2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

# 付加価値の高い高位機能検証へのシフト

# Shifting functional verification to high value HLV

龍田純一, シーメンス EDA ジャパン株式会社, junichi.tatsuda@siemens.com

**アブストラクト—機能検証がプロジェクトの多くの時間を占めるようになるなかで、高位合成を適用するプロジェクトも増えつつある。そのようなプロジェクトにとって機能検証を高位にシフトすることには大きな期待が寄せられる一方で、機能検証の最大の課題であるカバレッジクローズを実現するために、いかに高位検証の価値を高めるかが重要である。本稿では高位モデルに対するコードカバレッジや機能カバレッジの測定について触れ、高位検証によって検証プランとテストをより完全なものとすることで、プロジェクト全体として機能検証の生産性と品質を上げる手法について述べる。**

***Abstract*- Functional verification is taking up more and more of a project's time. On the other hand, more and more projects are applying high-level synthesis. For such projects, shifting functional verification to high-level is highly expected. However, what is important in high-level verification is how to increase the value to achieve coverage closure, which is the biggest challenge in functional verification. This paper discusses the measurement of code coverage and functional coverage for high-level models, and describes methods to increase the productivity and quality of functional verification for the project as a whole by making verification plans and tests more complete through high-level verification.**

***キーワード — 高位設計; 高位検証; 機能検証; C++; Systems; コードカバレッジ; 機能カバレッジ; High Level Verification; Code Coverage; Functional Coverage;***

## I. はじめに

Wilson Research Group が 2022 年にワールドワイドで実施した RTL（Register Transfer Level）設計の機能検証[1]に関する調査によれば、ASIC/IC プロジェクト全体に対して機能検証タスクが占める割合は中央値で 50%～60%、そして多くのプロジェクトにおいて 60%～70%にもなっているものの、リスピンさせることなく初回完動に成功しているプロジェクトは全体の 24%にしかすぎない。そしてこのリスピンの第一の原因は、機能的、論理的な誤りである。RTL 機能検証の質をいかに効率よく上げるかは、多くのプロジェクトが直面している大きな課題であることが伺える。

一方で RTL 設計に先駆けて C++や SystemC など高位抽象度のモデルを用いて設計の妥当性を確認することは従来から広く行われている。またこのような高位抽象度モデルを入力とする高位合成を活用するプロジェクトも一定数あり、前出の調査によれば ASIC/IC プロジェクトでは 30%、FPGA プロジェクトでは 40%となっている。そこで高位抽象度モデルを活用するプロジェクトにおいて、後段の RTL 機能検証の質をいかに効率よく上げるかという課題に対して、より多くの検証を高位で高速に行うためのフローについて提案する。

## II. シミュレーション高速実行が可能な高位モデル

図 1.は前出の Wilson Research Group の 2022 年の機能検証調査からのデータである。高位合成を使うことのメリットや動機について分析している。

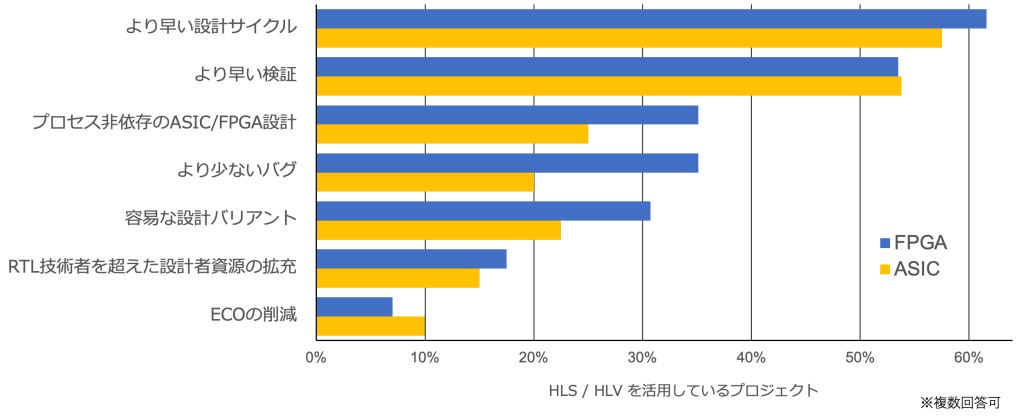


図 1. Wilson Research Group - 2022 機能検証調査「高位合成を使用する利点・動機」

高位合成を使うメリットとして「より早い設計サイクル」を挙げるプロジェクトも多いが、それと近い割合で挙げられているメリットが「より早い検証」である。一般的に高位合成では入力モデルとして C++や SystemC が多いが、そのような高位モデルは、等価な RTL モデルのシミュレーションに比べ、数百倍高速に実行が可能であることが知られている。プロジェクトのすべてが高位合成を活用しているわけではないものの、前出の機能検証における品質と生産性の課題に対して、高位モデルとその検証を解決策とするためには、RTL 機能検証で実施されている検証内容のより多くを高位モデルへとシフトさせる必要がある。

## III. 高位検証にシフトするための検証技術

RTL 機能検証で使用される検証技術には、コードカバレッジ、機能カバレッジ、アサーション、制約付きランダム検証、フォーマル検証、リントなどがある。これらの検証技術のすべてが高位で代替できるわけではない。そこで「機能検証における最大の課題」は何かが重要になる。図 2.は ASIC/IC および FPGA プロジェクトごとに示す「機能検証の最大の課題」を示している。

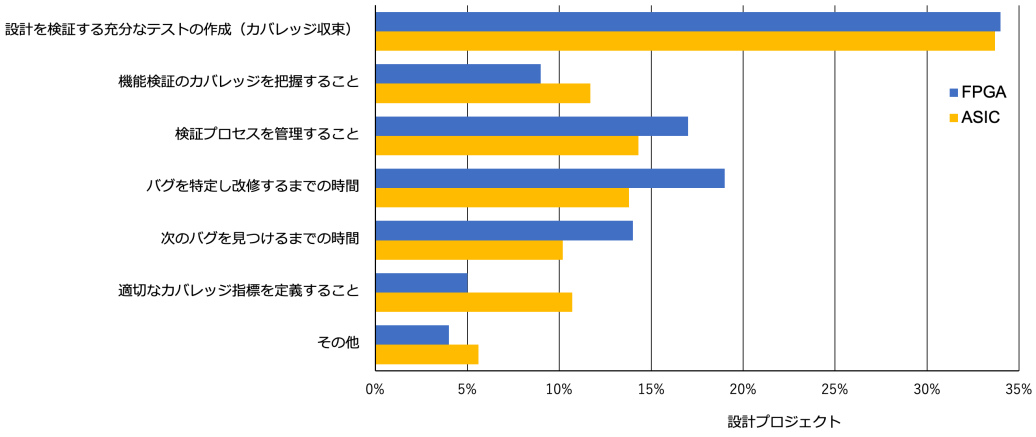


図 2. Wilson Research Group - 2022 機能検証調査「機能検証の最大の課題」

図 2.から分かるように、最も多くのプロジェクトが挙げる「機能検証の最大の課題」は、「設計を検証する充分なテストの作成（カバレッジ収束）」である。そこでこの最大の課題について、高位検証にシフトすることによって、課題解決につながる手法に取り組んでいる。その取り組みは主に以下の 3 点であり、いずれも特定ベンダーの高位合成ツールに依存しない手法である。

1. 高位モデルにおけるビット精度が表現可能なデータタイプの活用
2. ハードウェア実装を考慮した高位モデルにおけるコードカバレッジの測定
3. 高位モデルに対する検証プランに基づいた機能カバレッジの策定と測定

上記 1.については、すでにオープンソースのドメインに AC Data Type クラスライブラリ[2]として存在し、すでに広く使われていることから本稿のスコープ外とする。上記 2.については、gcov と同様の使い勝手で使用できる ccov が開発され使用されているが、そのコンセプトについて簡単に述べる。上記 3.については、IEEE Std. 1800 – 2017 SystemVerilog の機能カバレッジの仕様を手本に、C++クラスライブラリとして開発した。その概要と効果について述べる。

## IV. 高位検証におけるコードカバレッジ

C++や SystemC の高位モデルに対しては、従来からソフトウェアの多くのプロジェクトで使用され、Linux にも標準装備されている gcov を使うこともできる。ビルドの際にカバレッジのオプションを追加するだけで、コード活性化の解析やプロファイリング情報が取得できる。しかしながら、このようなユーティリティはソフトウェアコードの活性化カバレッジを対象としており、ハードウェアの RTL 実装を考慮したカバレッジを測定することはできない。ccov でどのようにハードウェア実装を考慮したコードカバレッジを測定するかについて述べる。

### *A. 関数のインスタンス単位のカバレッジ測定*

図 3.に C++による簡単な関数定義と、その関数を 2 回の呼出しているトップレベルのデザイン、そしてテストベンチを示す。

```
 1 #include "abs.h"
 2
 3 template <class T, class T2>
 4 void my_abs (T &a, T&b ){
 5    if (a<0) {
 6        b = -a ;
 7    } else {
 8        b = a ;
 9 }
10
11 void top_level (
12         data_in_a_t &a,
13         data_out_a_t &a_abs,
14         data_in_b_t &b,
15         data_out_b_t &b_abs,
16 ) {
17         my_abs(a, a_abs) ;
18         my_abs(b, b_abs) ;
19 }
```

```
// Testbench

data_in_a_t   a = -20 ;
data_in_b_t   b = 20 ;

top_level(a ,a_abs, b, b_abs);
```

図 3. 高位モデルとテストベンチ例

ソフトウェア用に開発された gcov では関数呼出しは関数単位へと合算されてレポートされるが、高位合成後のハードウェア実装記述では関数がインライン化され、個別のインスタンスとなる。つまりこのテストベンチでは a=-20 としてデザインの 17 行目が呼び出されても 17 行目に相当するハードウェアでは関数定義の 6 行目に相当する機能は実行されるが 8 行目に相当する機能は実行されない。同様に b=20 としてデザインの 18 行目が呼び出されても 18 行目に相当するハードウェアでは関数定義の 8 行目に相当する機能は実行されるが 6 行目に相当する機能は実行されない。コードカバレッジはそれぞれの呼出しで 50%としてレポートされることが必要であり、100%とするにはテストベンチに記述を追加する必要があることが高位モデルの段階で知ることができる。

*B. ループ展開を考慮したカバレッジ測定*

前出の機能検証調査において図 1.に示された選択肢にあるように、高位合成を使うことで「容易な設計バリアント」を得ることができる。つまり RTL 実装へと合成する際に、制約条件を設定することでさまざまなアーキテクチャを探索することができるが、この中で大きな要素がループの展開である。図 4.にループの展開（unrolling）と非展開（rolling）によるアーキテクチャの違いの例を示す。

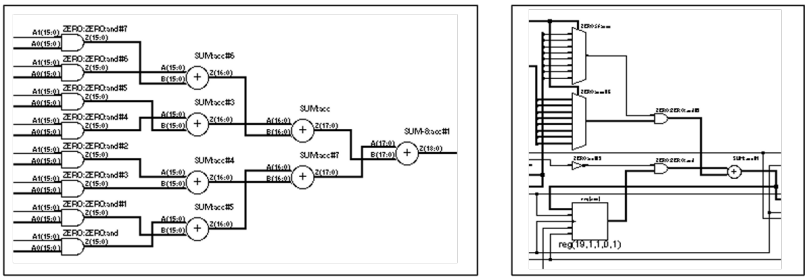

図 4. ループの展開（左）と非展開（右）によるアーキテクチャの違い

高位機能検証において RTL 実装と同じカバレッジを測定するには、高位合成においてどのようにループを扱うかに関する制約条件により、コードカバレッジの測定方法を変えられなくてはならない。

例として図 5.にループ文を含む高位モデルの記述例とテストベンチ記述例を示す。

```
1  #include "unrolling.h"
2  void testcase (
3     ac_int<8,false> &set_in,
4     ac_int<16,false> a[8],
5     ac_int<19,false> &result
6  ) {
7     ac_int<8,false> set = set_in ;
8     ac_int<16,false> partial[8] ;
9     ZERO:for(int i=0;i<8;i++) {
10      partial[i] = a[i] ;
11      if (set[i]==0) {
12           partial[i] = 0 ;
13    }
14   }
15    ac_int<19,false> acc = 0 ;
16    SUM:for(int i=0;i<8;i++) {
17              acc += partial[i] ;
18   }
19   result = acc ;
20 }
```

```
int main() {
  ac_int<8,false> set = 5;
  ac_int<16,false> a[8] =  {1,2,3,4,5,6,7,8};
  ac_int<19,false> result ;

  testcase(set, a, result);
}
```

図 5. ループ文を含む高位モデルとテストベンチ記述例

図 5.に示す高位モデル内の ZERO や SUM のループを展開するかしないかは、ハードウェア的な要素となるため、ソフトウェア用に開発された gcov などでは考慮されない。テストベンチとともに実行すると、ループを展開しない状態では 100%のコードカバレッジが得られるが、ループを展開するという指定をすると、if (set[i]==0) の分岐によって活性化されていないハードウェア要素が明らかになる。図 6.にループ非展開設定時のコードカバレッジを、図 7.にループ展開設定時のコードカバレッジを HTML で表示した例を示す。いずれも高位モデルの状態でのカバレッジ測定結果である。

**Coverage Summary by Structure:**

| Design Scope | Hits % | Coverage % |
|---|---|---|
| testcase_2 | 100.00% | 100.00% |

**Coverage Summary by Type:**

| Total Coverage: (Filtering Active) | | | | | 100.00% | 100.00% |
|---|---|---|---|---|---|---|
| **Coverage Type** | **Bins** | **Hits** | **Misses** | **Weight** | **% Hit** | **Coverage** |
| Statements | 11 | 11 | 0 | 1 | 100.00% | 100.00% |
| Branches | 6 | 6 | 0 | 1 | 100.00% | 100.00% |

図 6. ループ非展開設定時の高位モデルのコードカバレッジ

**Coverage Summary by Structure:**

| Design Scope | Hits % | Coverage % |
|---|---|---|
| testcase_2 | 77.27% | 71.42% |
| ZERO_1 | 50.00% | 50.00% |
| ZERO_2 | 75.00% | 75.00% |
| ZERO_3 | 50.00% | 50.00% |
| ZERO_4 | 75.00% | 75.00% |
| ZERO_5 | 75.00% | 75.00% |
| ZERO_6 | 75.00% | 75.00% |
| ZERO_7 | 75.00% | 75.00% |
| ZERO_8 | 75.00% | 75.00% |
| SUM_1 | 100.00% | 100.00% |
| SUM_2 | 100.00% | 100.00% |
| SUM_3 | 100.00% | 100.00% |
| SUM_4 | 100.00% | 100.00% |
| SUM_5 | 100.00% | 100.00% |
| SUM_6 | 100.00% | 100.00% |
| SUM_7 | 100.00% | 100.00% |
| SUM_8 | 100.00% | 100.00% |

**Coverage Summary by Type:**

| Total Coverage: (Filtering Active) | | | | | 77.27% | 71.42% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 28 | 26 | 2 | 1 | 92.85% | 92.85% |
| Branches | 16 | 8 | 8 | 1 | 50.00% | 50.00% |

図 7. ループ展開設定時の高位モデルのコードカバレッジ

高位モデルにおいてループ展開の情報を活用しながらコードカバレッジを取得することで、高速にシミュレーションが可能な状況でコードカバレッジが 100%となるようテストを追加することができる。

## V. 高位検証における機能カバレッジ

RTL 検証でもそうであったように、コードカバレッジはテストがいかに各ステートメント、ブランチなどを活性化できたか、テストの制御性の品質を示すことはできるが、検証対象のデザインが有する機能の正当性については示すことができない。そこで求められるのが機能カバレッジである。

高位モデルのテストにおいてはサイクル的なシーケンスは扱わないが、基本的に SystemVerilog の仕様にならい、C++ライブラリとして開発した。図 8.に機能カバレッジを構成する要素について示す。

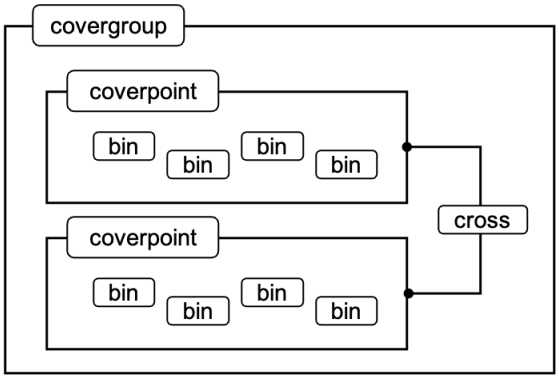


図 8. 高位モデルのテスト用機能カバレッジの構成

機能カバレッジはテストベンチ内に記述し、検証対象の高位モデルに対する入力、検証対象の高位モデルからの出力に対して適用する。機能カバレッジのモデリングにおいて、bin の取り扱いに関する以下のセマンティクスを用意している。

- Point-bin　特定の値を指定し、値が検出された場合にカバーとなる
- Range-bin　値の範囲を指定し、範囲内のいずれかが検出された場合にカバーとなる
- Full-Range-bin　範囲指定された各値について検出された場合にカバーとなる
- Conditional-bin　指定された条件が True となった場合にカバーとなる

- Auto-bin　　bin が指定されていない場合に Full-Range-bin を生成しチェックする
- Default-bin　　他の bin で指定したどの条件にも当てはまらない値をチェックする
- Illegal-bin　　本来ヒットしてはならず検出された場合にはエラーが出力される
- Ignore-bin　　指定されたビンの生成を抑制し coverpoint モデルの対象外とする

高位モデルのテストベンチに追加する機能カバレッジの記述例を図 9.に示す。

```
class MyCCoverGroup : public CCoverGroup {
public:
  ac_int<13, false> kType;
  ac_int<3, false> dataOut;
  MyCCoverGroup(std::string cg_inst_name):CCoverGroup(cg_inst_name)
  {
    CCoverPoint<ac_int<13, false> > *cp1 = AddCCoverPoint("cp_block_size", kType);
    cp1->AddPointBin("pb_zero", value<AC_VAL_MIN>(kType));
    cp1->AddRangeBin("rb_40_512_blksz_8", 40, 512);
    cp1->AddRangeBin("rb_2112_6144_blksz_64", 2112, 6144);

    CCoverPoint<ac_int<3, false> > *cp2 = AddCCoverPoint("cp_data_out", dataOut);
    cp2->AddFullRangeBin("rb_out",0,7);

    AddCCoverCross("cross_cp1_cp2",cp1, cp2);
  }
```

図 9. 高位モデルで使用する機能カバレッジの記述例

## VI.　高位検証におけるカバレッジ・クローズ

高位機能検証においても RTL 機能検証と同様にコードカバレッジと機能カバレッジを集約し、総合的に解析することでカバレッジ・クローズを図る。１つの高位モデルに対して複数テストを実行するが、毎回テストを行うごとに取得したコードカバレッジと機能カバレッジをカバレッジデータベースにダンプする。Accellera Systems Initiative ではカバレッジのデータベースに対する API 部分を UCIS（Unified Coverage Interoperability Standard）として標準化しており、この UCIS に対応する形でカバレッジをダンプすることで、RTL 機能検証で使用されている検証マネジメントツールをそのまま活用することができる。検証マネジメントツールによりダンプされた複数のカバレッジデータベースを１つに集約し、テストランキングなどを行いながら、総合的に解析する。

機能カバレッジについては検証プランを策定し、Excel などにまとめてレビュー、管理することが多いが、検証プランに含まれる検証項目ごとに機能カバレッジを対応させ、これもカバレッジデータベースに取り込むことで、検証プランとカバレッジ指標との紐付けも行える。

高位検証で取得、集約されたコードカバレッジと機能カバレッジの組合せから、大まかに以下のような分析[3]が可能となる。

表 1. コードカバレッジと機能カバレッジによる解析

| | | コードカバレッジ | |
|---|---|---|---|
| | | 低 | 高 |
| 機能カバレッジ | 低 | • テスト不足 | • コーナーケース未達<br>• DUT機能実装不足 |
| | 高 | • 未到達コードの存在<br>• 設計仕様にない機能の存在 | • 検証完了 |

このような分析をもとにテストを追加、修正し、高位モデルを対象とした機能検証におけるカバレッジをクローズする。RTL シミュレーションに対して数百倍高速に実行可能な環境において、コードカバレッジ、機能カバレッジ、それらの組合せを分析しながらテストそのものの完全性を高めることができる。

## VII. 考察とまとめ

論理合成技術が出現した当時、RTL シミュレーションは回路図ベースのゲートレベルシミュレーションよりも高速に実行可能であることから、広く受け入れられた。その後設計規模や複雑度が増すに連れて RTL 機能検証における課題が拡大し、プロジェクトのかなりの時間を占めるまでになった。

高位合成を使うプロジェクトが ASIC/IC では全体の 30%、FPGA では 40%に至っている中で、高位検証の価値を上げることは、プロジェクトの生産性と品質を高める上で必要である。その対策として提示したのは、以下の３つである。

1. 高位モデルにおけるビット精度が表現可能なデータタイプの活用
2. ハードウェア実装を考慮した高位モデルにおけるコードカバレッジの測定
3. 高位モデルに対する検証プランに基づいた機能カバレッジの策定と測定

1 は本稿のスコープ外とし、2 と 3 について解説した。

高位モデルは RTL シミュレーションに比べて数百倍高速に実行可能である。この環境で作成されたテストは、その後の RTL 機能検証においても、プロトコルに準拠したトランザクタを用いることで再利用することが可能である。RTL シミュレーションの速度でカバレッジを分析しながらテストの不足分を補う手法では、RTL 機能検証の段階で初めて検証プランの漏れや抜けに気づくことも想定される。高位モデルを対象に検証プランと検証用テストを完全なものにしておくことで、プロジェクト全体の検証効率を上げることが期待できる。

また本稿での提案した 3 つの技術適用以外にも、高位モデルに対するリントチェック、フォーマル検証を用いた到達性解析、AI/ML を用いたカバレッジホールの特定やテスト生成、Accellera Systems Initiative の Portable Test and Stimulus Standard との連携へと発展することも期待される。

## 参考文献


[1] Harry D. Foster, “2022 Wilson Research Group Functional Verification Study”

[2] https://hlslibs.org

[3] Verification Academy, “Coverage Cookbook”, https://verificationacademy.com/cookbook/coverage/introduction