Artgraphics

SystemVerilog による

ファンクショナルカバレッジとアサーション

Document Identification Number: ARTG-TD-003-2022
Document Revision: 1.2, 2024.04.09
アートグラフィックス
篠塚一也

SystemVerilog によるファンクショナルカバレッジとアサーション

〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

Functional Coverage and Assertions using SystemVerilog

8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

はじめに

本書は、SystemVerilog の検証機能、特に、ファンクショナルカバレッジとアサーションを初心者向けに分かり易く解説した資料です。これらの検証機能は互いに独立していますが、検証環境においては同時に使用される事も少なくありません。したがって、検証環境を構築する任務を持つ技術者は、何れかの時期において、これらの検証機能を知識として習得しておかなければなりません。とはいえ、SystemVerilog の言語仕様書 IEEE Std 1800-2023（以降、LRM と略称）の解説は、初心者にとって余りにも難しく重要な論点を見落としやすい事も事実です。一方、市販されている技術書はファンクショナルカバレッジとアサーション以外の話題も多く掲載しているため、必要な知識を得るまでの道のりが遠いという欠点があります。このような現状を踏まえ、本書は単刀直入にファンクショナルカバレッジとアサーションの解説をするようにしました。

ファンクショナルカバレッジは、決して難解な概念ではありません。確かに、多少複雑なシンタックスを持つ機能が存在しますが、理解に苦しむという機能はありません。ただし、カバレッジ機能の使い方においては多くの落とし穴があり細心の注意が必要です。特に、初心者にとっては、それらの落とし穴は間違いを経験するまで気が付かないという場合が多くあります。例えば、3 ビットのランダム変数 a と b の和に関するカバレッジを以下のように定義したと仮定します。結論から言えば、この定義にはカバレッジビンが定義されていないため間違いです。

```
class sample_t;
rand bit [2:0]   a, b;

covergroup cg;
   ab: coverpoint (a+b);
endgroup
...
endclass
```

何故なら、カバーポイント ab には、カバレッジビンとして auto[0]~auto[7]しか定義されないため、a+b の和としての値 8~14 をカバーできないからです。これは、SystemVerilog の加算の計算精度から引き起こされる間違い易い問題ですが、初心者がこの落とし穴に気が付くのは極めて稀です。要約すると、上記の記述例には以下の問題点があります。

- 計算式(a+b)の記述が正しくない。
- カバーポイント ab に対するカバレッジビンの定義がない。

これらの問題点とその解法に気が付かない方は本書を熟読すべきです。本書の第 3.4 節ではカバーポイントの定義法を詳しく解説しています。

検証作業を自動的に進めるためには、検証データの自動集計と検証データに対する結果との自動照合が必要になります。ファンクショナルカバレッジが検証データの自動集計を行います。一方、DUT への入力と DUT の処理結果との照合は、予め計算モデルを準備しておけば可能になります。しかし、生成された DUT の結果が時系列的に正しい順序で発生していたかを計算モデルで確認する事ができません。例えば、バスに入出力の要求が発生した時、その要求が 2~3 クロックサイクル以内に受け付けられなければならない仕様を確認するのは計算モデルでは不可能です。アサーションは、DUT の動作を時系列的に検証する機能を提供します。更に、システムの機能に不具合があった時、どこに問題があるかをピンポイントで指摘する機能も提供します。

アサーションは複雑な概念を持つ検証機能です。アサーションは DUT の動作を多くのクロッキングイベントに渡り検証するため、動作確認は非常に複雑になります。ハードウェアの動作を時系列的に検証するために、シーケンスやプロパーティを使用してハードウェア仕様を動作として表現しなければなりません。シーケンスは、クロッキングイベント発生時の状態

を表現し、プロパーティはそれらの状態がどうあるべきかを規定する機能です。そして、規定した通りにDUTが動作すれば、検証はパスします。ファンクショナルカバレッジと同様に、アサーションにも多くの落とし穴があります。それらの落とし穴を未然に防がなければ、正しい検証ができないばかりではなく、効率の良くない検証法となってしまいます。例えば、implicationオペレータ（|->、|=>）の左辺（antecedent）をレベルセンシティブに記述するか、またはエッジセンシティブにするかでアサーションの実行効率は大きく変わります（下表を参照）。本書は、アサーションの概念、機能、使い方、使用上の注意事項を詳細に解説し、正しく効率の良い検証法への知識を養います。

antecedent	仕様と注意点

レベルセンシティブ	antecedent がレベルセンシティブであると、条件が真であるとクロッキングイベント時に必ずconsequentの評価が行われ、条件が真である限りこの状態が続きます。
	
	`property data_end_rule1;`
	`data_end_exp |-> ##[1:2] $rose(frame);`
	`endproperty`

エッジセンシティブ	antecedent がエッジセンシティブであれば、特定のクロッキングイベント時にのみconsequentの評価が行われるため、効率が良くなります。
	
	`property data_end;`
	`$rose(data_phase) |-> ##[1:5] ready_exp;`
	`endproperty`

本書は、概要を含めて 4 章から構成され、第 2 章はランダムスティミュラスの生成、第 3 章はファンクショナルカバレッジ、第 4 章はアサーションの解説に割かれています。これらの検証機能の解説を単刀直入に開始するため、SystemVerilog に関する予備知識の解説を一切省いています。したがって、本書を読む前に SystemVerilog 入門書レベルの知識を備えておく必要があります。

ファンクショナルカバレッジとアサーションは独立した機能であり、どちらを先に学習しても構いませんが、本書の構成にしたがって順に読み進める事をすすめます。また、本書に掲載している使用例を自身で試し動作を確認する事をすすめます。なお、紙面の都合上、一部の記述は小さな書体で記述されています。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2022.09.20	1.0	初版。
2022.10.12	1.1	ランダムスティミュラスの生成の章を追加。
2024.04.09	1.2	IEEE Std 1800-2023 の追加機能の解説を追加。

目次

1 概要 1
1.1 検証環境におけるファンクショナルカバレッジとアサーション 1
1.2 本書の検証環境の構築 2
1.3 本書の対象者と目的 2
1.4 本書の構成 3
1.5 本書の記法 3
2 ランダムスティミュラスの生成 4
2.1 ランダム変数 4
2.2 RAND と RANDC 4
2.3 乱数発生メソッド 4
2.4 制約 6
2.4.1 一般式による制約 7
2.4.2 inside オペレータ 8
2.4.3 dist オペレータ 9
2.4.4 unique オペレータ 12
2.4.5 implication オペレータ 13
2.4.6 if-else 制約 14
2.4.7 foreach 制約 15
2.4.8 乱数決定順序文（solve 制約） 17
2.5 実行時に制約を定義する方法 18
2.6 ランダム変数の制御 19
2.7 制約の制御 21
2.8 STD::RANDOMIZE() 22
2.9 チェッカーとしての制約 23
2.10 RANDOMIZE(NULL) 24
3 ファンクショナルカバレッジ 26
3.1 概要 26
3.2 カバレッジモデルの定義 28
3.2.1 シンタックス 28
3.3 カバレッジ計算 28
3.3.1 カバーグループへの引数 30
3.3.2 サンプリングのタイミング指定 30
3.3.3 サンプリング関数 31
3.3.4 カバーグループの拡張 32
3.3.5 クラス外でのカバレッジ定義 34
3.3.6 カバレッジレポート 34
3.3.7 簡単な使用例 35
3.4 カバーポイントの定義 36
3.4.1 カバレッジビンの定義 37
3.4.2 カバーポイントの使用例 41
3.4.3 信号値の遷移 48
3.4.4 式のカバレッジ 50
3.4.5 制約とカバレッジ 53
3.5 クロスカバレッジ 54
3.5.1 シンタックス 54
3.5.2 クロスカバレッジビンの定義 58
3.6 自動カバレッジ収集 59
3.6.1 カバレッジ計算と検証コンポーネント 60
3.6.2 カバーグループの定義 60

3.6.3 カバレッジのサンプリング 61
3.6.4 自動カバレッジ収集例 61

4 アサーション 67

4.1 アサーションとは何か？ 67
4.1.1 概要 67
4.1.2 アサーションの種類 67
4.2 即時実行型アサーション 69
4.2.1 シンタックス 69
4.2.2 シンプル即時実行型アサーションと遅延即時実行型アサーション 70
4.3 並列型アサーション 72
4.3.1 検証条件の評価タイミング 72
4.3.2 サンプリング 72
4.3.3 アサーションクロック 73
4.3.4 アサーションの式 73
4.3.5 ブーリアン式（boolean expressions） 73
4.3.6 アサーションはマルチスレッド 73
4.3.7 アサーションとスケジューリング領域 74
4.3.8 シンタックス 74
4.3.9 implication オペレータ 75
4.3.10 レベルセンシティブ とエッジセンシティブ 77
4.3.11 アサーションスレッドダイアグラム 78
4.4 シーケンス 80
4.4.1 シーケンスの定義 80
4.4.2 シーケンスの結合 81
4.4.3 triggered()メソッド 82
4.4.4 便利なサンプル関数 85
4.4.5 シーケンスの操作 86
4.4.6 ##m 87
4.4.7 ##[m:n] 90
4.4.8 [*m] 92
4.4.9 [*m:n] 94
4.4.10 [=m] 96
4.4.11 [=m:n] 97
4.4.12 [->m] 99
4.4.13 [->m:n] 101
4.4.14 seq1 and seq2 102
4.4.15 seq1 intersect seq2 104
4.4.16 seq1 or seq2 105
4.4.17 exp throughout seq 107
4.4.18 seq1 within seq2 109
4.5 プロパーティ 110
4.5.1 概要 110
4.5.2 シーケンスとプロパーティ 111
4.5.3 プロパーティオペレータ 112
4.5.4 followed-by オペレータ 113
4.5.5 always property_expr 115
4.5.6 s_always [constant_range] property_expr 119
4.5.7 nexttime property_expr 121
4.5.8 nexttime [constant_expression] property_expr 123
4.5.9 s_nexttime property_expr 125
4.5.10 s_nexttime [constant_expression] property_expr 127
4.5.11 property_expr1 until property_expr2 129
4.5.12 property_expr1 s_until property_expr2 131
4.5.13 eventually [constant_range] property_expr 133

4.5.14 s_eventually property_expr 135
4.5.15 s_eventually [cycle_delay_const_range_expression] property_expr..... 137
4.6 マルチクロック 139
4.6.1 マルチクロックの定義 139
4.6.2 マルチクロックの例..... 141

5 参考文献 145

3　ファンクショナルカバレッジ

近年の検証手法では、デザインの大規模化に対応するためのテスト法として制約付きのランダムスティミュラス生成機能（CRT）を使用します。ランダムにテストデータを生成して検証する方法は便利で効果的なアプローチですが、実行したテストデータにより検証空間がどれだけカバーされたかを把握しなければ、検証計画を正確に管理する事はできません。したがって、テストデータを自動生成する検証手法には、検証に使用されたテストデータを自動的に集計する機能が必要になります。SystemVerilog のファンクショナルカバレッジはテストデータの自動集計機能を提供します。本章では、ファンクショナルカバレッジの概念と機能を解説します。

3.1　概要

ファンクショナルカバレッジは仕様のどれだけの部分が検査されたかを示す指標です。デザインを検証する意味ではなく、寧ろ、検査者、又は、検査計画の進捗度を示します。ゴールは、勿論、100%のカバレッジです。

ファンクショナルカバレッジは仕様を基にして、実際に検証で使用されたデータの確認をします。したがって、仕様を漏れなく、重複が無いように指定する必要があります。仕様は SystemVerilog の機能を使用して記述しますが、もれなく重複が無いように進めるのは検証者の責務です。例えば、3 ビットの `port` を計測すると仮定します。ただし、`port` は仕様上 0~5 の値だけを取る制約が付いているとします。その場合、検証用のトランザクションには以下のように `port` はランダム変数として定義されるだけでなく、値に関する制約も定義されます。

```
class simple_item_t;
...
rand bit [2:0]   port;

constraint C_PORT { port inside { [0:5] }; }
...
endclass
```

ランダム変数の宣言

port の制約定義

このように定義すると、クラスの `randimize()` メソッドを呼び出すとランダム変数 `port` には乱数が割り当てられます。検査で `port` が 0 から 5 の全ての値をとればカバレッジは 100% となりますが、一部の値を取らない場合、乱数発生に制約を追加して取り得る値を拡張しなければなりません。あるいは、テストする回数を増加しなければなりません。更にまた、もし `port` が 6 または 7 の値を取れば違反を報告しなければならない時もあります。

SystemVerilog では、ファンクショナルカバレッジの仕様をソースコード中に記述する事ができます。しかも、記述されたカバレッジ仕様はシミュレータにより実行されます。カバレッジ仕様を `covergroup` 文で記述し、`covergroup` に定義されているビルトインメソッド `sample()` を呼び出すとカバレッジ計算が行われます。例えば、前述の `port` に関しては、以下のようにカバレッジ条件を定義できます。

```
covergroup cg;

coverpoint port {
   bins value[] = { [0:5] };
}

endgroup
```

port のカバレッジ仕様

参考 3-1

`covergroup` をクラス内に定義できるのでトランザクション内にカバレッジ仕様を定義できますが、一般的にはトランザクション内にカバレッジ仕様を定義しません。理由は、トラン

ザクションは一時的に存在するオブジェクトであるため、カバレッジ情報を集計する時点でトランザクションが存在する保証がないからです。通常は、カバレッジ情報はカバレッジ計算をする検証コンポーネント内に定義します。
□

近年の検証環境におけるテストベンチでは、コレクターが DUT からのレスポンスを収集し、コレクター、又はモニターがカバレッジ計算を行います。図 3-1 は、その様子の一例を示しています。更に詳細な解析をするために、モニターは他の検証コンポーネントに DUT からのレスポンスをトランザクションとして送信します。これらの検証コンポーネントは、SystemVerilog クラスを使用して開発されます。

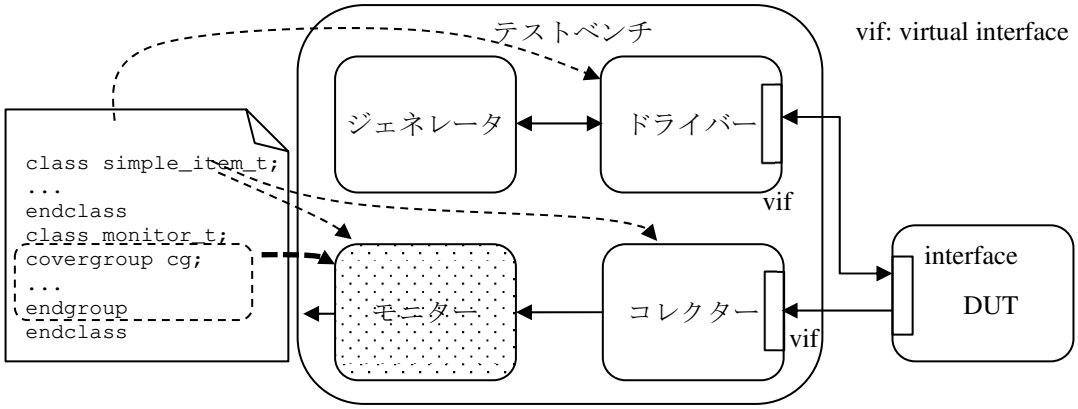

図 3-1　テストベンチに於けるカバレッジ計算の流れ

コレクターが DUT からのレスポンスを収集する時点では、テストに使用された DUT への入力と出力の対が確定しているので、コレクターはそれらの情報をトランザクションとして構成します。このトランザクションにより、実行結果の解析を完全に遂行する事ができます。例えば、予め計算モデルを準備しておけば、DUT の結果が正しい出力か否かを検証する事ができます。コレクターやモニターでは、その実行結果解析の一部分を担います。他の詳細な解析作業は、スコアボードやチェッカーに委ねられます。そのために、モニターはトランザクションを他の検証コンポーネントに送信します。

カバレッジ計算は、比較的簡単な解析処理であるため、しばしばモニターがカバレッジ計算を行います。その場合、モニターの定義には以下のようにカバレッジ定義を含みます。

```
class monitor_t;
...
covergroup cg with function sample(simple_item_t item);
    ...
endgroup

function void perform_coverage(input simple_item_t item);
    ...
    cg.sample(item);
endfunction

endclass
```

このようにカバレッジ定義をすると、`sample()`メソッドを介してトランザクションで使用されている変数に関するカバレッジ計算を行う事ができます。すなわち、DUT で使用された入出力信号に関するカバレッジ計算を行う事ができます。ここで、モニターがトランザクションをコレクターから受信すると、`perform_coverage()`メソッドが起動するように設定しておきます。こうして、ランダムスティミュラスを自動生成して自動検証する環境に自動

```
endfunction
endclass
```

テストベンチでは、乱数を発生してカバレッジ計算を以下のように行ないます。カバーグループに定義されているビルトインメソッド sample()を呼ぶとカバレッジ計算が行われます。

```
module test;
sample_t  sample;

initial begin
   sample = new;
   repeat (32) begin
      assert(sample.randomize());
      sample.cg.sample();
   end
end

endmodule
```

カバレッジ計算を行う

シミュレーションが終了するとカバレッジ情報がデータベースに記録されます。以下のように変数 a のカバレッジは 100%を達成しています。

```
COVERPOINT "a";
    COVERAGE 100.00 8 8;
    GOAL 100;
    WEIGHT 1;
    COMMENT "";
    ATLEAST 1;
    ABIN "auto" 0 6 "{0}";
    ABIN "auto" 1 2 "{1}";
    ABIN "auto" 2 2 "{2}";
    ABIN "auto" 3 7 "{3}";
    ABIN "auto" 4 3 "{4}";
    ABIN "auto" 5 4 "{5}";
    ABIN "auto" 6 3 "{6}";
    ABIN "auto" 7 5 "{7}";
ENDCOVERPOINT
```

■

3.4 カバーポイントの定義

カバレッジ計算の基本はカバーポイントです。カバーポイントの定義は、やや複雑なシンタックスを持ちます。細かいシンタックスには触れずに代表的なカバーポイントの定義法をマスターする事を目的とします。先ず、全体的なシンタックスは以下のようになります（[1]）。

```
[ [ data_type_or_implicit ] cover_point_identifier : ]
    coverpoint expression [ iff ( expression ) ] bins_or_empty
```

主なシンタックス要素の役割を表 3-3 に示します。

表 3-3　カバーポイントのシンタックス要素

シンタックス要素	説明
cover_point_identifier	カバーポイント名称を指定する事ができます。カバーポイントが式の場合には、カバーポイント名称を指定する方が結果を確認し易くなります。カバーポイントが変数の場合に、カバーポイント名称を省略すると、変数名がカバーポイント名称となります。

coverpoint expression	カバーポイントの定義を示すキーワードとカバーポイントを示す式を指定します。カバーポイントとしては、変数を指定するのが一般的です。
iff (expression)	条件（expression）が成立する時のみカバレッジが収集されます。
bins_or_empty	カバレッジ計算に使用するビンの定義を指定します。ビンはカバレッジを計算するためのグルーピングを意味します。例えば、32 ビットの整数型変数のカバレッジを求める際、全ての整数値のカバレッジを求める事は意味が無いと同時に不可能です。寧ろ、32 ビットの整数値を大、中、小等のグループに分けてカバレッジを求める方が合理的です。ビンはそのようなグルーピングを可能にします。ビン定義が省略されるとシミュレータは自動的にビンを定義します。それらのビンを auto ビンと言います。auto ビン数の標準値は 64 です。すなわち、auto[0]、auto[1]、…、auto[63]の 64 個のビンでカバレッジを計測します。標準値をカバレッジオプション auto_bin_max で変更する事ができます。

embedded カバーグループの例を以下に示します。カバーポイントとして式を指定しているので、カバーポイント名称を指定しています。

```
class sample_t;
rand bit [3:0]   a, b;

covergroup cg;
   a_xor_b: coverpoint (a^b);
endgroup

function new;
   cg = new;
endfunction
endclass
```

カバーポイント名称として a_xor_b を指定している

この例では、a_xor_b がカバーポイント名称です。省略すると変数名称がカバーポイント名称になります。上記の例のように、式（a^b）のカバレッジを求める場合には、カバーポイント名称を指定する方がカバレッジレポートを管理し易くなります。

3.4.1 カバレッジビンの定義

カバレッジビンの定義法には次のように幾つかの種類があります。

- 固定数のビンを定義する。
- それぞれの値に対して固有のビンを割り当てる。
- 複数の値に対して唯一つのビンを割り当てる。
- ビンを指定せずに auto ビンを使用する。
- 特殊なビンを定義する。

カバレッジビンを定義するための方法を以下に解説します。ビン定義はキーワード bins で始まります。

3.4.1.1 固定数のビン

固定数のビンを定義する方法として、LRM から引用した例を紹介します。この例ではカバーポイント a に 4 個のビンを定義しています。

4　アサーション

検証作業を自動的に進めるためには、検証データの自動集計と検証データに対する結果との自動照合が必要になります。既に紹介したように、ファンクショナルカバレッジが検証データの自動集計を行います。一方、DUT への入力と DUT の処理結果との照合は、予め計算モデルを準備しておけば可能になります。しかし、生成された DUT の結果が時系列的に正しい順序で発生していたかを計算モデルで確認する事はできません。例えば、バスに入出力の要求が発生した時、その要求が 2~3 クロックサイクル以内に受け付けられなければならない仕様を確認するのは計算モデルでは不可能です。アサーションは、DUT の動作を時系列的に検証する機能を提供します。更に、システムの機能に不具合があった時、どこに問題があるかをピンポイントで指摘する機能も提供します。このように、アサーションは自動検証機能を提供するだけでなく、システムのデバッガーの役割も果たします。本章では、このように優れた検証機能を提供するアサーションに関する基礎知識を解説します。

4.1　アサーションとは何か？

アサーションはシステムの動作（すなわち、仕様）を記述するための手段です。主として、アサーションはデザインの検証に使用されます。その他、入力となるスティムラスの検証にも使用されます。本項では、アサーションの意義、役割、種類を明確にします。

4.1.1　概要

アサーションではシーケンス、及び、プロパーティを用いて仕様を記述します。ただし、それらの記述自身だけでは起動しないため、`assert`、`cover`、`assume` 等のアサーション文を使用して検証を行ないます。検証とは、仕様とデザインが一致する事を確認する作業です。

アサーションにおいて、検証者は仕様、及び、合否処理（pass_statement と fail_statement）を記述します。その他の全ての検証タスクはシミュレータが担当します。例えば、デザインをシミュレーションし、（仕様 `!=` デザイン）となる状況が発生すると fail_statement が実行します。シーケンス（及びプロパーティ）は複数のクロッキングイベントを経過して評価を終了します。評価終了時に真となれば、仕様とデザインが一致すると判断されます。途中で評価が偽となれば、仕様とデザインが一致しないと判定されます。本書では、この判定結果をマッチ、及び、マッチしないと表現します。あるいは、pass、及び、fail とも表現します。

アサーションは機能を検証するための文として記述されて実行されます。アサーションには以下のような文があります。

① `assert`：デザインが遂行しなければならない動作（仕様）を検証します。
② `assume`：シミュレータに対しては、成立しなければならない環境条件を検証する手段となります。フォーマル・ツールの場合には、前提条件として使用します。
③ `cover`：動作に対してのカバレッジを収集します。
④ `restrict`：フォーマル・ツールへの制約を規定します。シミュレータには無効な命令です。

4.1.2　アサーションの種類

アサーションには、即時実行型と並列型の二種類があります。即時実行型アサーションは、通常の実行文と同じように動作し、即時に実行が終了します。したがって、時間を消費する概念がありません。一方、並列型アサーションは、デザインのシミュレーションと並行して実行し、検証仕様が成立、又は不成立するまで実行を続ける機能です。並列型アサーションでは、クロッキングイベント時に信号値をサンプリングして、検証仕様の評価を行います。検証仕様には有限、又は無限のクロックサイクル区間に渡って満たすべき仕様が含まれているため、並列型アサーションの実行は、複数のクロックサイクルを経過して検証を終了します。

以下の記述は、最も簡単な即時実行型アサーションの例です。

```
class sample_t;
rand logic [2:0]    port;
rand logic [31:0]   addr;
...
endclass

module test;
sample_t  sample;

   initial begin
      sample = new;
      assert( sample.randomize() );
      ...
   end

endmodule
```

即時実行型アサーションにより乱数発生処理の結果を確認する。

この記述は、ランダム変数への乱数発生処理が正しく行われたかを確認するために即時実行型アサーションを使用しています。もし乱数発生に失敗すれば、標準的なエラーメッセージが発行されます。即時実行型アサーションで検証すべき式は、一般に、ブーリアン式です。ブーリアン式は、最も簡単なシーケンスです。

並列型アサーションは、一回の評価では結論を出す事ができない複雑な検証仕様から構成されます。例えば、検証仕様を以下のように準備します。

```
property check_a_b;
   $rose(check) |-> a ##[1:3] b;
endproperty
```

この検証仕様は、クロッキングイベント発生時に、`$rose(check)==1'b1` であれば、`a==1'b1` で、しかも、1~3 クロックサイクルの間に `b==1'b1` が成立しなければならない仕様を記述しています。この仕様には、クロッキングイベントの定義が無いため、このままでは使用する事ができません。一般的には、この検証仕様を使用する側がアサーションの環境を設定します。例えば、以下のように検証環境を準備します。

```
module test(input clk,a,b);

default clocking cb @(posedge clk); endclocking

property check_a_b;
   $rose(check) |-> a ##[1:3] b;
endproperty

assert property (check_a_b)
   else $display("@%0t: FAIL",$time);

// ...
endmodule
```

標準クロッキングブロックの定義

並列型アサーションの実行

上記の設定では、標準クロッキングブロックを指定してクロッキングイベントの定義を行っています。この場合には、`(posedge clk)` が発生する度に、検証仕様 `check_a_b` が実行されます。そして、その実行は `assert` 文により指定されています。図 4-1 は検証仕様が満たされる一例を示しています。

ここで、s1 と s2 は独立したシーケンスです。シーケンス s1 はシーケンス s2 とは無関係に実行しています。シーケンス s2 において、(posedge clk) が発生した時、a==1'b1 であれば、1 クロックサイクル後に、s1 はマッチしなければなりません。しかし、s1 がいつ開始されるかは問題ではありません。

図 4-8 において、シーケンス s1 は時刻 T でマッチすると仮定しています。シーケンス s2 は時刻 T の次のクロッキングイベントで b==1'b1 なのでマッチします。

```
sequence s2;
   @(posedge clk) a ##1 s1.triggered ##1 b;
```

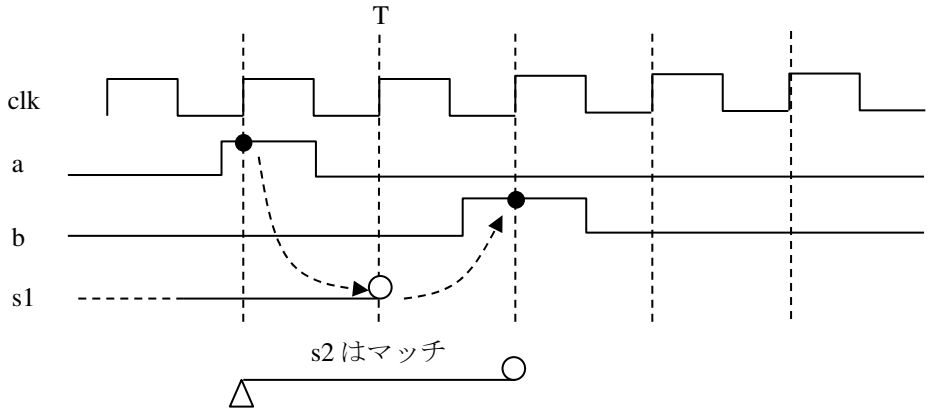

図 4-8　triggered メソッドの使用例

例 4-4　triggered メソッドの使用例

検証すべきプロパーティと独立に実行しているシーケンス unlock がマッチしている状態を参照する例を紹介します。下記のプロパーティ p では、現在のクロッキングイベント時に x が真であれば、その後の 1~2 クロックサイクルの間に unlock.triggered が真になる事を要求しています。

```
module test;
bit clk, a, b, c, x, y, check;

sequence unlock;
   @(posedge clk) a ##1 b ##1 c;
endsequence

property p;
   @(posedge clk) $rose(check) |->
          x ##[1:2] unlock.triggered() ##1 y;
endproperty

A1: assert property (p)
      else $display("@%3t: FAIL",$time);
A2: cover property (p)
      $display("@%3t: PASS",$time);

initial begin
   $display("--- Triggered_N001 ---");
   fork
      begin #20 check = 1; #20 check = 0; end
      begin #20 x = 1; #20 x = 0; end
      begin #80 y = 1; #20 y = 0; end
      begin #20 a = 1; end
```

```
        begin #40 b = 1; end
        begin #60 c = 1; end
    join
end

initial repeat( 11 ) #10 clk = ~clk;
endmodule
```

実行結果は以下のようになります。

```
@ 90: PASS
```

関連するアサーションダイアグラムは、図 4-9 のようになります。スレッドの活動は、表 4-6 のようになります。

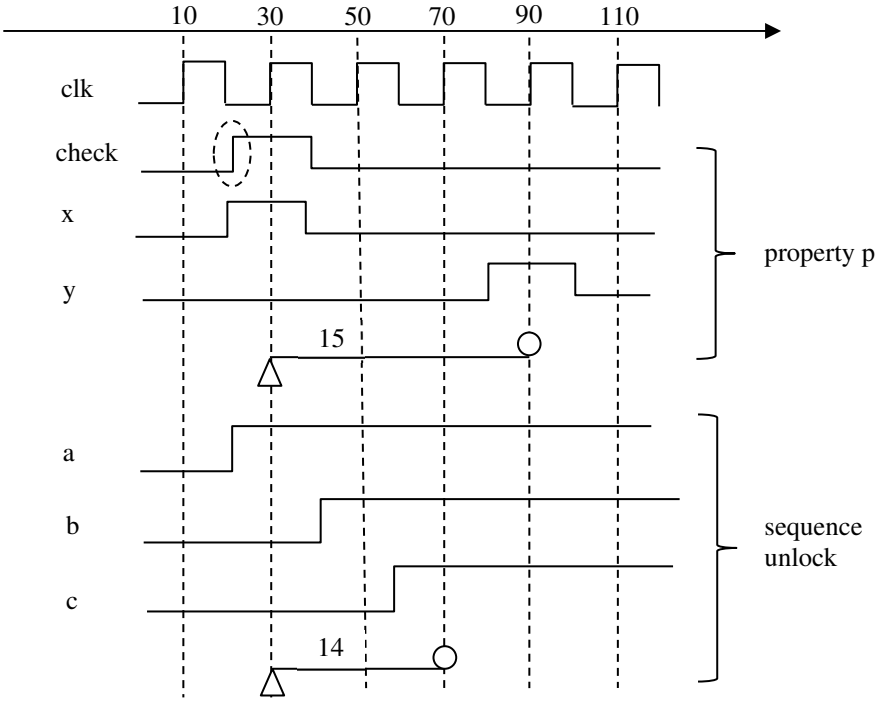

図 4-9　triggered メソッドの使用例

表 4-6　スレッドの活動

$time	スレッド	スレッドの活動
30	14	シーケンス unlock のスレッドが$time==30 で開始します。このスレッドは、$time==70 で pass します。
	15	$time==30 において、x==1'b1 なので、スレッド 15 は、その後の 1~2 クロックサイクルの間に unlock.triggered が真になるのを待ちます。 $time==70 でスレッド 14 が pass するので、スレッド 15 の unlock.triggered はマッチします。 その直後のクロックサイクルで y==1'b1 なので、スレッド 15 は $time==90 で pass します。

■