LDRA Software Technology

Technical White Paper

オブジェクトコード検証（OCV）：なぜ重要なのか

www.ldra.com

© LDRA Ltd. This document is property of LDRA Ltd. Its contents cannot be reproduced, disclosed or utilized without company approval.

目次

はじめに 3
例題について 3
オブジェクトコード検証：ツール認定例 1 4
追加テストの考案 6
新しいテスト 1.　62 行目、Block 3 の最後、L3 への分岐 6
新しいテスト 2.　47 行目、Block 1 の最後、L3 への分岐 7
新しいテスト 3.　51 行目、Block 2 の最後、L5 への分岐 8
DO-178C とオブジェクトコード検証 10
機能安全の世界におけるコンパイラ 11
認証済のコンパイラ 12
使用するコンパイラを信頼すること 12
ソースコード単体テストに対する「as-if」ルールの影響 14
問題への対応 15
ツール認定パック OCV 16
アプリケーションコードでの単体テスト OCV 16
完全なアプリケーションコード上の OCV 17
結論 17
引用文献 18
付録：ツール認定の例 2 19
追加テストの考案 21
新しいテスト 1.　47 行目、Block 1 最後の L5 への分岐 21
新しいテスト 2.　61 行目、Block 3 最後の L5 への分岐 22
新しいテスト 3.　64 行目、Block 4 最後の L3 への分岐 23

はじめに

機能安全、セキュリティ、コーディング標準の規格（IEC 61508 [1]、ISO 26262 [2]、IEC 62304 [3]、MISRA C [4]、MISRA C++ [5]、CWE [6] など）がお墨付きを与える検証や妥当性確認のプロセスにおいては、テスト対象のアプリケーションがどの程度実行されたかを示すことが非常に重要であるとされます。経験上、コードが正しく実行されることが示されていれば、現場で不具合に遭遇する確率はかなり低いといえます。しかし、C/C++他、どれであれ、ほとんど例外なく高水準言語でのソースコードに焦点が当てられたものです。このようなアプローチでは、コンパイラが開発者の意図を忠実に再現するオブジェクトコードを作成していることが前提になっています。

オブジェクトコードにおける制御フローとデータフローが、それの生成元であるソースコードの正確な鏡像になっていないことは必然ですので、ソースコードですべてのパスが確実に実行できることを証明しても、オブジェクトコードでも同じだと証明することにはなりません。さらに悪いことに、ソースコードの単体テストは、誰もが価値を認めるものですが、それも誤解を与える可能性があります。なぜなら、単体テストハーネスにラップされた関数をコンパイルして得られるオブジェクトコードは、完全なシステムのコンテキストで生成されたものとは著しく異なっているかもしれないからです。

航空宇宙産業で使用されている DO-178C [7] だけが、開発者の意図と実行コードの動作との間に危険な不整合が生じる可能性に着目している規格です。それでも、そのような不整合が検出されないままになる可能性が明らかな代替策を提唱する人は多いものです。どのように弁解されようとも、そのようなアプローチによって、クリティカルアプリケーションすべてにおいて、ソースコードとオブジェクトコードの違いが壊滅的な結果をもたらし得るという事実に変わりはありません。本稿では、オブジェクトコード検証（OCV：object code verification）が、障害によって悲惨な結果を招くシステム、そして実際、ベストプラクティスだけしか十分とはならないシステムすべてにとってベストプラクティスとなるプロセスに対して大きく貢献できる理由を説明します。

例題について

ここでの例題はすべて、高水準言語として C を使用していますが、強調している問題は多かれ少なかれ、言語に関係なく当てはまります。たとえば、C++ 全般の性質、特に C++14 や C++17 など後のバージョンで導入された拡張機能は、実行ファイル（オブジェクトコード）内のより多くの要素がソースまで追跡できないことを意味するものです。

例題は非常にシンプルなものです。もちろん、実プロジェクトでは考慮すべきコードの規模はもっと大きく、多くは残りのコードベースと同レベルで厳密な解析を必要とするライブラリなどサードパーティのコードを含んでいます。このような場合のベストプラクティスは、認定されたライブラリにデプロイするか、サードパーティのコードを内部開発のコードと同じ妥当性確認および検証作業の対象にすることです。

オブジェクトコード検証：ツール認定例 1

最初に、コンパイラが生成したオブジェクトコードの制御フロー構造が、元となるアプリケーションのソースコードとどのように異なるか、なぜ異なるかを理解するために、非常に単純な C ソースコード（ LDRA tool qualification pack [8] からの引用 ）を見てみます。付録で LDRA tool qualification pack からの 2 つ目の例を示します。

```
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--;
    }
}
```

次の関数呼び出し 1 回で、この C コードの 100% のソースコードカバレッジを達成できることがわかります。

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

コードは、1 行 1 操作にリフォーマットして、フローグラフ上で、直線状のコード系列である「基本ブロック」をノードとして表現され(図 1)、基本ブロック間の関係はノード間の有向エッジで表現されます。

ソースコードをコンパイルすると、結果として得られるアセンブラコードのフローグラフは、ソースコードのフローグラフとはまったく異なるものになりがちです。なぜなら、C や C++コンパイラが従うルールでは、バイナリが「*同じものであるかのように（as if）*」動作するのであれば、好きなようにコードを変更することが許されているからです。

C++規格 §1.9/1 [9]から

「*この規定は、as-if ルールと呼ばれることもある。なぜなら、実装では、観察可能なプログラムの振る舞いから判断できる範囲において、結果があたかもその要件に従っているかのようである限り、この文書のいかなる要件も無視してかまわないからである。たとえば、その値が使用されずプログラムの観察可能な動作に影響を与える副作用が発生しないと演繹できる場合、式の一部分を評価する必要はない*」

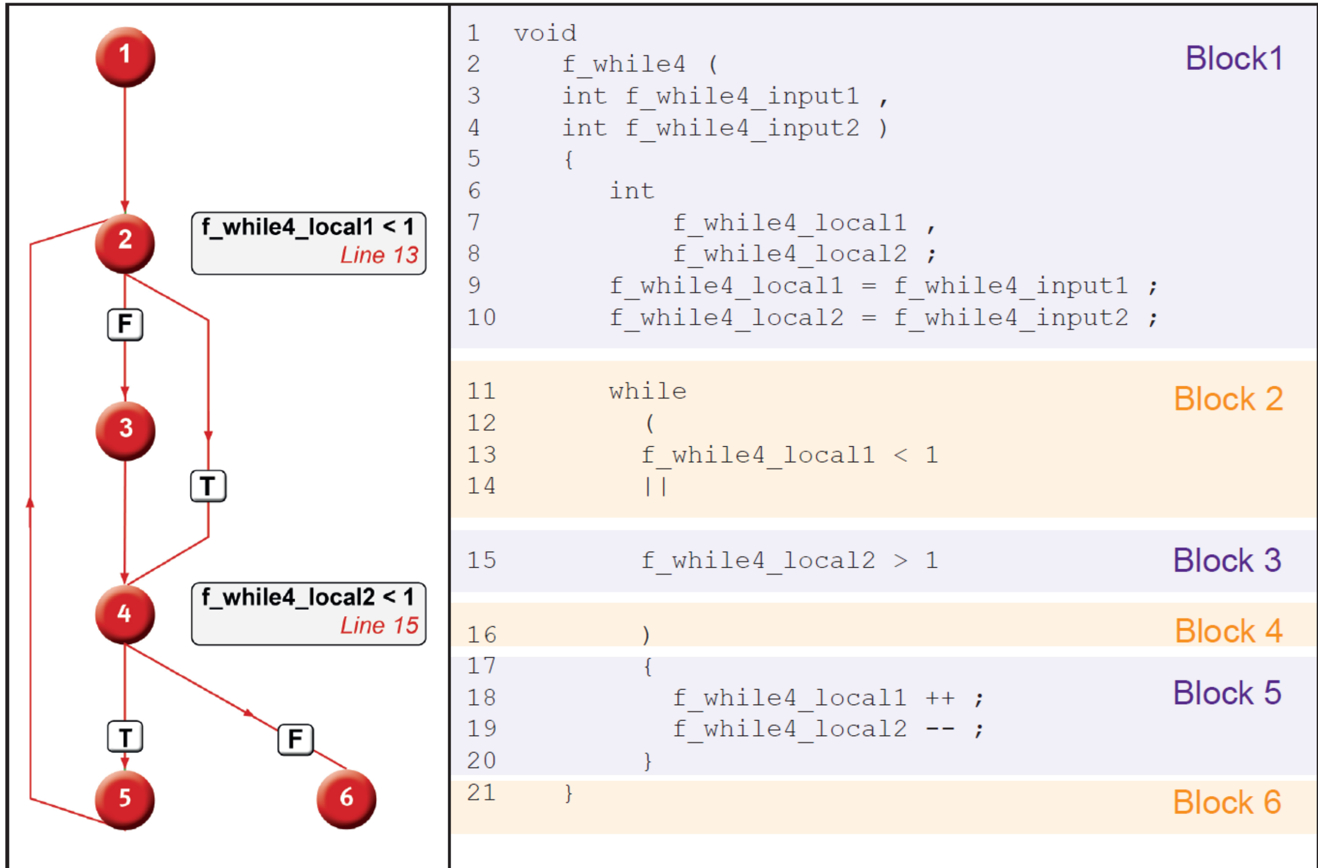

図 1：1 行 1 操作にリフォーマットし、フローグラフ上で
直線状のコード系列である「基本ブロック」をノードとして表現した元のソースコード

このコードを広く使用されている市販コンパイラでコンパイルすると、どれもたいていの場合、フローグラフは上記のソースコードに対するものとは異なるようになります (図 2)。図はこの例題で、TI 社コンパイラを最適化なしで使用した場合の結果を示すものです。フローグラフの赤色の要素は、以下の呼び出しによっては実行されていないコードを表しています。

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

オブジェクトコードとアセンブラコードは 1 対 1 に対応するので、アセンブラコードを利用してオブジェクトコードでどの部分が未実行であるかを明らかにでき、テスト担当者が追加テストを考案して完全なアセンブラコードカバレッジを達成することで、オブジェクトコードカバレッジを達成させます。

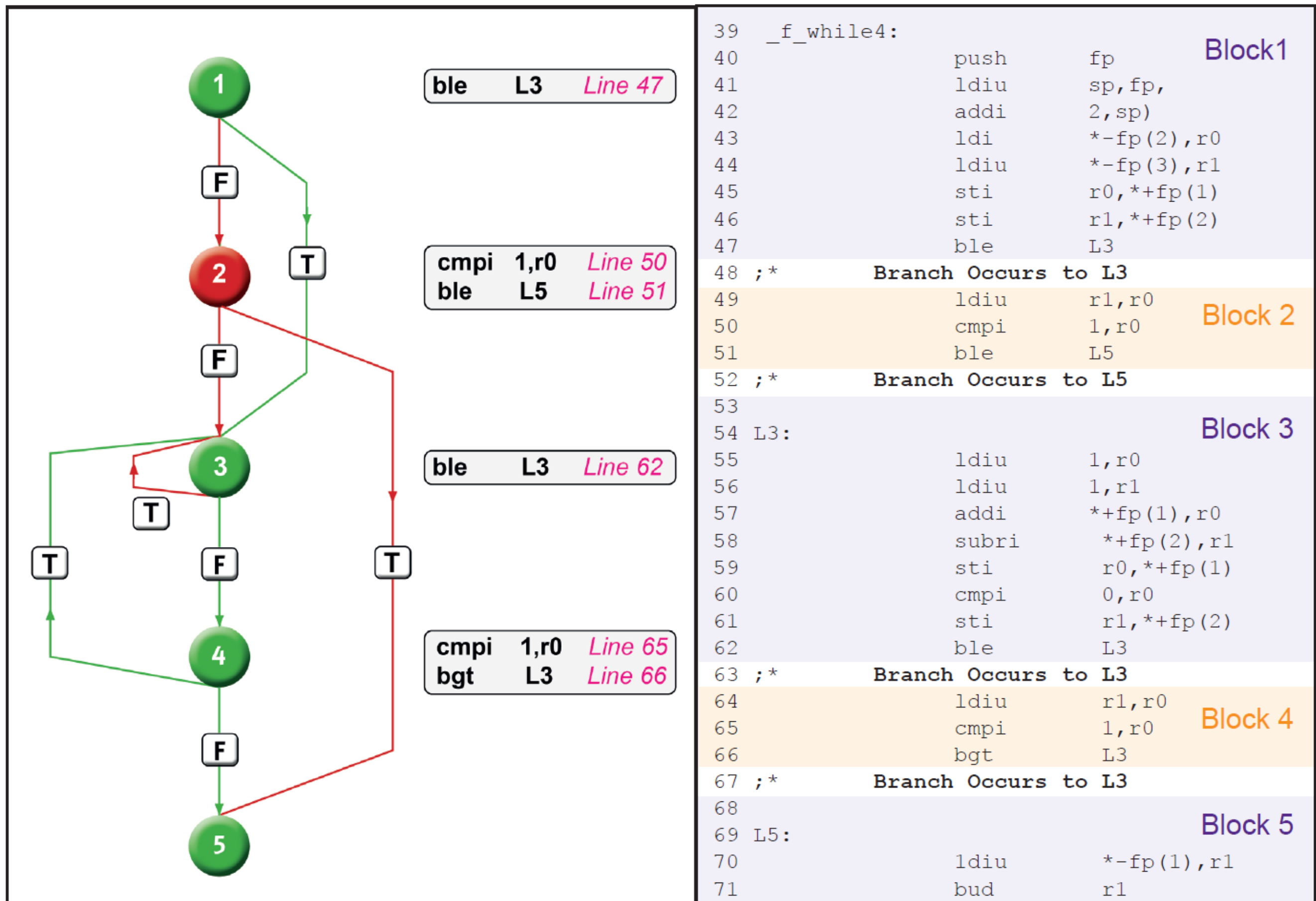

図 2：ソースコードでは 100%のカバレッジであるが、オブジェクトコードでのコードカバレッジ解析により、いくつか未実行のパスがあることが判明する

追加テストの考案

新しいテスト 1.　62 行目、Block 3 の最後、L3 への分岐

現時点のテストデータでは Block 3 のループは 1 回しか繰り返されないため、62 行目の `ble` 条件分岐命令の条件は false と評価され、繰り返しを続行するかどうかを確認するテストの 2 つの可能な結果の 1 つだけが発生しています。ソースコードで以下の新しいテストケースを追加して、ループが 2 回目の繰り返しを実行するようにし、この `ble` 条件分岐命令で true と false の両方のケースが実行されるようにします。

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

図 3 は、上記 2 つのテストケースでの累積カバレッジを示しています。

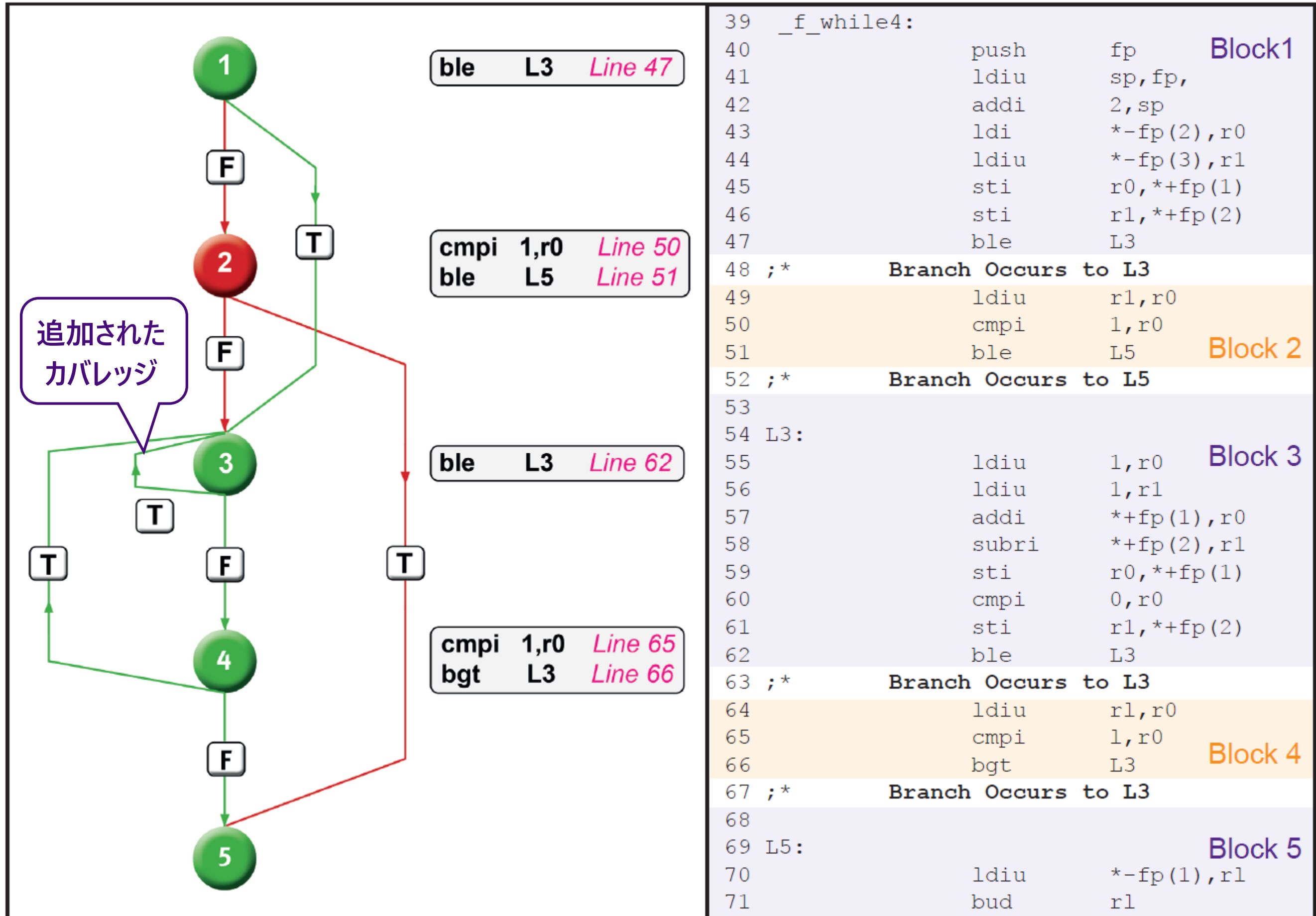

図3：テストケースf_while4(-1, 3) を追加適用した場合に得られる
追加のオブジェクトコードカバレッジの結果

新しいテスト 2.　47 行目、Block 1 の最後、L3 への分岐

ソースコードには、`while` ループの制御式に `||`（or 条件）が含まれています。現時点のテストケース２つはどちらも、この `||` の左辺コード `f_while4_local1 < 1` が true を返し、制御式全体が true と評価されます。（短絡評価により、右辺コード `f_while4_local2 > 1` は実行されません）

右辺コードが実行されるよう、左辺コードが false を返す以下の新しいテストケースを追加することでこれに対処します。

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

図 4 で、新しいテストケースによって累積のカバレッジが向上する様子が示されます。

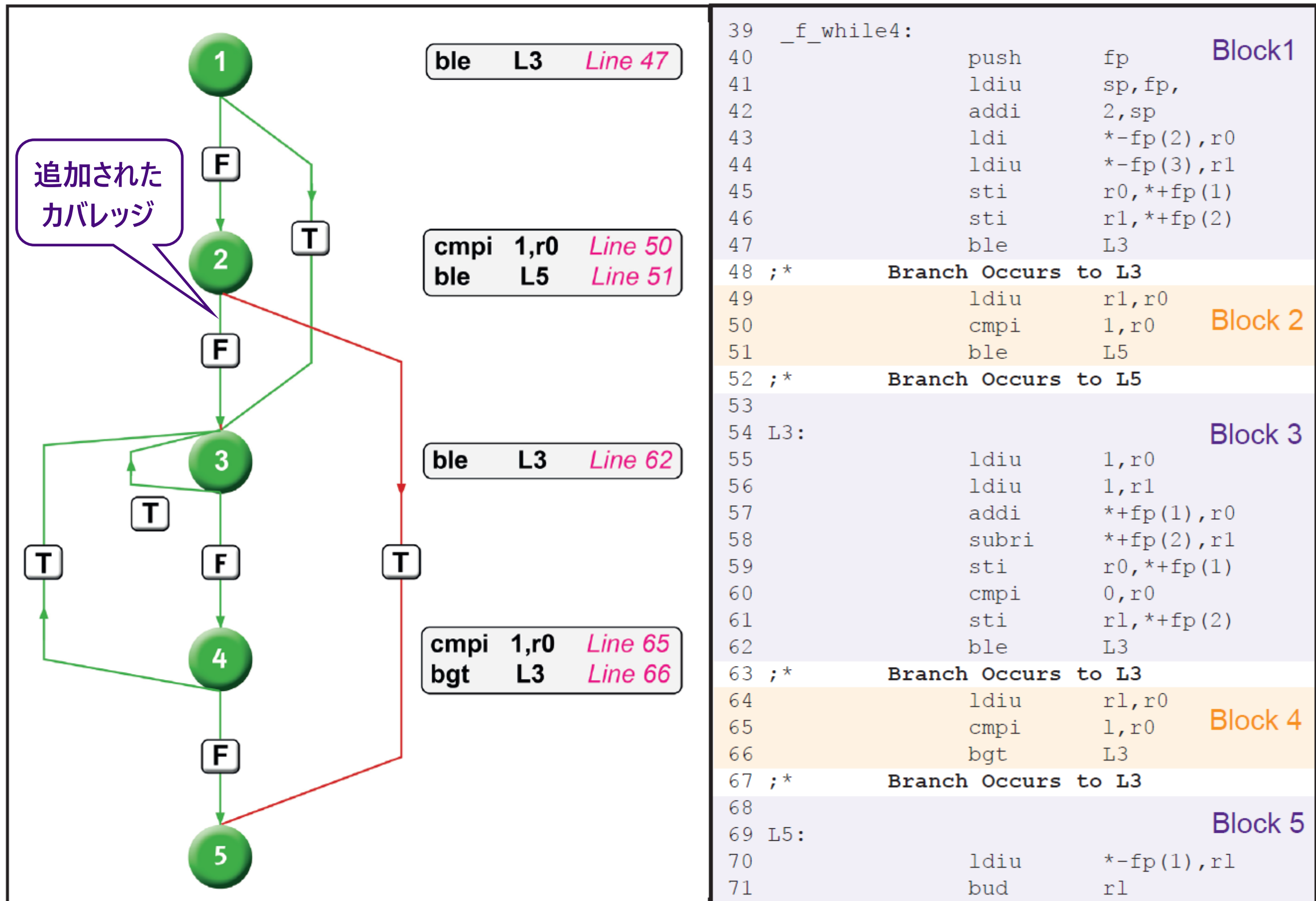

図 4：追加のテストケース `f_while4(3, 3)` の適用でオブジェクトコードカバレッジが増加する

新しいテスト 3.　51 行目、Block 2 の最後、L5 への分岐

未実行で残っている分岐は、ソースコードの `while` 文の初期条件で `||` の両辺いずれもが false である場合に、51 行目の `ble` 命令で `L5` に分岐することで、このループ本体内のオブジェクトコード全体がバイパスされるものです。

そのようになる状況を提供するために、次のテストを追加します（この例で最後のテストです）。

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

これにより、図 5 に示すように、ステートメントとブランチのカバレッジが 100% になります。

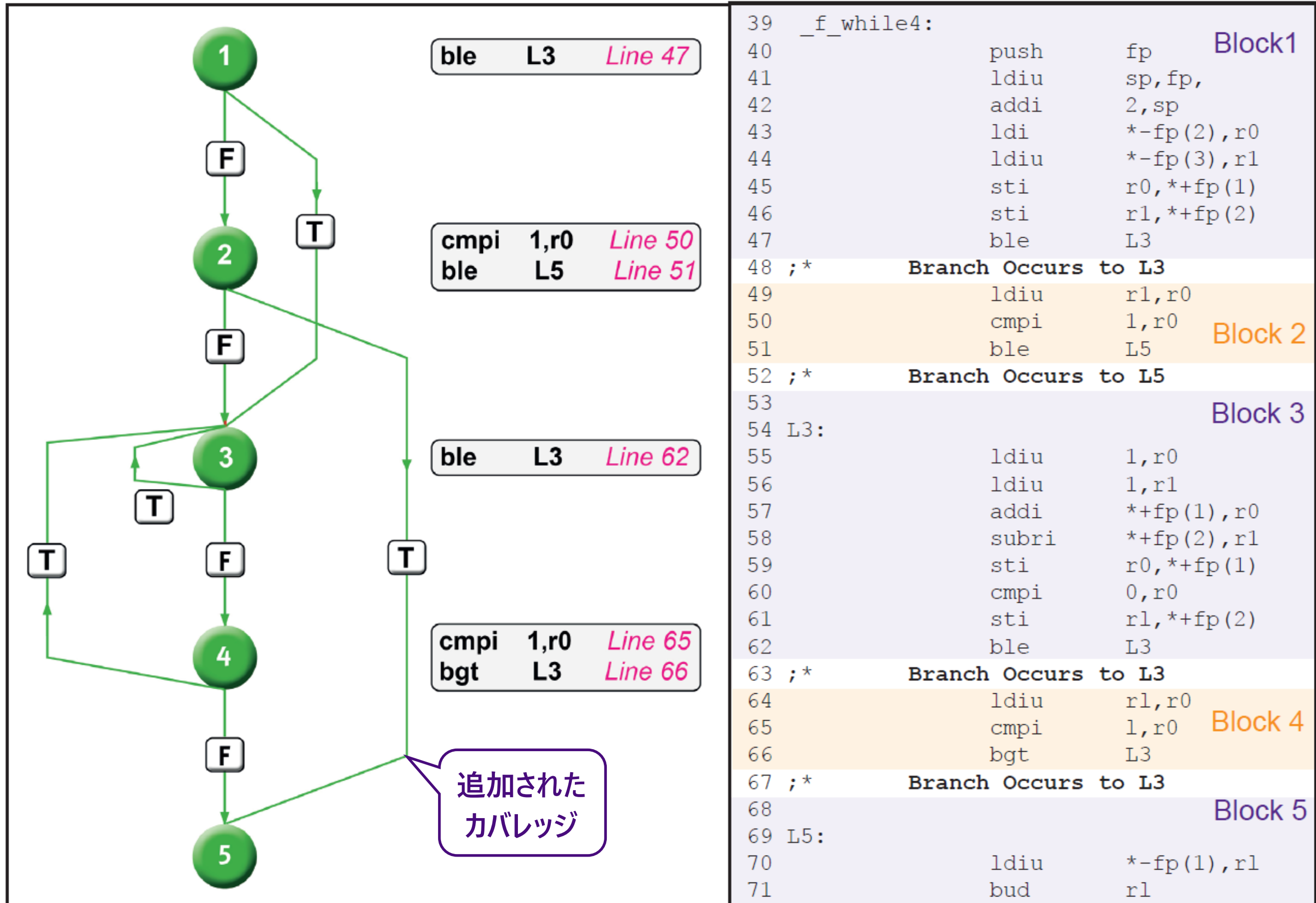

図 5：テストケース f_while4(3, 0) によって、100%のオブジェクトコードカバレッジとなる

この例のすべてのオブジェクトコードを実行する「main」関数は、次のようになります。

```
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とオブジェクトコード検証

上記の前提を踏まえて、DO-178C と航空宇宙領域で要求されるもの、特に「構造カバレッジ解析」と題された 6.4.4.2 項に戻ります。DO-178C が構造カバレッジを要求する理由は、以下のように、構造カバレッジが、要件ベーステストでアプリケーションを完全に実行したことを保証する最良の方法だからです。

「*この解析の目的は、要件ベースのテスト手順でどのコード構造が実行されなかったかを判断することである。要件ベースのテストケースではコード構造が完全には実行されていない可能性があるため、構造カバレッジ解析を実施して、構造カバレッジを得るための追加の検証を生成する*」

パラグラフ 6.4.4.2b では、次のようにアプリケーションがレベル A である場合の追加の要件について説明しています。

「*構造カバレッジ解析は、ソフトウェアがレベル A ではないか、コンパイラがソースコードの文に直接トレースできないオブジェクトコードは生成しないという場合には、ソースコードに対して実施してもよい。その場合、生成されたオブジェクトコード系列の正しさを証明するために、オブジェクトコードに対して追加の検証を行う必要がある。コンパイラが生成したオブジェクトコード内の配列限界チェック用コードは、ソースコードには直接トレースできないオブジェクトコードの例である*」

つまり、アプリケーションのレベルが A であって、コンパイラがソースコードにトレースできないオブジェクトコードを生成している場合には、オブジェクトコードの追加検証を行う必要があるのです。

構造コード解析の目的が「*コード構造を完全に実行する*」ことであるなら、オブジェクトコードのパスが実行されないままであれば、完全に実行することは実現されていないという議論があるかもしれません。

しかし一方で、DO-178C では、この作業を自動化する義務もオブジェクトコードのカバレッジを達成する義務もありません。実際、この規格の他の部分に関しても、そして本当に、機能安全規格が管理するほとんどすべての開発作業に関しても、セクターを問わず、何かを自動化する義務はないのです。この点を強調するために、2002年の FAA のポジションペーパー CAST-12「Guidelines for Approving Source Code to Object Code Traceability」[10]（現在 FAA のサイトにリンクなし）では、手作業によるレビューをどのように行うかだけでなく、そのような手作業による解析に費やす時間を最適にするために、典型的な構造を使用してサンプルアプリケーションを考案する方法についても概説されています。

しかし、セクターに関係なく、このようなアプローチ単独で本当にベストプラクティスなのでしょうか? そして、すべてのソフトウェアアプリケーションの中で最もクリティカルなものに対して、それを正当化できるのでしょうか?

機能安全の世界におけるコンパイラ

これらの疑問に答えるために、機能安全なソフトウェア全般に課された義務について考えてみます。すべてのセクターにおいて、規格は、防御的なコーディングを適用すること、コードがテスト可能であること、適切なコードカバレッジを照合できること、そして、アプリケーションコードが要件にトレース可能でシステムがその要件を完全かつ一意に実装していることを保証すること、を要求しています。

機能安全なコードには、さまざまな原因で起こりえる予期せぬ事象から守るための防御的コードを含める必要があります。たとえば、コーディングエラーや宇宙線によるメモリの破損は、コードのロジックからは「不可能な」コードパスが実行されることにつながる可能性があります。

高水準言語、特に C/C++ にはコードが準拠する言語仕様で動作が規定されていない機能が驚くほど多くあります。定義されていない動作が、取り除くことができず潜在的に悲惨な結果につながることは明らかであり、これは機能安全なアプリケーションにおいて防御されなければなりません。

また、コードには高いレベルのコードカバレッジを実現することも要求され、一部の (特に自動車) 分野では、高度な外部診断や、キャリブレーション、そして開発用のツールに対応した設計が要求されることが非常に一般的です。

ここで問題となるのは、防御的コーディングや外部データアクセスなどのプラクティスは、コンパイラが認識する世界に属するものではないということです。たとえば、C も C++ もメモリ破壊を考慮していません。そのため、メモリ破壊から保護するように設計されたコードは、メモリ破壊がないときにアクセスできなければ、最適化で無視されてしまう可能性があります。したがって、防御的コードは、それが「最適化で削除」されないために、構文的にも意味的にも到達可能でなければなりません。

未定義の動作も予期せぬ事態の原因になり得ます。単に避けるべきだと言うのは簡単ですが、多くの場合、それらを特定することが非常に困難です。未定義の動作が存在する場合、コンパイルされた実行コードの動作が開発者の意図と一致する保証はありません。また、デバッグツールで使用されるデータへの「バックドア」アクセスも言語が考慮していない別の状況を表しているため、予期しない結果をもたらす可能性があります

これらすべての領域はコンパイラベンダーの検討事項には含まれていないため、コンパイラの最適化が大きな影響を与える可能性があります。一見健全な防御的コードが「実行不可能性」に結びつく場合、つまり、どのような入力値の集合によってもテスト・検証できないパス上に存在する場合に、それが最適化によって削除されてしまう可能性があります。さらに憂慮すべきは、単体テスト中に存在すると示された防御的コードが、システム実行ファイルが構築されるときに削除される可能性があることです。つまり、単体テスト中に防御的コードのカバレッジが達成されたからといって、完成したシステムにその防御的コードが存在することは保証されないのです。

認証済のコンパイラ

ソフトウェア開発には、設計ツールやコード生成ツール、コンパイラ/リンカ、ライブラリ、テストツール、構造カバレッジツールなど多くのツールが必要です。DO-178 ツール認定は、コンパイラを含む開発およびテストツールに関係します。

コンパイラベンダーの中には、この認定プロセスをサポートする認証済コンパイラを提供するところもあり、それを用いればより効率的です。しかし、コンパイラ自体は使用される高水準言語が定める規則に従うものであり、要するに、ここで概説した課題はどれも、認証済コンパイラを使用するだけでは解決できません。

使用するコンパイラを信頼すること

コンパイラベンダーは立派で、技術的要求の非常に厳しいプロフェッショナルな仕事をしています。他のソフトウェアと同様バグは当然ありますが、通常、コンパイラの実装はその設計要件を満たそうとするものです。しかし、これまで述べてきたように、それらの要件が必ずしも機能安全なシステムのニーズを反映しているわけではありません。

コンパイラは通常、作成者の目標に対して機能的に正しいものであると考えることができます。しかし、それが完全に望まれていることや期待されていることであるとは限らないのは、CLANG コンパイラによるコンパイル結果である別の例（図 6）を見るとよくわかります。

```
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;
}
```

定数値（13、23）はアセンブラには現れません
最適化で、0 と 1 になります

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

図 6：「as if」義務の意味を理解すること

ここで、防御的なコードである error 関数への呼び出しがアセンブラコード内にないことは明らかです。

state オブジェクトは、初期化時および、S0 と S1 の case 内のみで変更されるため、コンパイラは state に与えられる値が S0 と S1 だけであると推論できます。コンパイラは、メモリ破壊がないものと仮定して（実際、コンパイラはまさにそうします）、state が決してそれら以外の値を保持することがないため、default 節は必要ないと結論付け、削除しているのです。

また、コンパイラは、実際の state オブジェクトの値（13 と 23）は数値としては使用されないため、単に 0 と 1 の値を使用して状態を切り替え、排他的 or（xor）命令 を使用して state の値を更新します。このバイナリは「as-if」の義務に準拠し、コードは高速でコンパクトです。その責任の範囲内で、コンパイラは良い仕事をしています。

この動作は、リンカによるメモリマップのファイルを使用してオブジェクトに間接的にアクセスする「キャリブレーション」ツールや、デバッガを介した直接のメモリアクセスなどに影響を与えます。繰り返しになりますが、このような事項はコンパイラの検討範囲には含まれないため、最適化やコード生成時には考慮されません。

ここで、コードに変更はありませんが、コンパイラに提示されるコードのコンテキストが少し変更されているとします（図 7）。

```
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;
}

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

定数値(13、23)は、ここでも update() に対するアセンブラには現れません
しかし、f() 内で int に変換されると出現します

```
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
```

図 7：state 変数の値を返す関数を追加したことによる影響

state 変数の値を整数で返す関数が追加されました。そうすると、コンパイラに入力されるコードでは、数値として 13 と 23 が重要です。しかし、それらの値は update 関数の中では操作されず（update 関数は変更されていません）、新しい関数 f の中だけで出現するものです。

要するに、コンパイラは 13 と 23 の値をどこで使用すべきかについての価値判断を（正しく）行い続けます。そして、それらは可能性のある状況すべてに適用されるわけではありません。

この新しい関数 f で、state 変数へのポインタを返すように変更すると、アセンブラのコードが大きく変わります (図 8)。ポインタを介したエイリアスアクセスの可能性があるため、コンパイラは state オブジェクトで何が起こっているかを推測できません。そのため、13 と 23 という値は重要でないと結論付けることができず、アセンブラ内で明示的に表現されるようになりました。

```
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
```

図 8：state 変数へのポインタを返すと、結果のアセンブラコードが大きく変わる

ソースコード単体テストに対する「as-if」ルールの影響

ここで、架空の単体テストハーネスの例 (図 9) を考えます。このハーネスでは、state 変数の値を意図的に操作して switch 文の default 節が「最適化で削除」されないようにし、この default 節のカバレッジが取得できるようにします。このようなアプローチは、ソースコードの残りの部分に関連するコンテキストを持たず、すべてをアクセス可能にすることが要求されるテストツールではまったく正当なものです。

コンパイラは、ポインタを介して任意の値が state 変数に書き込まれることを認識し、ここでも 13 と 23 の値が重要でないと結論付けることはできません。結果、その値がアセンブラ内で明示的に表現されるようになりました。この場合、S0 と S1 だけが state 変数のとり得る値であると結論付けることはできず、それはこの default 節のパスが実行可能であることを意味します。state 変数の操作は目的を達成し、アセンブラコード内で error 関数の呼び出しが出現します。

```
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;
}

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

これですべてが変わります！

```
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
```

```
testDefault(): # @testDefault()
  mov dword ptr [rip + state], -1
  jmp update() # TAILCALL
```

図 9：ソースコード単体テストにおける「as if」ルールの影響[1]

しかし、この操作は製品として出荷されるコードには存在しないため、実際には error()の呼び出しは完全なシステムには存在しません。

これは明らかに重大な問題を提起しています。もしコンパイラが、完成したアプリケーションとテストハーネスとで異なるコードを扱うとしたら、ソースコードの単体テストカバレッジに（あるいはオブジェクトコードの単体テストカバレッジにさえ）本当に価値があるのでしょうか？

答えは条件付きの「yes」でなければなりません。多くのシステムは、このようなアーティファクトによる証拠に基づいて認証されており、サービスにおいて安全で信頼できると証明されています。しかし、もし、開発プロセスが最も詳細な精査に耐え、ベストプラクティスに準拠するのであれば、あらゆるセクターのクリティカルシステムにおいて、ソースレベルの単体テストカバレッジをアプリケーション全体の OCV によって補完しなければならないという事実は変わりません。

問題への対応

DO-178C の要件に注目すると、DO-178B に対して追加して明確化したにもかかわらず、この点に関して規格は贔屓目に見ても曖昧なままです。開発チームには、行っている作業と、その作業によってエラーが発生した場合のリスクを考慮した上で、正当化できる方法を選ぶ責任があります。

[1] この例の「単体テスト」コードには、'S' へのポインタを無関連な型（int*）へのポインタに変換するという未定義の動作が含まれています。

以下で説明する 3 つの手法において、オブジェクトレベルで達成された異なるカバレッジ結果の例は、コード構造が違っていることを示すものであるため、その違いを確認して文書化する機会を提供します。強調されるべきことは、オブジェクトコードに追加された（ソースにはない）分岐構造が実行不可能であることは非常に多く、したがってオブジェクトコードレベルで100%のカバレッジを達成することは不可能だということです。それはコード実行の目的ではないため、できなかったからといって、規格の DAL-A 要件を満たさないことを意味するものではありません。

ツール認定パック OCV

ここでは 2 つの例を使用しています。最初の例、関数 `f_while4` は、LDRA のツール認定サポートパックから引用したものです。このサポートパックは、セーフティクリティカルなシステムで使用される可能性のあるすべてを網羅する要素から構成されており、構成要素それぞれが個別に証明できるものです。これによって、コンパイラがすべての構成要素を正しく解釈できることが証明されていれば、それら構成要素が完全なアプリケーションの部分として使用される場合は信頼性が確保されているという考え方を裏付けることができます。LDRA ツールスイートは、このようなアプローチをサポートする機能を備えています（図 10）。

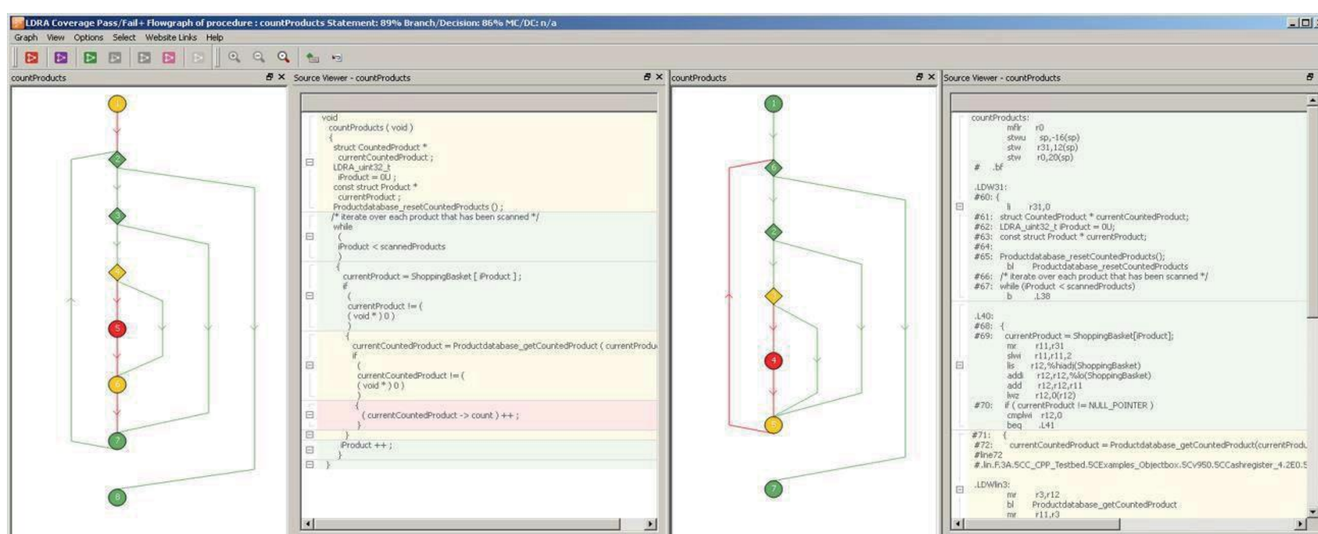

図 10：LDRA ツールスイートを使用したソースコードとオブジェクトコードのカバレッジの比較

アプリケーションコードでの単体テスト OCV

より一般的な妥協案は、ソースコード解析を完了するために利用したものと同じ単体テストに OCV を適用してカバレッジ結果を比較し、オブジェクトコードとソースコードの間で異なっている結果があればそれを調査することです。

このようなアプローチだけでもツール認定パックの単体テストだけに頼るよりも優れた解決策であることは明らかです。なんといっても、このシナリオでは、ツール認定パックの例とは違ってテスト対象のコードはアプリケーションの一部であるので、`update` の例で強調されている条件にもかかわらず、それがターゲットで実行されるアプリケーションをより代表することは間違いありません。

オブジェクトレベルでのアプリケーションコードのカバレッジは、DO-178C 規格の要件ではありませんが、この規格の要求事項である高レベルコードとオブジェクトレベルコードの構造の違いをすばやく簡単に特定する方法を提供します。

完全なアプリケーションコード上のOCV

この立場に立つにしても、本稿の 2 番目の例（update）は、最も正確で包括的なテストが完全なアプリケーションの OCV を実施することによってのみ保証できることを明らかにしています。LDRA ツールスイートが提供する機能が、それを可能にします。

完全なアプリケーションコード上の OCV は必要でしょうか? もちろん、DO-178C 他どの規格から見ても必要ではありません。ベストプラクティスでしょうか? その立場に異議を唱えるのは難しいです。商業的に正当化されますか? それは開発チームが判断することです。

結論

OCV を適用するためのこれらの異なるアプローチ (およびそれらを容易にするツール) は、ソースコードとオブジェクトコードの間の不整合を検査し、正当化するための全体的な戦略の一部として考えることが最善です。システムのさまざまな部分に対して、ここで述べた手法 (おそらく組み合わせて) だけでなく、手作業によるコード検査や、実行パスをトレースするためのデバッグツールの使用など、他の手法も含めさまざまなアプローチを取ることができます。

DO-178C で文書化された要件に戻ると、レベル A（すなわち DAL A）のアプリケーションは、まさに危険にさらされているクリティカルアプリケーション相当を代表するものです。オブジェクトコード検証は、規格の要求が満たされていることを確認するうえで非常に有用なツールを提供します。しかし、これらの要求は曖昧であり、何が適切で必要かを決定することが開発チームの判断に委ねられているのは事実です。そして、それは究極のベストプラクティスではないかもしれませんが、現実的な妥協です。

OCV は、どのセクターのいかなる規格でも義務付けられていません。民間航空においてさえ、CAST や他のアプローチを述べた論文があるように、OCV を回避する方法があります。しかし、OCV を完全に回避することは、本稿で説明するものよりもさらに妥協であり、ベストプラクティスから離れることを意味します。

それだけではありません。コンパイラ開発者の検討事項と機能安全の要求との間の不一致は、防御的コードや一部の開発ツールの使用にも影響を及ぼします。たとえば、配列境界エラーの検出や未定義動作の処理などのように、問題の検出を妨げる可能性も出てきます。

OCV を使用しなければこのような問題は検出されないかもしれません。それらの問題が示すものは、高度にクリティカルなソフトウェアアプリケーションに使用されるコンパイラは、非常に稀な状況を除いて一般的に設計基準を満たしていると想定できるが、その設計基準には機能安全への検討事項が含まれていないという事実です。オブジェクトコード検証は、現在のところ、これらの検討事項を信頼の輪の中に取り込むための最も確実なアプローチです。

引用文献

[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 Common 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]. (https://ldra.com/products/tool-qualification-support-packs-tqsp/)

[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.

付録：ツール認定の例 2

次のソースコードは、LDRA ツール認定パックからの別の簡単な例を表しています。

```
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--;
  }
}
```

この例で、C コードの 100%カバレッジのために提供されている標準のテストケースは、この関数 `f_while10` に対する次の 2 つの呼び出しで構成され ます。

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

多くのコンパイラで、上記の呼び出しから生成されたアセンブラコードが実行され、ソースレベルではステートメントと分岐のカバレッジが 100% になりますが、この場合では結果のカバレッジが低くなります。

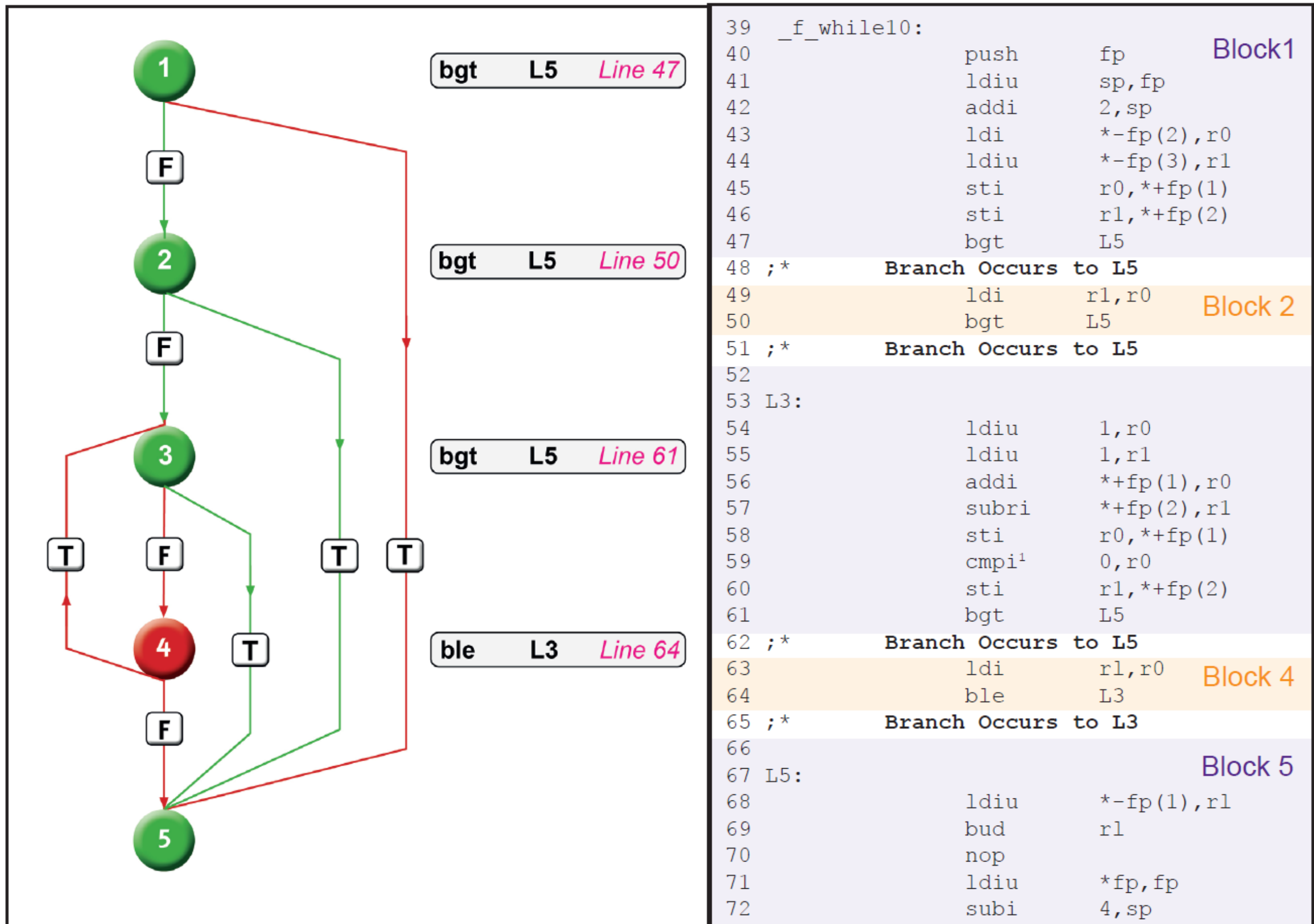

図 11：ソースコードでは 100%のカバレッジであるにもかかわらず、オブジェクトコードのカバレッジ解析では、未実行のパスがあることがわかる

完全なカバレッジを実現するには、新しいテストが必要です。

追加テストの考案

新しいテスト 1. 47 行目、Block 1 最後の L5 への分岐

このアセンブラコードは、while ループ本体の処理が行われるかどうかを確認するための初期検査です。処理が行われない場合、分岐により残りのコードはその地点からバイパスされます。次のテストケースは、while ループ本体を実行する条件のいずれも満たさないため、分岐が実行されます。

```
f_while10(10, -10)
```

それに応じてカバレッジが改善されます。

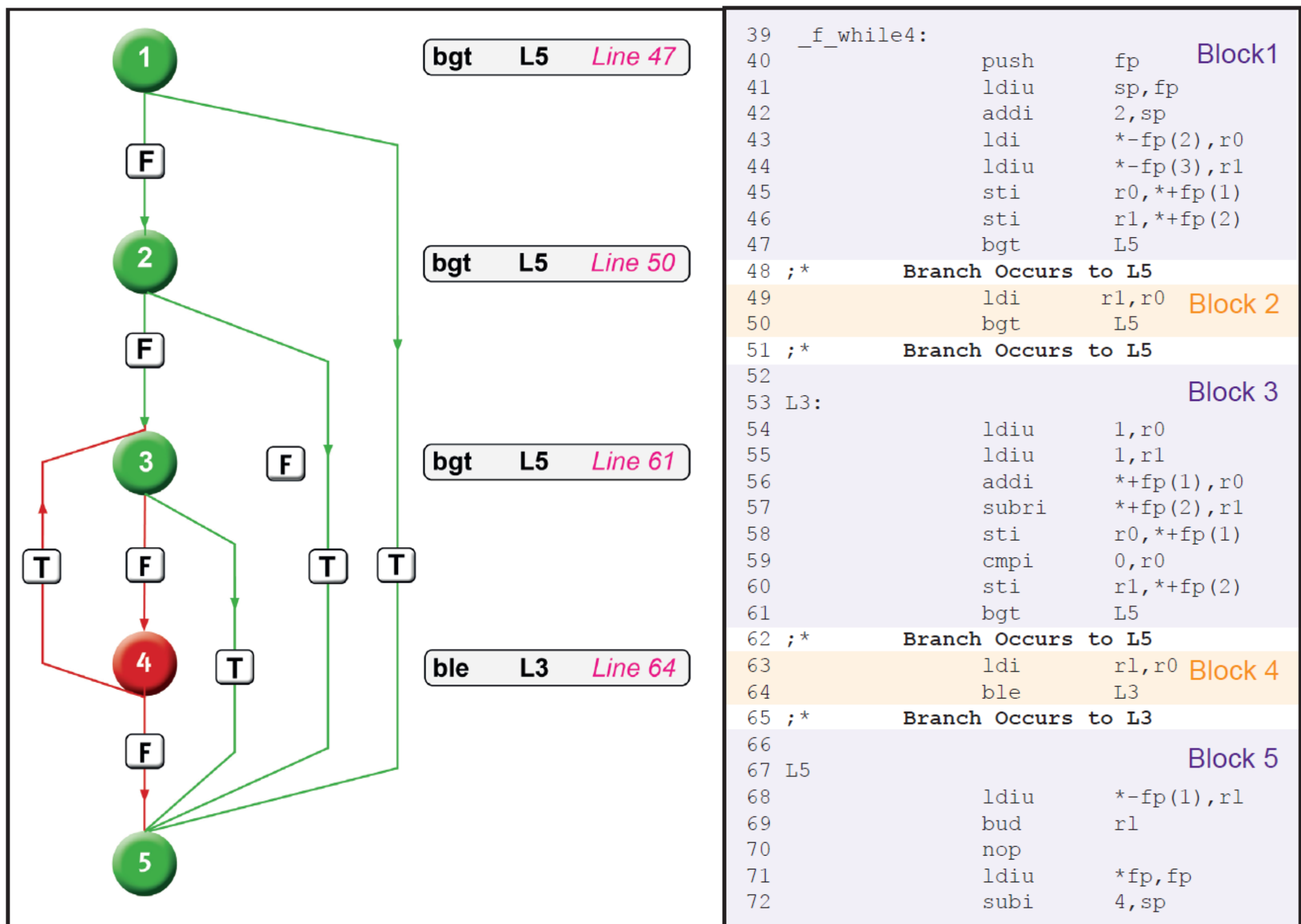

図 12：while ループ本体を実行する条件のいずれも満たさないテストケースは、そのループ本体をバイパスするコードを実行する

新しいテスト 2.　61 行目、Block 3 最後の L5 への分岐

　ここまで bgt（より大きい場合に分岐）命令での条件判定は true ばかりです。これは C ソースコードでいうと、これまでのどのテストケースでもループ本体が 2 回以上実行されたことがないということに対応します。以下の新しいテストケースで、カバレッジが再び向上します。

```
f_while10(-5, -5)
```

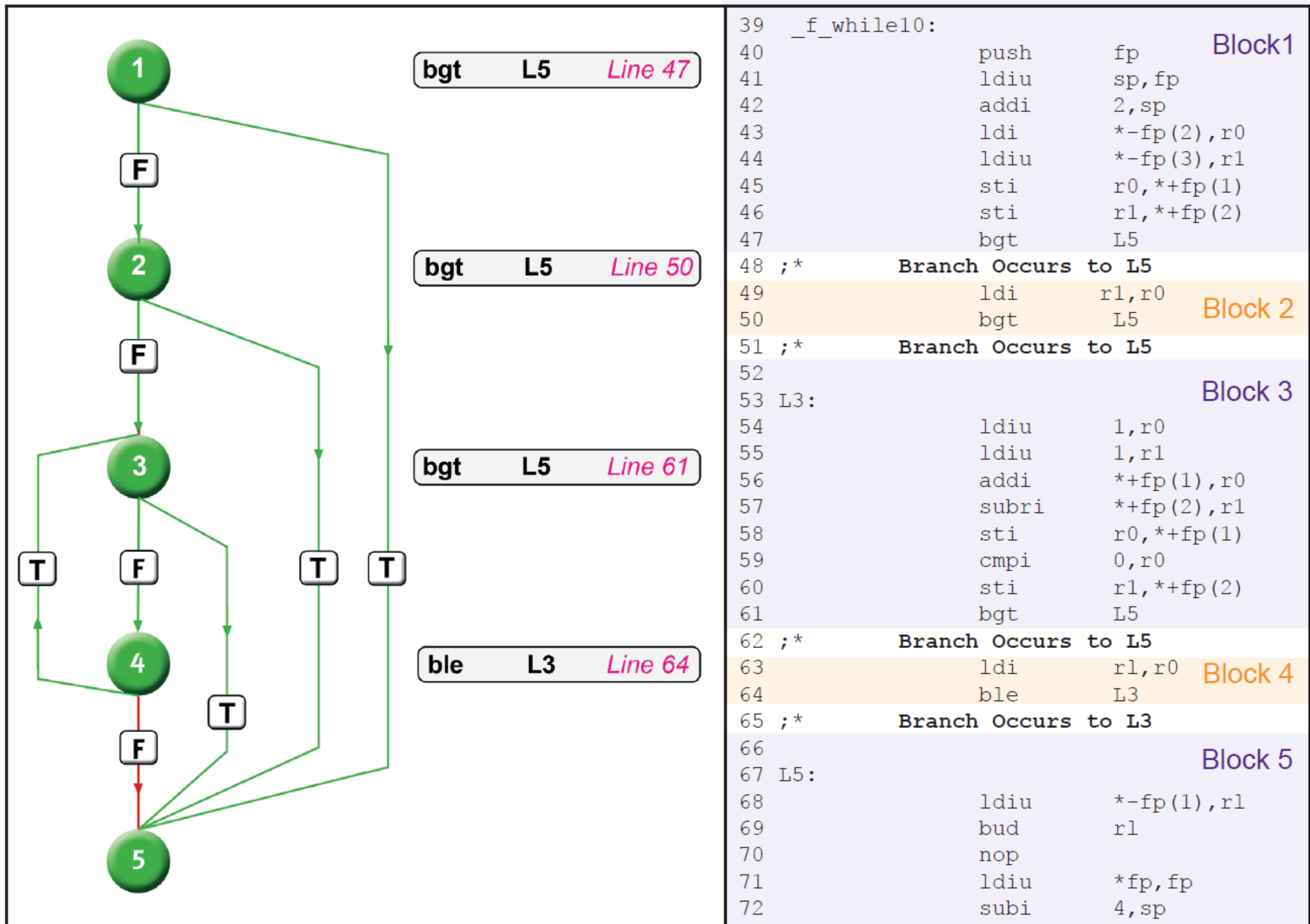

図 13：while ループ本体を 2 回以上実行するテストケースで、カバレッジが再び向上する

新しいテスト 3.　64 行目、Block 4 最後の L3 への分岐

アセンブラコードを調べると、64 行目の ble（より小さいか等しい場合に分岐）命令での条件判定は、while 文の制御式の第 2 項の式、

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

が false と評価される場合にのみ false となることがわかります。つまり、この第 2 項の条件でループの実行が終了します 。

コードの性質、また f_while10_local2 がデクリメントされるということから、前の例よりも対応が少し難しくなります。つまり、int 型の整数 f_while10_local2 が負数オーバーフロー[2]した場合にのみ、第 2 項の判定結果で while ループが終了できます 。この例では、int 型整数は 32 ビット符号付き 2 の補数表現であるため、負数オーバーフローは、

$$-1*(2^{31}) = -2147483648$$

からデクリメントした際に発生します。

したがって適切なテストは以下になります。

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

これにより f_while10_local2 が「負数オーバーフロー」して正になったときにループが終了することが保証されます。このテストを実施すると、アセンブラコードでのステートメントと分岐のカバレッジ 100%が達成されます。

この例のすべてのオブジェクトコードを実行する「main」関数は、次のようになります。

```
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);
}
```

[2] このテストケースは、未定義の動作（整数オーバーフロー）に依存してカバレッジを取得しています（オーバーフローで例外が発生し捕捉されるプロセッサでは、これは不可能です）。この例のコードには、ほぼ間違いなく欠陥があり、OCV テストでアルゴリズムの潜在的な欠陥が検出されます。

一度に複数の未使用パスを実行するような代替案を考案すれば、`f_while10` 関数の呼び出し回数を減らせる可能性があります。

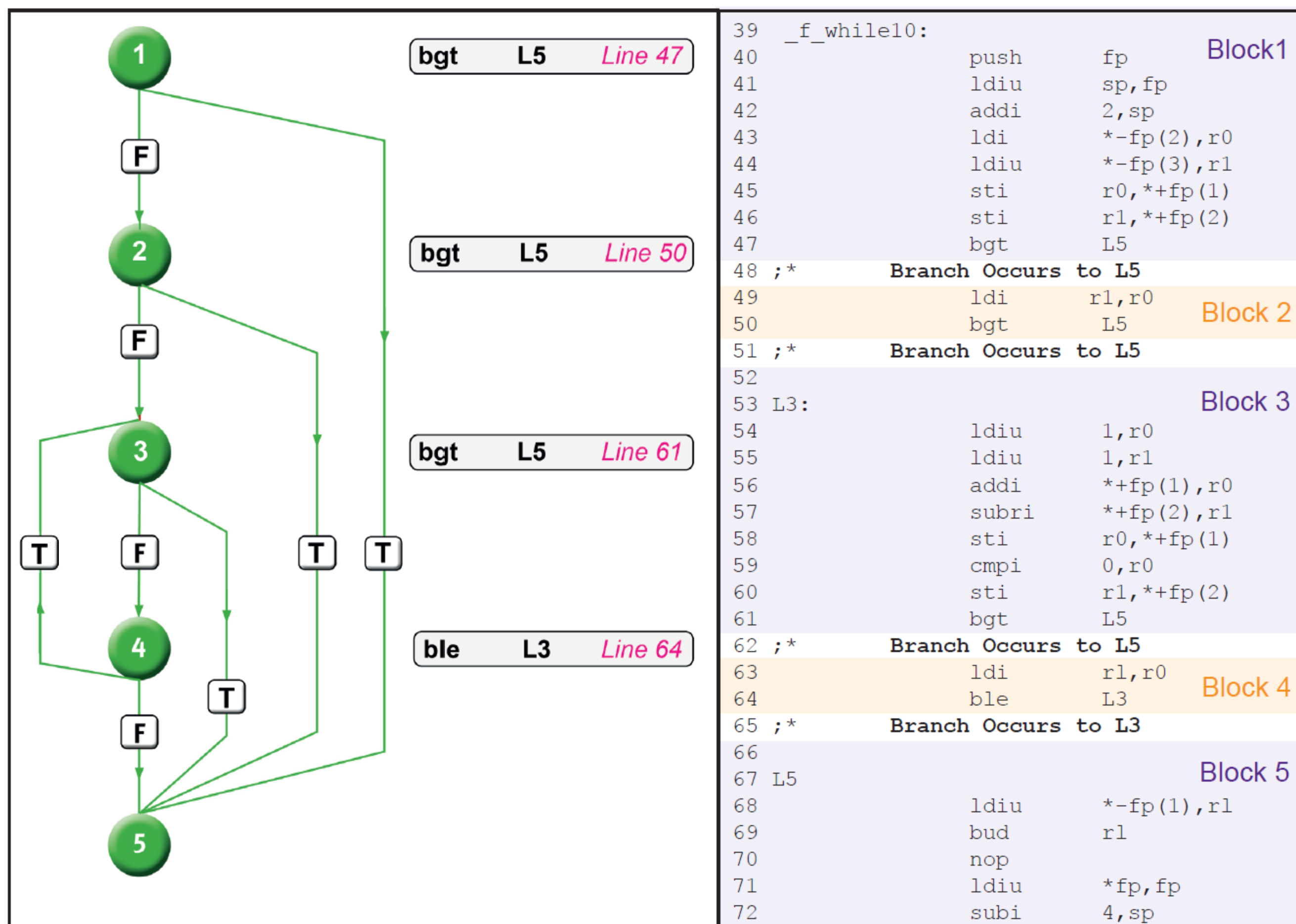

図 14：整数オーバーフローによるカバレッジが完了することの確認

www.ldra.com

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

FUJI SETSUBI

富士設備工業株式会社 電子機器事業部
http://www.fuji-setsu.co.jp