then the probability of failure in the field is considerably lower. And yet almost without exception, 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 even then, it is not difficult to find advocates of workarounds with clear potential to leave those inconsistencies undetected. However such approaches are excused, the fact remains that the differences between source and object code can have devastating consequences in ANY critical application. This paper will explain why object code verification, or OCV, can make a major contribution to a best practice process for any system for which there are dire consequences associated with failure – and indeed, for any system where only best practice is good enough.

About the examples

The examples in this paper all use 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 examples are 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.

Object code verification: Tool qualification example 1

Initially, 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 – in fact, an example from an LDRA tool qualification pack [8]. A second example from a tool qualification pack is shown in the appendix.

```
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 -- ;
     }
}
```

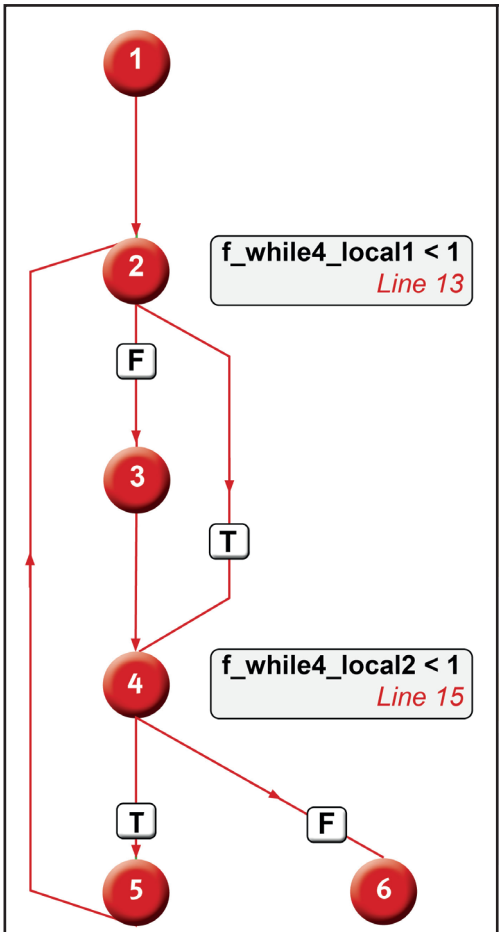
This C code can be demonstrated to achieve 100% source code coverage by means of a single call thus:

```
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 sequence of straight line code (Figure 1). The relationship between the basic blocks is represented using directed edges between the nodes

When source code is compiled, the flowgraph for the resulting assembler code is likely to be quite different to that for the source because 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."




```
1   void
2      f_while4 (                                          Block1
3      int f_while4_input1 ,
4      int f_while4_input2 )
5      {
6         int
7             f_while4_local1 ,
8             f_while4_local2 ;
9         f_while4_local1 = f_while4_input1 ;
10        f_while4_local2 = f_while4_input2 ;

11        while                                            Block 2
12          (
13          f_while4_local1 < 1
14          ||

15          f_while4_local2 > 1                            Block 3

16          )                                              Block 4
17          {                                              Block 5
18            f_while4_local1 ++ ;
19            f_while4_local2 -- ;
20          }
21    }                                                    Block 6
```

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 code coverage – and hence achieve object code coverage.

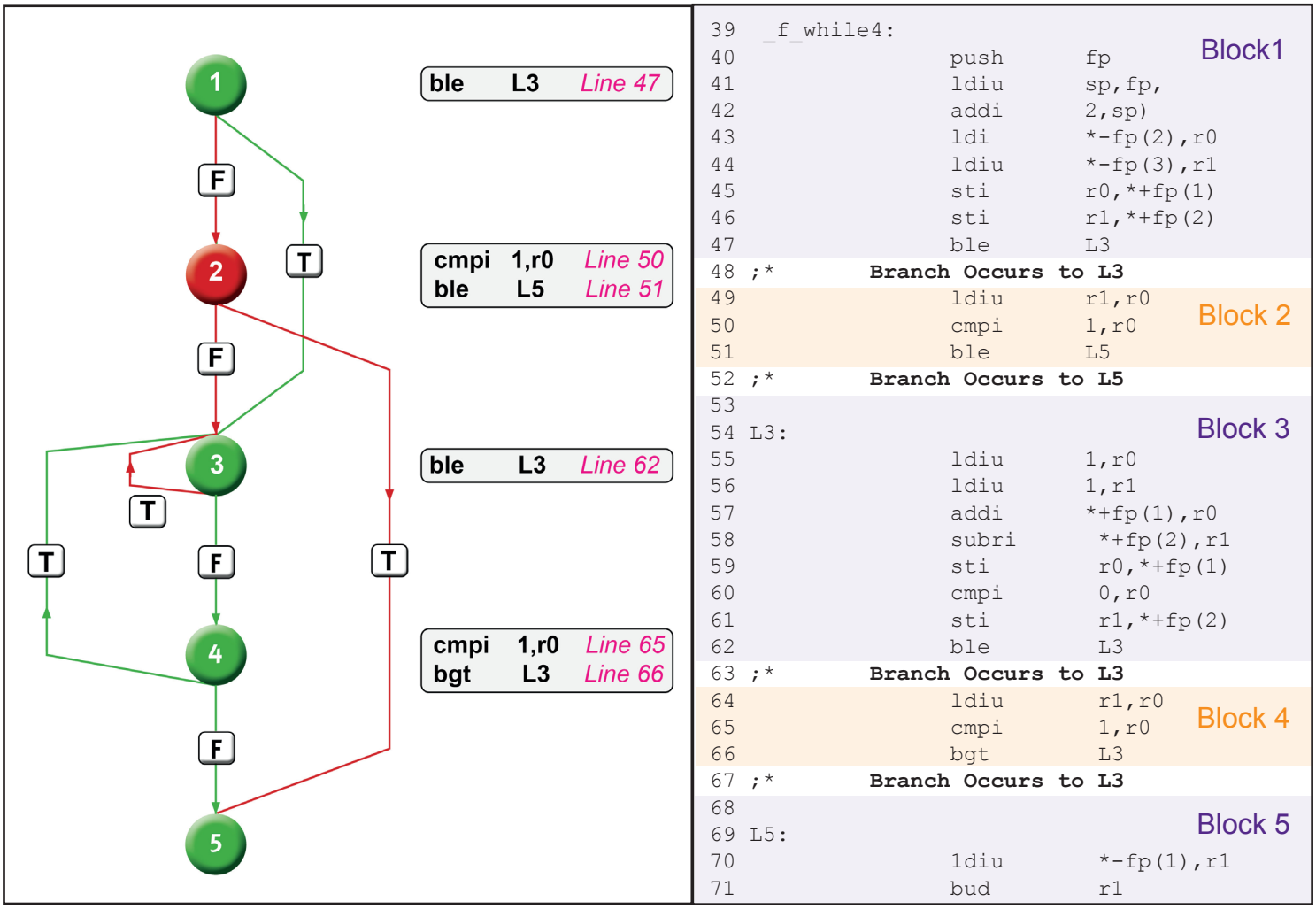


Figure 2: Despite the full coverage of the source code, code coverage analysis of the object code reveals several unexercised paths

Devising additional tests

New test 1. Line 62. End of block 3. Branch to L3

This `ble` branch 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 ensure a second pass around that loop exercises both true and false cases.

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

Figure 3 shows the cumulative coverage from the two test cases.

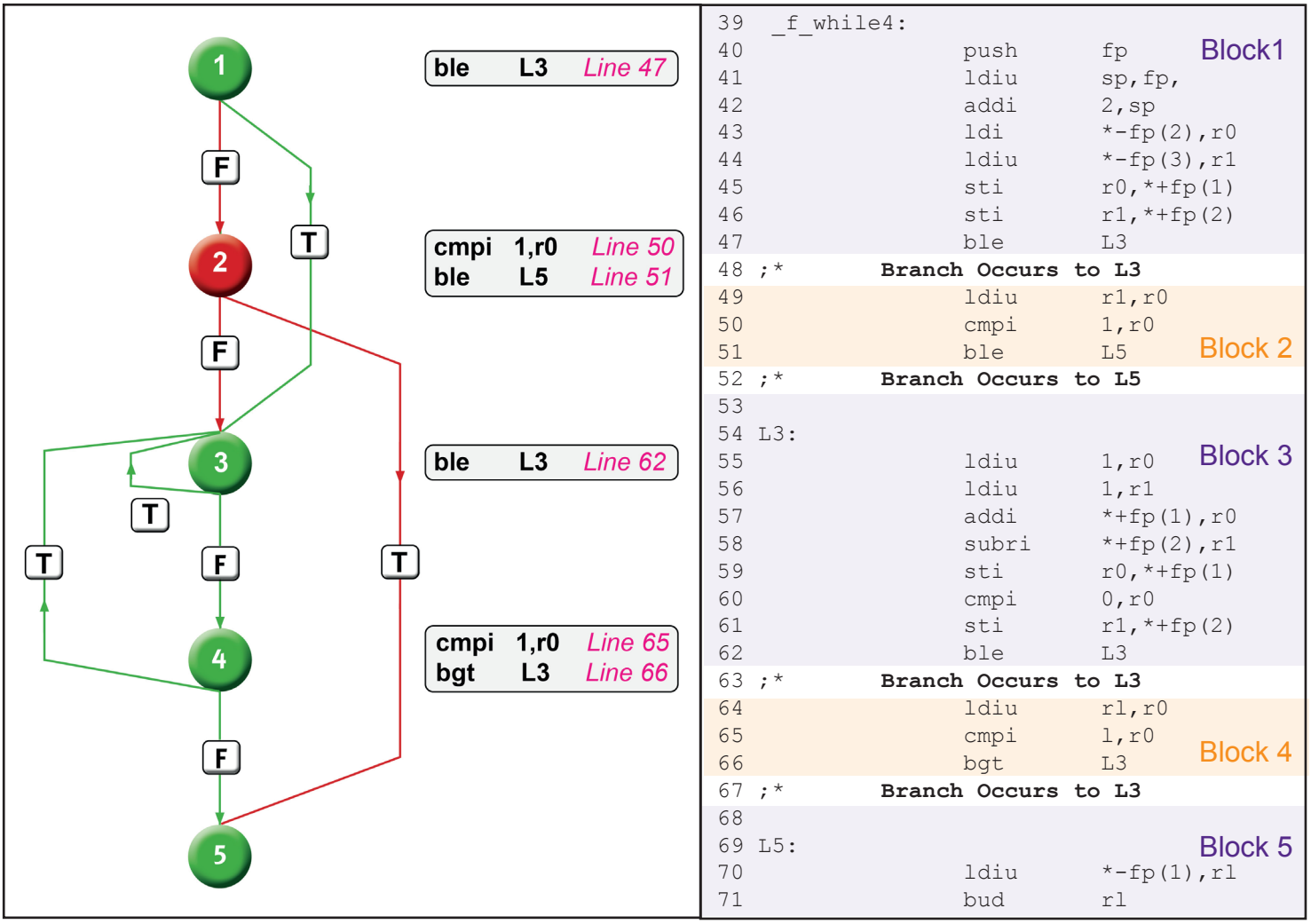


Figure 3: Additional object code coverage results from the application of an additional test case `f while(-1,3)`

New test 2. Line 47. End of block 1. Branch to L3

The source code contains an `or` statement in the `while` loop conditions. The existing test cases both result in the code `f_while4_local1 < 1` returning a "true" value.

The addition of a new test case to return a "false" value will address that:

```
f_while4(3,3)
```

Figure 4 shows how the new test case improves the cumulative coverage again:

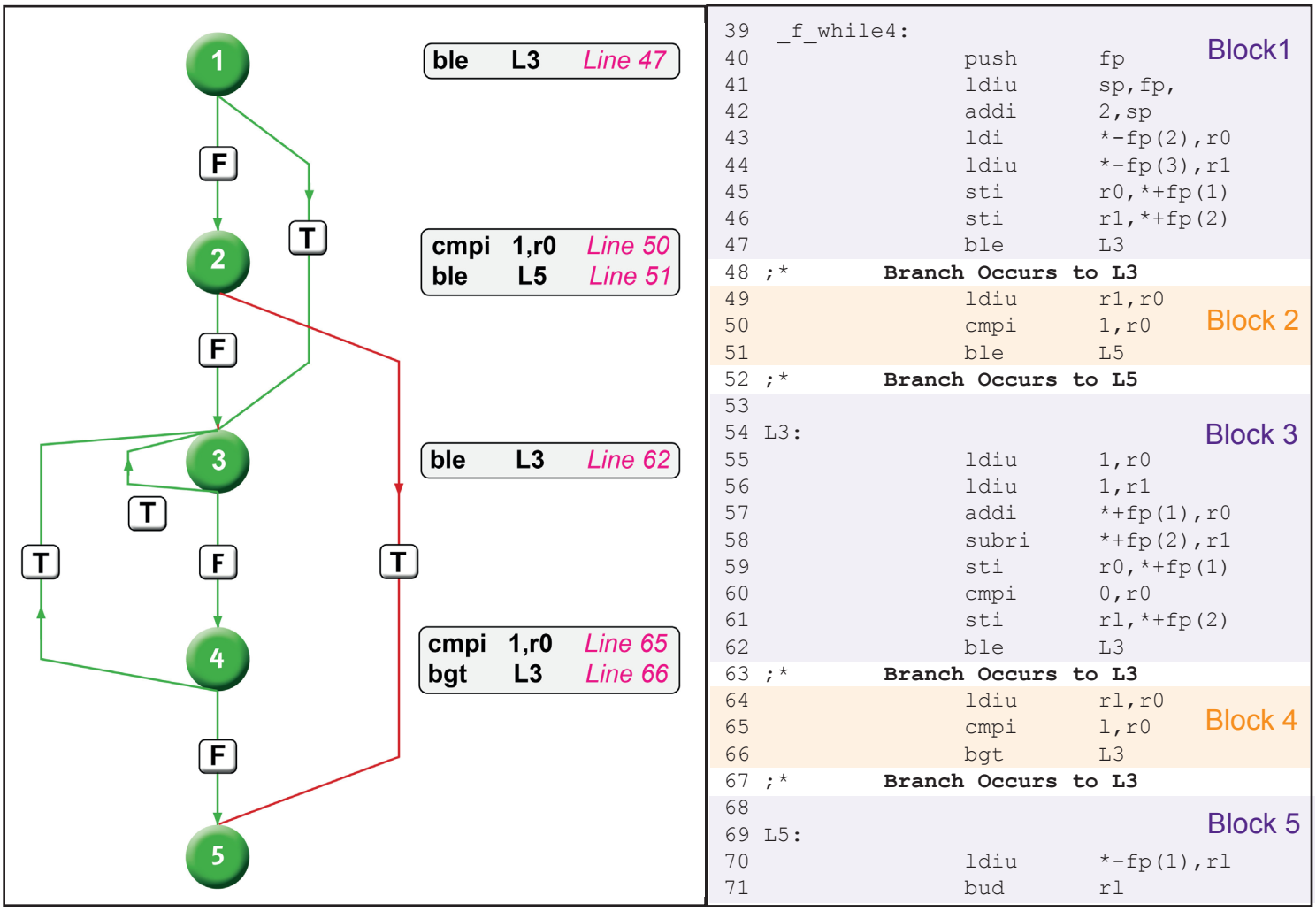


Figure 4: Additional object code coverage results from the application of an additional test case `f_while(3,3)`

New test 3. Line 52. End of block 2. Branch to L5

The remaining unexercised branch is the result of the fact that if neither of the initial conditions in the source code `while` statement is satisfied, then the object code within the loop is bypassed altogether via the `ble` branch.

So, the final test is added to provide such a circumstance.

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

This results in 100% statement and branch coverage as shown in Figure 5.

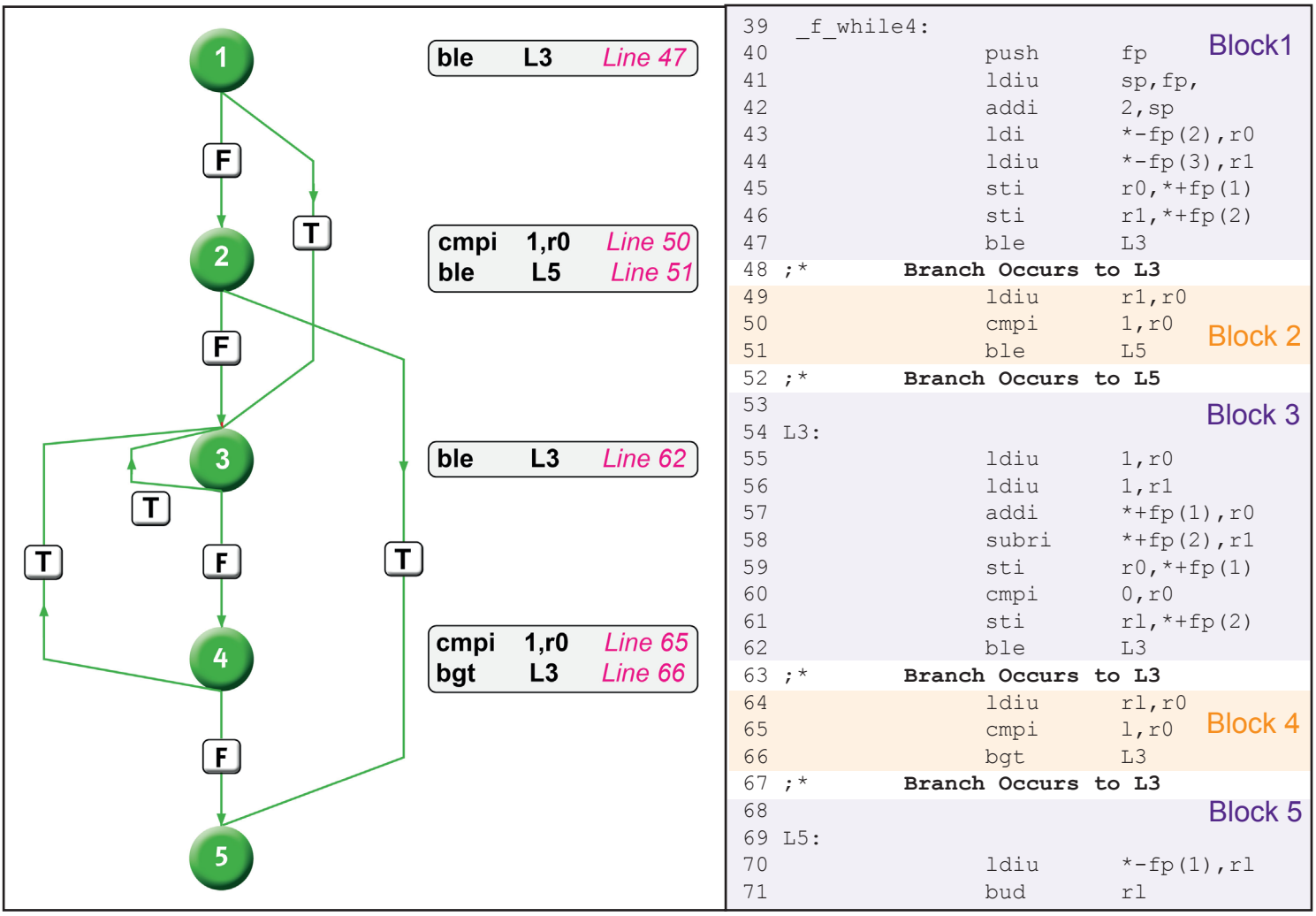


Figure 5: Full object code coverage is finally completed using test case `f_while4(3,0)`

A "main" function to exercise all object code in the example would look 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. A compiler-generated array-bound check in the object code is an example of object code that is not directly traceable to the Source Code.*"

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.

Some might argue that if the stated aim of structural code analysis is to "*completely exercise the code structure*" then that hasn't happened if object code paths remain unexecuted.

Putting that to one side, however, there is no obligation in DO-178C to automate this exercise or indeed to achieve object code coverage. In fact, as for elsewhere in the standard – and, indeed, for almost all development work governed by functional safety standards across the sectors – there is no obligation to automate anything. 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.

But irrespective of sector, does such an approach in isolation really represent best practice? And can it possibly be justified for the most critical of all software applications?

Compilers in the world of functional safety

To begin to answer those questions, it is useful to reflect on the obligations placed on functionally safe software in general. Across the sectors, standards require that defensive coding is applied, that code is testable, that it is possible to collate adequate code coverage, and that application code is traceable to requirements to ensure that the system implements them fully, and uniquely.

Functionally safe code is required to include defensive 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++, 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.

The problem that arises is that 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 simply be ignored when the code is optimized. Consequently, 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 match 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.

Certified compilers

Software development requires many tools including design tools, code generation tools, compilers/linkers, libraries, test tools, and structural coverage tools. DO-178 tool qualification pertains to development and testing tools, including compilers.

Some compiler vendors offer certified compilers that support this qualification process, which makes it more efficient. However, the compilers themselves will still adhere to the rules dictated by the high level language in use. In short, none of the challenges outlined here can be addressed simply by using such a compiler.

Trusting your compiler

Compiler vendors are honourable people, doing a very technically demanding and professional job. Of course there will be bugs – as in any other software – but usually, a compiler's implementation will fulfil its design requirements. However, as has been demonstrated, those requirements do not always reflect the needs of a functionally safe system.

In short, then, a compiler can usually be assumed functionally true to the objectives of its creators. But that may not be entirely what is wanted or expected, as best illustrated by reference to another example - this time resulting from compilation with the CLANG compiler (Figure 6).

```
extern void error ( void );

static enum { S0 = 13, S1 = 23 } state = S0;

bool update ( void )
{
  switch ( state )
  {
    case S0: state = S1; break;
    case S1: state = S0; break;
    default: error();    break;
  }
  return state == S0;
}
```

Constant values are NOT expressed within assembler – optimized to use '0' and '1'

```
update(): # @update()
  mov al, byte ptr [rip + state]
  mov ecx, eax
  xor cl, 1
  mov byte ptr [rip + state], cl
  ret
```

Figure 6: Understanding the implications of the "as if" obligation

It is clear that the defensive call to the `error` function has not been represented in the assembler code.

The `state` object is only modified when it is initialized, and within the `S0` and `S1` cases and so the compiler can reason that the only values given to `state` are `S0` and `S1`. The compiler concludes that the `default` is not needed because `state` will *never* hold any other values, assuming that there is no corruption – and indeed, the compiler makes exactly that assumption.

The compiler has also decided that because the values of the actual objects (`13` and `23`) are not used in a numeric context, it will simply use the values of 0 and 1 to toggle between states and then use an exclusive "or" (`xor`) to update the state value. The binary adheres to the "as if" obligation, and the code is fast and compact. Within its terms of reference, the compiler has done a good job.

This behaviour has implications for such as "calibration" tools that use the linker memory map file to access objects indirectly, and for direct memory access via a debugger. Again, such considerations are not part of the compiler's remit and are therefore not considered during optimization and/or code generation.

Now suppose the code remains unchanged, but its context in the code presented to the compiler changes slightly (Figure 7).

```
extern void error ( void );

static enum { S0 = 13, S1 = 23 } state = S0;

bool update ( void )
{
  switch ( state )
  {
    case S0: state = S1;
    case S1: state = S0;
    default: error();
  }
  return state == S0;
}

int f ( void )
{
  return state;
}
```

Constant values are still NOT expressed within the assembler for update(), but they are considered when converting to int within f()

```
update(): # @update()
  mov al, byte ptr [rip + state]
  mov ecx, eax
  xor cl, 1
  mov byte ptr [rip + state], cl
  ret
```

```
f(): # @f()
  mov al, byte ptr [rip + state]
  test al, al
  mov ecx, 23
  mov eax, 13
  cmovne eax, ecx
  ret
```

Figure 7: The implications of an additional function returning the value of the state variable

There is now an additional function which returns the value of the state variable as an integer. This time the absolute values `13` and `23` matter in the code submitted to the compiler. Even so, those values are not manipulated within the update function (which remains unchanged) and are only apparent within our new `f` function.

In short, the compiler continues (rightly) to make value judgements about where the values of `13` and `23` should be used - and they are by no means applied in all situations where they might be.

If the new function is changed to return a pointer to our state variable, the assembler code changes substantially (Figure 8). Because there is now the potential for alias accesses through a pointer, the compiler can no longer deduce what is happening with the state object. It cannot conclude that the values of `13` and `23` are unimportant and so they are now expressed explicitly within the assembler.

```
extern void error ( void );

static enum { S0 = 13, S1 = 23 } state = S0;

bool update ( void )
{
  switch ( state )
  {
    case S0: state = S1; break;
    case S1: state = S0; break;
    default: error();    break;
  }
  return state == S0;
}

void * f ( void )
{
  return &state;
}
```

```
update(): # @update()
  push rax
  mov eax, dword ptr [rip + state]
  cmp eax, 23
  je .LBB0_3
  cmp eax, 13
  jne .LBB0_4
  mov dword ptr [rip + state], 23
  xor eax, eax
  jmp .LBB0_5
.LBB0_3:
  mov dword ptr [rip + state], 13
  mov al, 1
  jmp .LBB0_5
.LBB0_4:
  call error()
  cmp dword ptr [rip + state], 13
  sete al
.LBB0_5:
  pop rcx
  ret
```

Figure 8: Returning a pointer to the state variable sees the resulting assembler code change substantially

The implications of the "as if" rule for source code unit test

Now consider the example in the context of an imaginary unit test harness (Figure 9). This harness deliberately manipulates the value of the state variable, so that the default is not "optimized away" and hence coverage of the default can be obtained. Such an approach is entirely justifiable in a test tool which has no context relating to the remainder of the source code, and which is required to make everything accessible.

The compiler recognises that an arbitrary value is written to the state variable via a pointer, and again, it cannot conclude that the values of `13` and `23` are unimportant. Consequently they are now expressed explicitly within the assembler. On this occasion it cannot conclude that `S0` and `S1` represent the only possible values for the state variable which means that the default path may be feasible. The manipulation of the state variable achieves its aim and the call to the `error` function is now apparent in the assembler.

```
extern void error ( void );

static enum S { S0 = 13, S1 = 23 } state = S0;

bool update ( void )
{
  switch ( state )
  {
    case S0: state = S1; break;
    case S1: state = S0; break;
    default: error();    break;
  }
  return state == S0;
}

bool testDefault ( void )
{
  *( int * )&state = -1;
  return update();
}
```

This changes everything!

```
update(): # @update()
  push rax
  mov eax, dword ptr [rip + state]
  cmp eax, 23
  je .LBB0_3
  cmp eax, 13
  jne .LBB0_4
  mov dword ptr [rip + state], 23
  xor eax, eax
  jmp .LBB0_5
.LBB0_3:
  mov dword ptr [rip + state], 13
  mov al, 1
  jmp .LBB0_5
.LBB0_4:
  call error()
  cmp dword ptr [rip + state], 13
  sete al
.LBB0_5:
  pop rcx
  ret
testDefault(): # @testDefault()
  mov dword ptr [rip + state], -1
  jmp update() # TAILCALL
```

Figure 9: The implications of the "as if" rule for source code unit test[1]

However, this manipulation will not be present in the code that will be shipped within a product, and so the call to `error()` is not really there in the complete system.

Clearly, that raises a serious question. If the compiler is going to handle code differently in the test harness as compared to the completed application, then is source code unit test coverage – or even object code unit test coverage - really of any worth?

The answer must be a qualified "yes". Many systems have been certified on the evidence of such artefacts, and proven safe and reliable in service. But the fact remains that for the most critical of systems across all sectors, if the development process is to withstand the most detailed of scrutiny and adhere to best practice then source level unit test coverage must be supplemented by OCV of the application as a whole.

Addressing the issues

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 code 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.

[1] It is acknowledged that the "unit test" code in this example contains undefined behaviour where a pointer to '`S`' is converted to a pointer to an unrelated type (`int*`)

Tool qualification pack OCV

Two examples were leveraged within this document. The first of these, 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 10).

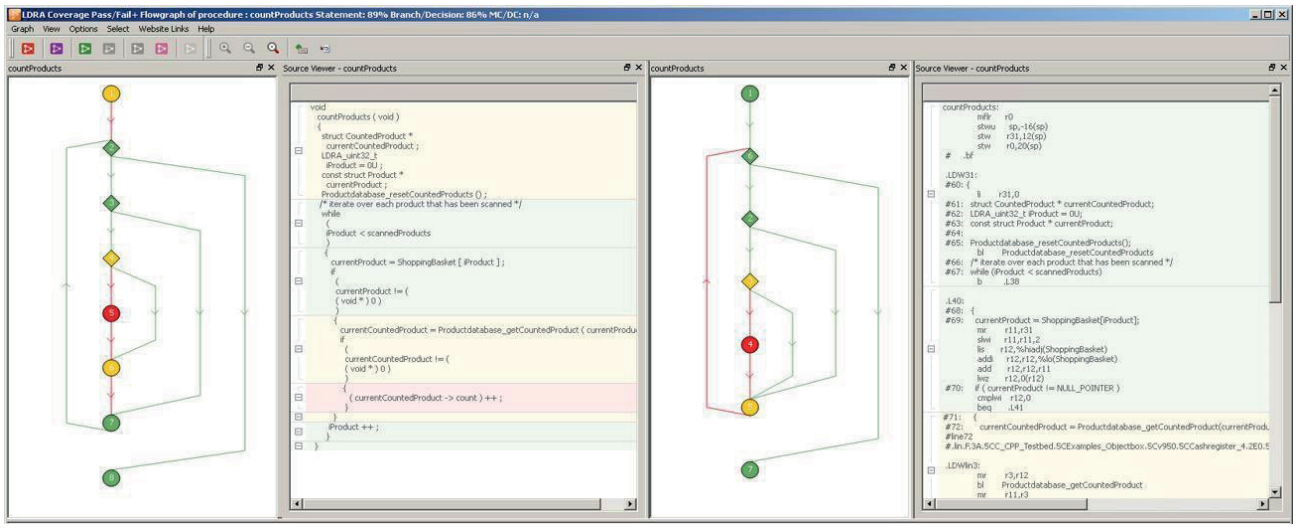


Figure 10: Using the LDRA tool suite to compare coverage of source and object code

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 such an approach alone is a better solution than relying solely on tool qualification pack unit test. After all, in this scenario the code under test is part of the application, unlike the tool qualification pack examples and so despite the reservations highlighted by the `update` example, it will undoubtedly be more representative of the application that will run on the target.

Coverage of application code at the object level is not a stated requirement of DO-178C standard but it does provide a quick and easy way of identifying differences in structure between the high-level and object-level code, which IS a requirement of the standard.

OCV on complete application code

Despite that position, the second example in this paper (`update`) 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

These differing approaches to the application of OCV (and the tools that facilitate them) are best considered as part of an overall strategy to inspect and justify inconsistencies between source and object code. Different approaches may be taken for different parts of the system, which might include the techniques mentioned here – perhaps in combination – as well as other techniques including manual code inspection, and the use of debugging tools to trace execution paths.

Returning to the documented requirements of DO-178C, level (or DAL) A applications represent exactly the kind of critical application at risk. Object Code Verification 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.

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 even more of a compromise and a move away from best practice than anything discussed in this document.

Neither does it stop there. The mismatch between the remit of complier developers and the demands of functional safety also has implications relating to defensive code, and to the use of some development tools. It can also hinder the detection of issues including array bounds error detection and the handling of undefined behaviour, to quote just two examples.

Such issues might well go undetected without the use of OCV. They manifest the fact that 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.

Appendix: Tool qualification example 2

The following source code represents another simple example from an LDRA tool qualification pack.

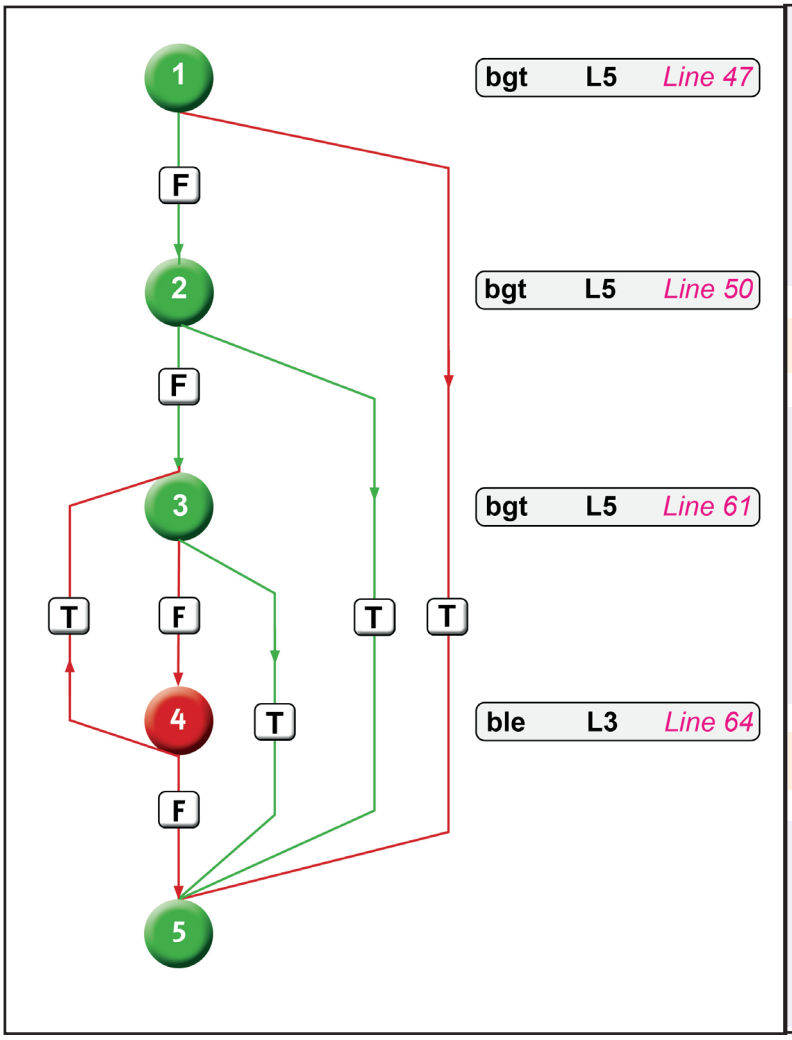
```
void f_while10( int f_while10_input1, int f_while10_input2 )
{

  int f_while10_local1, f_while10_local2 ;
  f_while10_local1 = f_while10_input1 ;
  f_while10_local2 = f_while10_input2 ;
  while( f_while10_local1 < 1 && !( f_while10_local2 > 0 ) )
  {
    f_while10_local1 ++ ;
    f_while10_local2 -- ;
  }
}
```

The standard test cases provided with this example to provide 100% coverage of the C code consists of two calls to the function `f_while10`, as follows:

```
f_while10(0,0)
f_while10(0,1)
```

For many compilers the assembler code derived from these calls is exercised to provide 100% statement and branch coverage but in this instance, the resulting coverage is lower.




```
39  _f_while10:                                          Block1
40                  push        fp
41                  ldiu        sp,fp
42                  addi        2,sp
43                  ldi         *-fp(2),r0
44                  ldiu        *-fp(3),r1
45                  sti         r0,*+fp(1)
46                  sti         r1,*+fp(2)
47                  bgt         L5
48  ;*          Branch Occurs to L5
49                  ldi         r1,r0                    Block 2
50                  bgt         L5
51  ;*          Branch Occurs to L5
52
53  L3:
54                  ldiu        1,r0
55                  ldiu        1,r1
56                  addi        *+fp(1),r0
57                  subri       *+fp(2),r1
58                  sti         r0,*+fp(1)
59                  cmpi[1]     0,r0
60                  sti         r1,*+fp(2)
61                  bgt         L5
62  ;*          Branch Occurs to L5
63                  ldi         r1,r0                    Block 4
64                  ble         L3
65  ;*          Branch Occurs to L3
66                                                       Block 5
67  L5:
68                  ldiu        *-fp(1),r1
69                  bud         r1
70                  nop
71                  ldiu        *fp,fp
72                  subi        4,sp
```

Figure 11: Despite the full coverage of the source code, code coverage analysis of the object code reveals several unexercised paths

New tests are needed in order to achieve full coverage.

Devising additional tests

New test 1. Line 48. End of block 1. Branch to L5

This assembler code represents an initial check to see whether there will be any processing in the while loop. If not, the remaining code is bypassed from that point. The following test case meets neither of the while loop's entry criteria, and hence exercises that branch.

```
fwhile10(10,-10)
```

The coverage is improved accordingly.

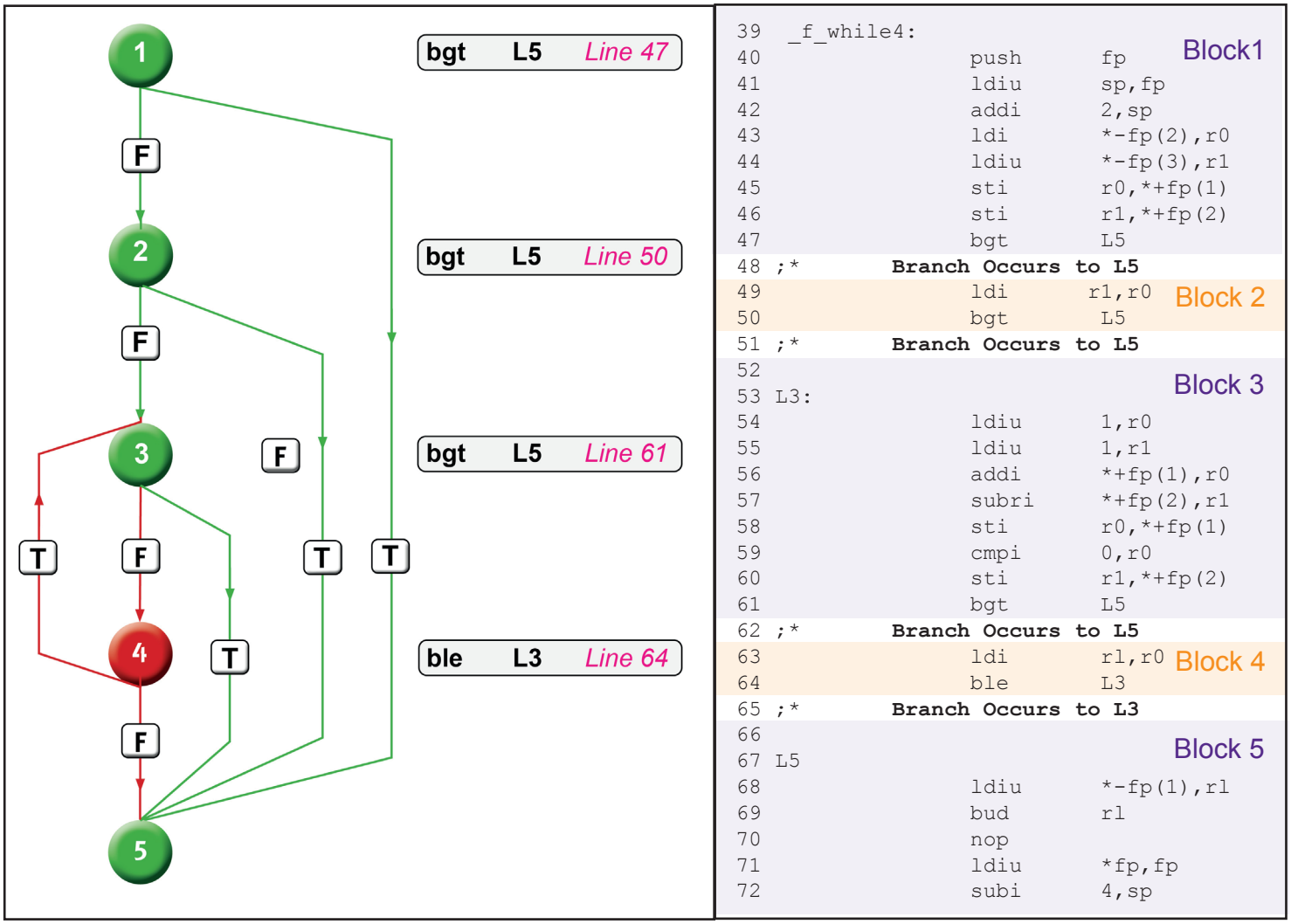


Figure 12: A test case meeting neither of the while loop's entry criteria exercises code that bypasses that loop

New test 2. Line 61. End of block 3. Branch to L5

The bgt (that is, branch if greater than) test has only ever been true. Referring back to the C code, this results from the fact that the loop has never exercised more than once for any of the test cases thus far. This new test case improves the coverage once again.

```
fwhile10(-5,-5)
```

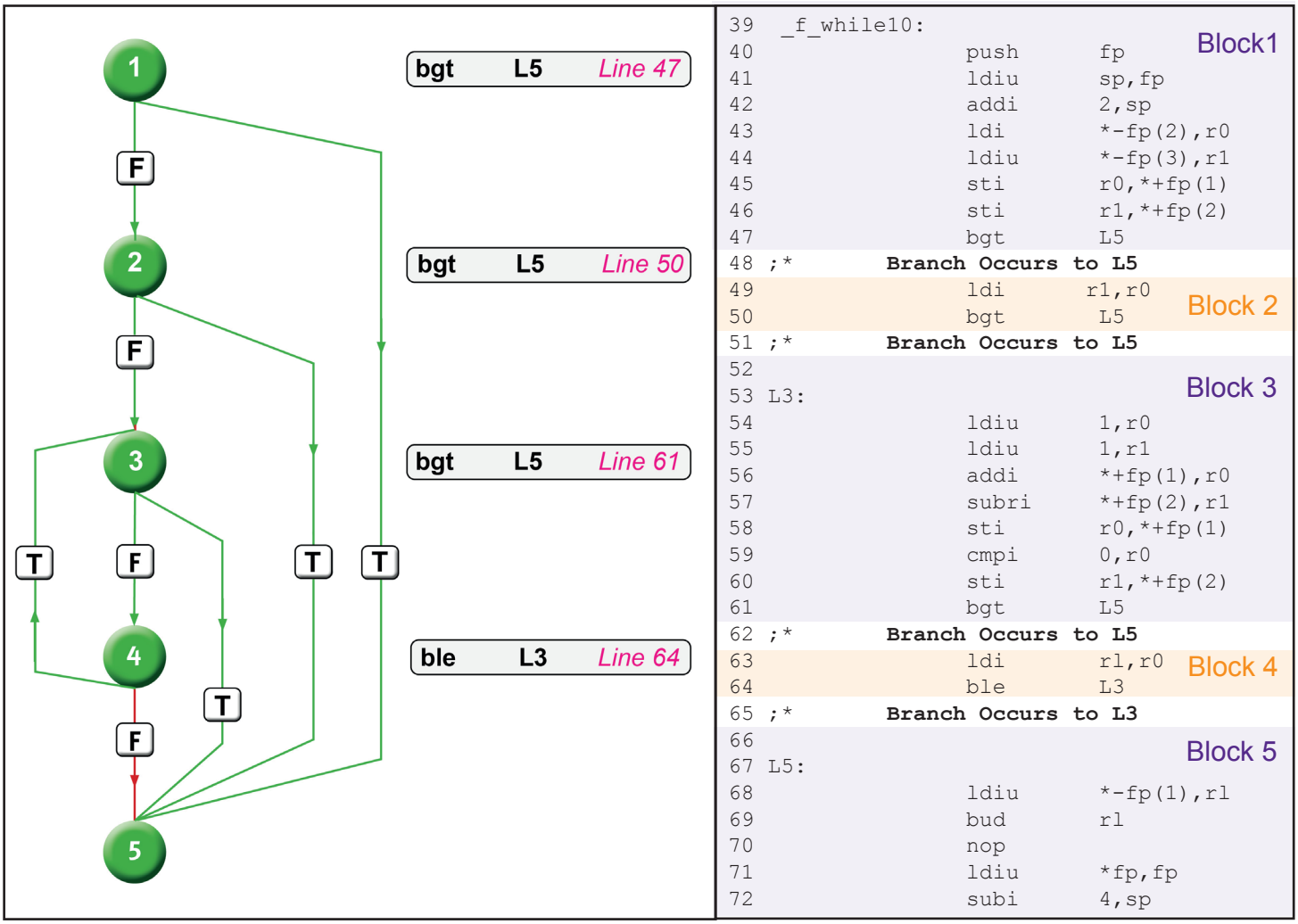


Figure 13: A test case that exercises the while loop more than once improves coverage again

New test 3. Line 64. End of block 4. Branch to L3

Inspection of the code shows that the result of the ble test is only false when

```
!( f_while10_local2 > 0 )
```

evaluates to false, causing the loop execution to terminate due to the second condition in the *while* statement.

Due to the nature of the code and the fact that `f_while10_local2` is decremented, this is a little more tricky to accommodate than previous examples. It implies that the while loop can only be terminated as a result of the second test if the integer rolls over[2]. In this example, the integer is 32 bit and is signed, so roll over will occur at

-1*(2^32) = -2147483648.

So an appropriate test is

```
f_while10(-20, - 2147483644)
```

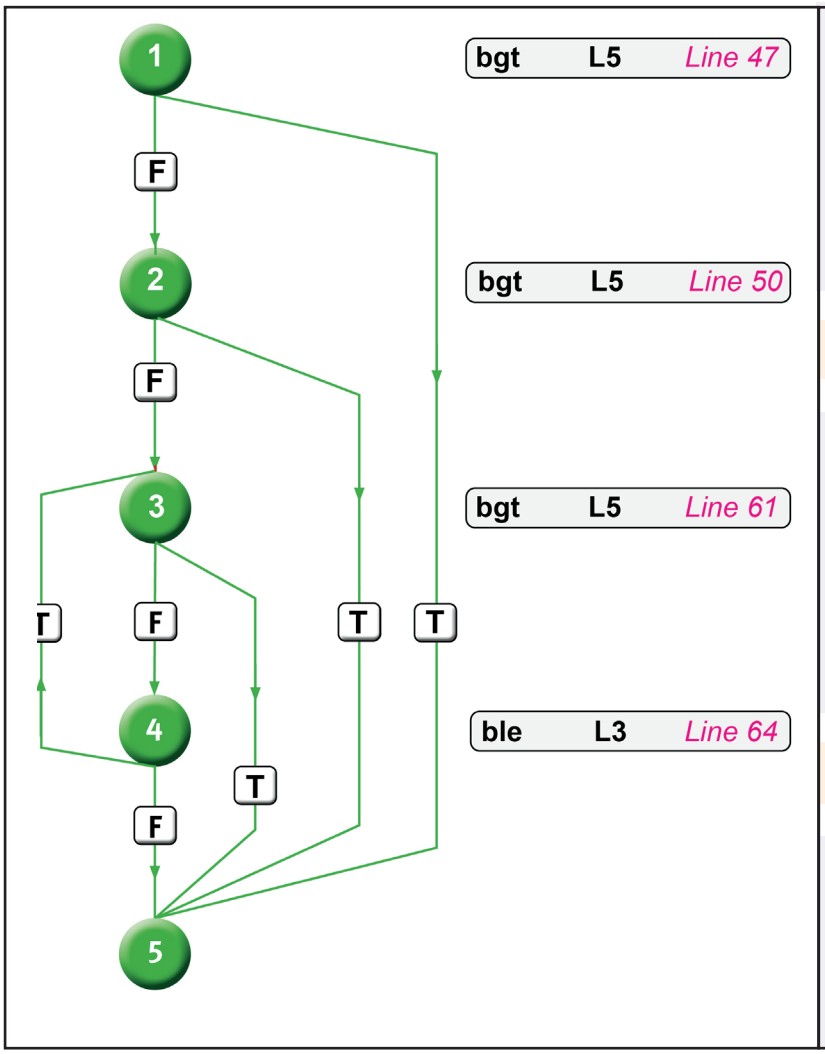
because it will ensure that the loop terminates when `f_while10_local2` "rolls over" and becomes positive. With this test in place, 100% statement and branch coverage of the assembler code is achieved.

A "main" function to exercise all object code in the example would look like this:

```
int main()
{
  int a = 0;
  int b = 1;
  int c = 0;
  a = b || c;
  f_while10(0,0); //Contributes to initial 100% source code coverage
  f_while10(0,1); //Contributes to initial 100% source code coverage
  f_while10(10, -10);
  f_while10(-5,-5);
  f_while10(-20, -2147483644);
  return(0);
}
```

It is possible that the number of calls to `f_while` could be reduced if replacements were to be devised to exercise more than one of the unexercised paths at a time.

[2] This test case is relying on undefined behaviour (integer overflow) to obtain coverage (which would not be possible for a processor that caught this by throwing an exception). Arguably the code is defective, and OCV testing has detected a potential flaw in the algorithm




```
39  _f_while10:                                          Block1
40                  push        fp
41                  ldiu        sp,fp
42                  addi        2,sp
43                  ldi         *-fp(2),r0
44                  ldiu        *-fp(3),r1
45                  sti         r0,*+fp(1)
46                  sti         r1,*+fp(2)
47                  bgt         L5
48  ;*      Branch Occurs to L5
49                  ldi         r1,r0                    Block 2
50                  bgt         L5
51  ;*      Branch Occurs to L5
52
53  L3:                                                  Block 3
54                  ldiu        1,r0
55                  ldiu        1,r1
56                  addi        *+fp(1),r0
57                  subri       *+fp(2),r1
58                  sti         r0,*+fp(1)
59                  cmpi        0,r0
60                  sti         r1,*+fp(2)
61                  bgt         L5
62  ;*      Branch Occurs to L5
63                  ldi         r1,r0                    Block 4
64                  ble         L3
65  ;*      Branch Occurs to L3
66
67  L5                                                   Block 5
68                  ldiu        *-fp(1),r1
69                  bud         r1
70                  nop
71                  ldiu        *fp,fp
72                  subi        4,sp
```

Figure 14: Ensuring integer roll-over completes coverage

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