Artgraphics

SystemVerilog ファンクショナル・カバレッジ入門

Document Revision: 1.0, 2019.02.24

アートグラフィックス
篠塚一也

SystemVerilog ファンクショナル・カバレッジ入門

©2019 アートグラフィックス
〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

SystemVerilog Functional Coverage Primer

©2019 Artgraphics. All rights reserved.
8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

はじめに

この入門書は SystemVerilog によるファンクショナル・カバレッジを簡単に纏めた学習用の素材です。ファンクショナル・カバレッジの基礎を把握する事を主眼にしている為、詳細の機能については触れていません。この入門書を読了後に、原書（文献[1]）を熟読する事を薦めます。

誤字訂正、及び、説明の補足を随時行います。従って、常に本書の内容は更新を余儀なくされます。ご了承ください。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2019.02.24	1.0	第一版。

Copyright 2019 © Artgraphics. All rights reserved.

目次

1 概要 1
1.1 検証手法とファンクショナル・カバレッジ 1
1.2 SYSTEMVERILOG ファンクショナル・カバレッジ 1
1.3 カバレッジ・モデルの定義 1
2 ファンクショナル・カバレッジ 4
2.1 シンタックス 4
2.2 COVERGROUP への引数 4
2.3 サンプリングのタイミング指定 4
2.4 サンプリング関数（SAMPLE） 5
2.5 クラス内の COVERGROUP 5
3 カバー・ポイント 7
3.1 シンタックス 7
3.2 カバー・ポイント名称（COVER_POINT_IDENTIFIER） 7
3.3 カバレッジ収集条件 7
3.4 カバレッジ・ビン 7
3.5 カバレッジ計算に関する注意事項 10
3.6 ビンの定義 13
3.6.1 固定数のビン 13
3.6.2 それぞれの値にビンを確保する方法 13
3.6.3 唯一つのビン 14
3.6.4 auto ビン 14
3.6.5 default ビン 15
3.6.6 illegal_bins 16
3.6.7 ignore_bins 17
3.6.8 信号値の遷移（transitions） 18
3.7 制約とカバレッジ・ビンの定義 21
4 クロス・カバレッジ 24
4.1 シンタックス 24
4.2 クロス・カバレッジの記述 24
4.3 簡単な使用例 26
5 カバレッジ・オプション 27
6 補足 29
6.1 シミュレーション領域（SIMULATION REGIONS） 29
7 参考文献 31

Copyright 2019 © Artgraphics. All rights reserved.

1 概要

1.1 検証手法とファンクショナル・カバレッジ

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。

近年のテスト法は、CRT（Constrained Random Tests）をベースしています。CRT では、制約を満たすテスト・データをランダムに生成します。CRT は、膨大な検証空間を効率よく処理する為の効果的な方法です。従来の手作業によるデータ生成（directed tests）は、CRT がカバー出来ない特殊な集合を検査する為に使用されます。

CRT により自動的に、且つ、ランダムにデータを発生して検査をする場合、検査結果の集計も自動的に行なわなければなりません。従って、CRT による検証手法にはファンクショナル・カバレッジが必須なツールとなります。

1.2 SystemVerilog ファンクショナル・カバレッジ

SystemVerilog では、ファンクショナル・カバレッジをソース・コード中に記述する事が出来ます。そして、記述内容にデザインで使用している信号値のモニターをする機能を含む事が出来ます。信号値を値域で分類して集計する事も出来ます。SystemVerilog ファンクショナル・カバレッジの特徴を以下の様に纏める事が出来ます。

① SystemVerilog ファンクショナル・カバレッジでは、信号値だけでなく信号の値の遷移（transitions）の計測も行う事ができます。
② 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出来ます。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。
③ カバレッジを集計するタイミングを指定する事が出来ます。

1.3 カバレッジ・モデルの定義

SystemVerilog covergroup はカバレッジ仕様を定義する為の構文です。実際には、クラスと同じ様にデータ・タイプです。covergroup には以下の機能を含む事が出来ます。

- カバレッジ情報を取得するタイミングを示すクロック・イベント
- カバー・ポイント
- クロス・カバレッジ
- カバレッジ・オプション

covergroup を以下の様なスコープ内に定義する事が出来ます。

① package
② module
③ program
④ interface
⑤ checker
⑥ class

ここで、class の中に covergroup を定義する方法が最も一般的です。しかも、それは最も便利な方法です。例えば、次の様にして covergroup を定義します。

Copyright 2019 © Artgraphics. All rights reserved.

```
class sample_t;
      bit [31:0]  data;
      bit [2:0]   port;

      covergroup cov;
            coverpoint port;
            dataXport: cross data, port;
      endgroup

      function new;
            cov = new;
      endfunction
endclass
```

クラス内に covergroup を定義すると、その名称を持つ変数（この例では cov）が自動的に確保されます。covergroup を使用する為には、そのインスタンスを必ず作成しなければなりません。この例では、クラスのコンストラクタ内で covergroup のインスタンスにオブジェクトを割り当てています。

この例では、カバレッジを集計するタイミングを指定していない為、テストベンチで明示的にサンプリング時期を指定します。以下の様に covergroup に備えられているビルトイン関数 sample() を明示的に呼び出します。

```
module test;
sample_t    sample;
// ...
sample = new;
// ...
always @(posedge clk) begin
      // ...
      sample.cov.sample();
```

一方、以下の様にすると自動的な色彩が濃くなります。

```
class sample_random_t;
      rand bit [31:0]   data;
      rand bit [2:0]    port;
      bit               sw;

      covergroup cov @sw;
            coverpoint port;
            dataXport: cross data, port;
      endgroup

      function new;
            cov = new;
      endfunction
endclass
```

イベント@sw が起こる度にカバレッジ情報が収集されます。従って、テストベンチでは特別なサンプリング動作を記述する必要はありません。この例では、data、及び、port に rand 属性を付加しています。sample_random_t::randomize() 関数が呼ばれる度に、data、及び、port に値がランダ

Copyright 2019 © Artgraphics. All rights reserved.

ムに設定されます。従って、randomize() 関数を実行した後、sw に関するイベントが発生すれば、カバレッジ集計が行われます。例えば、sample_random_t::post_randomize() 内で sw のイベントを発生させれば、randomize() 関数が呼ばれる度にカバレッジ集計が行われる事になります。以下の記述例を参考にして下さい。

```
class sample_random_t;
      rand bit [31:0]    data;
      rand bit [2:0]     port;
      bit                sw;

      covergroup cov @sw;
            coverpoint port;
            dataXport: cross data, port;
      endgroup

      function new;
            cov = new;
      endfunction

      function void post_randomize();
            sw = ~sw;
      endfunction
endclass
```

この場合、sample_random_t::randomize() 関数が呼ばれる度にカバレッジが集計されます。この様に、SystemVerilog ファンクショナル・カバレッジでは、自動的、又は、マニュアルでカバレッジの集計をする事が出来ます。

ランダム・スティムラスを生成する場合、制約なしで生成する事は皆無です。例えば、上記の例で data は 32 ビット符号なし整数ですが、制約なしでは全く意味がありません。また、port も 0~7 の間の全ての値を取るとは限らないので制約が必要になります。

Copyright 2019 © Artgraphics. All rights reserved.

2 ファンクショナル・カバレッジ

2.1 シンタックス

SystemVerilog ファンクショナル・カバレッジは以下の様なシンタックスを持ちます（[1]）。

```
covergroup_declaration ::=
        covergroup covergroup_identifier [ ( [ tf_port_list ] ) ] [ coverage_event ] ;
                { coverage_spec_or_option }
        endgroup [ : covergroup_identifier ]
coverage_spec_or_option ::=
        {attribute_instance} coverage_spec
        | {attribute_instance} coverage_option ;
coverage_option ::=
        option.member_identifier = expression
        | type_option.member_identifier = constant_expression
coverage_spec ::=
        cover_point
        | cover_cross
coverage_event ::=
        clocking_event
        | with function sample ( [ tf_port_list ] )
        | @@( block_event_expression )
block_event_expression ::=
block_event_expression or block_event_expression
        | begin hierarchical_btf_identifier
        | end hierarchical_btf_identifier
hierarchical_btf_identifier ::=
hierarchical_tf_identifier
        | hierarchical_block_identifier
        | [ hierarchical_identifier. | class_scope ] method_identifier
```

キーワード covergroup 及び endgroup を使用してカバレッジ情報を定義します。

2.2 covergroup への引数

カバレッジ名称（covergroup_identifier）には引数を添える事が出来ます。引数を、covergroup 内で定義されていない変数を参照する為に使用する事が出来ます。この機能を使用すると、covergroup の定義が汎用化します。例えば、以下の様に指定をします。

```
covergroup cg(int low,int high);
       coverpoint a
       {
              bins b[] ={ [low:high] };
       }
endgroup
```

cg を new する時に、low、及び、high に対して値をパスしなければなりません。例えば、new(0,100)の様にして使用します。

2.3 サンプリングのタイミング指定

カバレッジを集計するタイミングを coverage_event で指定する事が出来ます。指定をしない場合には、明示的に sample()ファンクションを呼び出さなければなりません。例えば、次の様に coverage_event を指定します。

Copyright 2019 © Artgraphics. All rights reserved.

```
covergroup cov @sw;
      coverpoint a
      {
            bins fb[] = (1=>2=>3=>4), (6=>7);
            ignore_bins ignore_vals = (2=>3);
            bins others = default sequence;
      }
endgroup
```

イベント@sw が起こる時にカバレッジ情報が収集されます。従って、テストベンチでは特別なサンプリング動作を記述する必要はありません。

参考 2-1

同じシミュレーション時刻 T に於いてイベントが複数回起こる場合、複数回カバレッジの計算が行われます。時刻 T に於いて複数回イベントが起こっても唯一回のカバレッジ計算で済ませる為には、covergroup の type_option.strobe に 1 を設定しなければなりません。このオプションの標準値は 0 になっています。

2.4　サンプリング関数（sample）

covergroup の定義にサンプリングのタイミングを指定しない場合、テストベンチでサンプリングを明確に記述しなければなりません。次の様にサンプリングは sample()を呼び出して行います。

```
covergroup cov(int low,int high);
      coverpoint a { bins b[] ={ [low:high] }; }
endgroup
...
cov   cg1 = new(0,100);
...
cg1.sample();
```

ここで使用している sample はビルトイン（built-in）関数でパラメータを引き渡すことが出来ません。パラメータを引き渡す為には、sample を定義し直す必要があります。この機能により、covergroup でアクセスする事が出来ない変数のカバレッジを収集する事が出来ます。

例えば、次の様にして sample 関数を定義します。

```
covergroup cov with function sample(bit a,logic [1:0] x);
      coverpoint a;
      coverpoint x;
endgroup
```

2.5　クラス内の covergroup

クラス内に covergroup を定義すると以下の利点があります。

① covergroup の名称を持つハンドルが自動的に確保される。
② 簡単にクラス・プロパティのカバレッジを収集する事が出来る。

クラス内に定義された covergroup は embedded covergroup と呼ばれます。ハンドルに covergroup のインスタンスを割り当てる為には、コンストラクタ内で行います。それ以外の場所で割り当て

Copyright 2019 © Artgraphics. All rights reserved.

る事は出来ません。以下の例を参考にして下さい。

```
class packet_t;
bit [3:0]    a, b;
bit          sw, clk;

covergroup cg1 @sw;
      coverpoint a;
endgroup

covergroup cg2 @(posedge clk);
      coverpoint b;
endgroup

function new;
      cg1 = new;
      cg2 = new;
endfunction
endclass
```

covergroup のハンドル cg1、及び、cg2 はコンパイラーが自動的に確保してくれます。然し、ハンドルにオブジェクトを割り当てるのはユーザの仕事です。

Copyright 2019 © Artgraphics. All rights reserved.

3 カバー・ポイント

カバー・ポイントは個々の信号値の集計を行う機能です。信号値の遷移（transitions）の集計を計算する事も出来ます。

3.1 シンタックス

以下の様な複雑なシンタックスを持ちます（[1]）。

```
cover_point ::=
        [ [ data_type_or_implicit ] cover_point_identifier : ]
                coverpoint expression [ iff ( expression ) ] bins_or_empty
bins_or_empty ::=
        { {attribute_instance} { bins_or_options ; } }
        | ;
bins_or_options ::=
        coverage_option
        | [ wildcard ] bins_keyword bin_identifier [ [ [ covergroup_expression ] ] ] =
        { covergroup_range_list } [ with ( with_covergroup_expression ) ]
        [ iff ( expression ) ]
        | [ wildcard ] bins_keyword bin_identifier [ [ [ covergroup_expression ] ] ] =
cover_point_identifier with ( with_covergroup_expression ) [ iff ( expression ) ]
        | [ wildcard ] bins_keyword bin_identifier [ [ [ covergroup_expression ] ] ] =
set_covergroup_expression [ iff ( expression ) ]
        | [ wildcard] bins_keyword bin_identifier [ [ ] ]
                = trans_list [ iff ( expression ) ]
        | bins_keyword bin_identifier [ [ [ covergroup_expression ] ] ]
                = default [ iff ( expression ) ]
        | bins_keyword bin_identifier = default sequence [ iff ( expression ) ]
bins_keyword::= bins | illegal_bins | ignore_bins
covergroup_range_list ::= covergroup_value_range { , covergroup_value_range }
covergroup_value_range ::=
covergroup_expression
        | [ covergroup_expression : covergroup_expression ]
with_covergroup_expression ::= covergroup_expression26
set_covergroup_expression ::= covergroup_expression27
covergroup_expression ::= expression
```

3.2 カバー・ポイント名称（cover_point_identifier）

カバー・ポイントには名称（cover_point_identifier）を定義する事が出来ます。省略すると変数名称がカバー・ポイント名称になります。

3.3 カバレッジ収集条件

カバレッジの収集を制御する為には、iff (expression)を指定します。条件が成立する時のみカバレッジが収集されます。

3.4 カバレッジ・ビン

bins_or_empty でカバレッジのビンを定義します。ビンはカバレッジを計算する為のグルーピングを意味します。例えば、3 ビットの port 変数があるとします。port は 5 個の値（0~4）の値を取るとします。3 ビット変数に乱数を発生すると 8 種類の値が出現します。その内の 3 つの値は検証に無関係です。この様な場合、ビンを定義して無関係な値を類別する事が出来ます。勿論、制約を与えて、0~4 の範囲の乱数を発生する様にすればビンを定義する必要性は低くなります。但し、100%カバレッジを達成する事は出来ません。5~7 の間の数に対する auto ビンがカバーされません。

ビンの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 bins と言います。auto ビンの数をカバレッジ・オプション auto_bin_max（標準値は 64）で制御する事が出来ます。

Copyright 2019 © Artgraphics. All rights reserved.

例 3-1

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

covergroup cg;
      coverpoint        a;
      coverpoint        b;
endgroup

function new;
      cg = new;
endfunction
endclass

module test;
Simple      simple;

      initial begin
            simple = new;

            for( int i = 0; i < 32; i++ ) begin
                  simple.randomize();
                  simple.cg.sample();
            end
      end
endmodule
```

この例では、2 つの変数 a、及び、b のカバレッジを求めていますが、何れの変数に対してもビンを定義していません。この為、auto ビンが採用されます。

また、covergroup に対してサンプリングのタイミングをしていないので、テストベンチの for-ループ内で明示的に sample()を呼び出しています。

この例の場合には、何れのカバレッジも 100%になります。例えば、変数 a のカバレッジは以下の様になります。変数 b に対しても同様です。

```
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 5 "{2}";
      ABIN "auto" 3 3 "{3}";
      ABIN "auto" 4 4 "{4}";
      ABIN "auto" 5 5 "{5}";
      ABIN "auto" 6 5 "{6}";
      ABIN "auto" 7 2 "{7}";
ENDCOVERPOINT
```


Copyright 2019 © Artgraphics. All rights reserved.

集計表によると、a==0 は 6 回発生しています。for-ループで 32 の代わりに 8 を指定すると、一般的には、100%のカバレッジを達成する事は出来ません。十分な試行回数が必要です。
■

例 3-2

```
class SimpleBin;
rand bit [7:0]          a;

covergroup cg;
      coverpoint        a
      {
            bins zero = { 0 };
            bins small[2] = { [1:127] };
            bins large[4] = { [128:255] };
      }
endgroup

function new;
      cg = new;
endfunction
endclass

module test;
SimpleBin   simple;

      initial begin
            simple = new;

            for( int i = 0; i < 32; i++ ) begin
                  simple.randomize();
                  simple.cg.sample();
            end
      end
endmodule
```

この例ではビンを指定しています。変数 a は 0~255 の値を取るのでビンを使用して値を類別しています。small ビンは 1～127 の値を 2 分割しています。large ビンは 128~255 の値を 4 分割しています。以下の様な集計結果を得ます。

```
COVERPOINT "a";
      COVERAGE 100.00 7 7;
      GOAL 100;
      WEIGHT 1;
      COMMENT "";
      ATLEAST 1;
      SBIN "zero" 1 "{0}";
      FBIN "small" 0 7 "{[1:63]}";
      FBIN "small" 1 6 "{[64:127]}";
      FBIN "large" 0 3 "{[128:159]}";
      FBIN "large" 1 6 "{[160:191]}";
      FBIN "large" 2 5 "{[192:223]}";
      FBIN "large" 3 4 "{[224:255]}";
ENDCOVERPOINT
```

■

Copyright 2019 © Artgraphics. All rights reserved.

3.5 カバレッジ計算に関する注意事項

本節は文献（[2]）を参考にしています。カバレッジの計算には式を指定する事が出来ます。その場合には、式が評価する結果のビット数を意識する必要があります。例を示して説明をします。次の例は正しくない記述例です。

例 3-3

```
class Sample;
rand bit [2:0]     a;
rand bit [3:0]     b;

       function new;
             cg = new;
       endfunction

       covergroup cg;
             a_plus_b_4: coverpoint (a+b);
             a_plus_b_5: coverpoint (a+b+5'b0);
       endgroup
endclass

module test;
Sample       sample;

       initial begin
             sample = new;
             for( int i = 0; i < 256; i++ ) begin
                   sample.randomize();
                   sample.cg.sample();
             end
       end
endmodule
```

変数 a は 3 ビットで 0~7 の値を取ります。変数 b は 4 ビットで 0~15 の値を取ります。従って、式 (a+b)は 0~22 の値を取ります。然し、(a+b)は 4 ビットなので 16~22 の間の数は計測されていません。

coverpoint a_plus_b_5 は、その欠点を補う為に、5’b0 を加算して式の結果を 5 ビットに変更しています。この拡張により 0~22 の全ての値を計測する事が出来る様になります。然し、新たな問題が発生します。5 ビットである為、auto ビン数は 32 個となります。一方、式（a+b+5’b0）の最大値は 22 である為、auto[23]~auto[31]には値が計測されません。即ち、coverpoint a_plus_b_5 のカバレッジが 100%になる事は決してありません。

上記の記述を実行した結果を以下に示します。先ず、coverpoint a_plus_b_4 はカバレッジが 100%になりますが、0~15 の値のみ計測されています。16~22 の値は無視されている為、正しいファンクショナル・カバレッジではありません。

Copyright 2019 © Artgraphics. All rights reserved.

```
COVERPOINT "a_plus_b_4";
        COVERAGE 100.00 16 16;
        GOAL 100;
        WEIGHT 1;
        COMMENT "";
        ATLEAST 1;
        ABIN "auto" 0 8 "{0}";
        ABIN "auto" 1 16 "{1}";
        ABIN "auto" 2 21 "{2}";
        ABIN "auto" 3 20 "{3}";
        ABIN "auto" 4 20 "{4}";
        ABIN "auto" 5 15 "{5}";
        ABIN "auto" 6 17 "{6}";
        ABIN "auto" 7 20 "{7}";
        ABIN "auto" 8 10 "{8}";
        ABIN "auto" 9 20 "{9}";
        ABIN "auto" 10 20 "{10}";
        ABIN "auto" 11 17 "{11}";
        ABIN "auto" 12 11 "{12}";
        ABIN "auto" 13 13 "{13}";
        ABIN "auto" 14 14 "{14}";
        ABIN "auto" 15 14 "{15}";
ENDCOVERPOINT
```

次に、coverpoint a_plus_b_5 を見ます。auto[23]~auto[31]は全くカバーされていない為、カバレッジは 71.88% です。

```
COVERPOINT "a_plus_b_5";
        COVERAGE 71.88 23 32;
        GOAL 100;
        WEIGHT 1;
        COMMENT "";
        ATLEAST 1;
        ABIN "auto" 0 1 "{0}";
        ABIN "auto" 1 3 "{1}";
        ABIN "auto" 2 9 "{2}";
        ABIN "auto" 3 11 "{3}";
        ABIN "auto" 4 11 "{4}";
        ABIN "auto" 5 11 "{5}";
        ABIN "auto" 6 14 "{6}";
        ABIN "auto" 7 20 "{7}";
        ABIN "auto" 8 10 "{8}";
        ABIN "auto" 9 20 "{9}";
        ABIN "auto" 10 20 "{10}";
        ABIN "auto" 11 17 "{11}";
        ABIN "auto" 12 11 "{12}";
        ABIN "auto" 13 13 "{13}";
        ABIN "auto" 14 14 "{14}";
        ABIN "auto" 15 14 "{15}";
        ABIN "auto" 16 7 "{16}";
        ABIN "auto" 17 13 "{17}";
        ABIN "auto" 18 12 "{18}";
        ABIN "auto" 19 9 "{19}";
        ABIN "auto" 20 9 "{20}";
        ABIN "auto" 21 4 "{21}";
        ABIN "auto" 22 3 "{22}";
        ABIN "auto" 23 0 "{23}";
        ABIN "auto" 24 0 "{24}";
        --- 省略 ---
        ABIN "auto" 30 0 "{30}";
        ABIN "auto" 31 0 "{31}";
ENDCOVERPOINT
```

結論として、上記の covergroup の記述は正しくない事が分かります。この問題を解決するには、ビンを明示的に定義する必要があります。以下に書き換えた記述を示します。

Copyright 2019 © Artgraphics. All rights reserved.

```
class Sample;
rand bit [2:0]    a;
rand bit [3:0]    b;

      function new;
            cg = new;
      endfunction

      covergroup cg;
            a_plus_b: coverpoint (a+b+5'b0)
                  { bins value[] = { [0:22] }; }
      endgroup
endclass

module test;
Sample      sample;

      initial begin
            sample = new;
            for( int i = 0; i < 256; i++ ) begin
                  sample.randomize();
                  sample.cg.sample();
            end
      end
endmodule
```

結果は以下の様になります。100%カバレッジを達成します。

```
COVERPOINT "a_plus_b";
       COVERAGE 100.00 23 23;
       GOAL 100;
       WEIGHT 1;
       COMMENT "";
       ATLEAST 1;
       UBIN "value" 0 1 "{0}";
       UBIN "value" 1 3 "{1}";
       UBIN "value" 2 9 "{2}";
       UBIN "value" 3 11 "{3}";
       UBIN "value" 4 11 "{4}";
       UBIN "value" 5 11 "{5}";
       UBIN "value" 6 14 "{6}";
       UBIN "value" 7 20 "{7}";
       UBIN "value" 8 10 "{8}";
       UBIN "value" 9 20 "{9}";
       UBIN "value" 10 20 "{10}";
       UBIN "value" 11 17 "{11}";
       UBIN "value" 12 11 "{12}";
       UBIN "value" 13 13 "{13}";
       UBIN "value" 14 14 "{14}";
       UBIN "value" 15 14 "{15}";
       UBIN "value" 16 7 "{16}";
       UBIN "value" 17 13 "{17}";
       UBIN "value" 18 12 "{18}";
       UBIN "value" 19 9 "{19}";
       UBIN "value" 20 9 "{20}";
       UBIN "value" 21 4 "{21}";
       UBIN "value" 22 3 "{22}";
ENDCOVERPOINT
```

Copyright 2019 © Artgraphics. All rights reserved.

参考 3-1

```
bit [2:0]   a;
bit [3:0]   b;
bit [4:0]   sum;
// ...
a = 7;
b = 15;
sum = a+b;                                      ← ①
$display("sum=%b, a and b make %b",sum,a+b);    ← ②
```

上記の記述に於いて、式（a+b）が二度出現していますが、全く異なる値を示します。文①に使用されている式（a+b）は 5 ビットですが、文②に使用されている式（a+b）は 4 ビットです。従って、以下の様な結果を得ます。

```
sum=10110, a and b make 0110
```

sum には 5 ビットの和が求まっていますが、②に書かれた式（a+b）は 4 ビットとなっています。■

3.6 ビンの定義

3.6.1 固定数のビン

3.6.1.1 固定数のビン

固定数のビンを定義する為には、例えば、以下の様にします。

```
covergroup cg;
      coverpoint a
      {
            bins value[4] = { [1:10], 1, 4, 7 };
      }
endgroup
```

各ビンに値を一様に分配します。与えられた値の数は 13 個です。ビンは、`value[0]`、`value[1]`、`value[2]`、`value[3]`の 4 個になります。13/4 個の数を順に `value[0]`から割当てます。余った値を最後のビンに入れます。この宣言により、4 つのビンが以下の様に定義されます。

```
value[0] ::= { 1, 2, 3 }
value[1] ::= { 4, 5, 6 }
value[2] ::= { 7, 8, 9 }
value[3] ::= { 10, 1, 4, 7 }
```

`a` が値 2 を取ると、`value[0]`の計測値は `1` だけ加算されます。`a` が他の値を取る時も同様です。

3.6.2 それぞれの値にビンを確保する方法

それぞれの値に対してビンを定義するには以下の様にします。

Copyright 2019 © Artgraphics. All rights reserved.

```
covergroup cg;
      coverpoint a
      {
            bins value[] = { [1:10], 1, 4, 7 };
      }
endgroup
```

この宣言により、10 個のビンが以下の様に定義されます。

```
value[0] ::= { 1 }
value[1] ::= { 2 }
...
value[9] ::= { 10 }
```

3.6.3 唯一つのビン

複数の値に対して唯一つのビンを定義する為には以下の様にします。

```
covergroup cg;
      coverpoint a
      {
            bins value = { [1:10], [12:15] };
      }
endgroup
```

3.6.4 auto ビン

ビンの定義を省略するとシミュレータが自動的にビンを定義します。各値に対して一つのビンを割り当てる様にビンを定義します。例えば、3 ビットの変数に対しては、8 個のビンが割り当てられます。ビン数の最大値は auto_bin_max（標準値は 64）を超えない様に定義されます。例えば、32 ビットの変数に対しては 64 個のビンが定義されます。

例 3-4

```
class Simple;
rand bit [15:0]         a;

covergroup cg;
      coverpoint        a;
endgroup
function new;
      cg = new;
endfunction
endclass

module test;
Simple      simple;
      initial begin
            simple = new;
            for( int i = 0; i < 256; i++ ) begin
                  simple.randomize();
                  simple.cg.sample();
            end
      end
endmodule
```


Copyright 2019 © Artgraphics. All rights reserved.

この例ではビンの定義をしていない為、auto ビンが適用されます。変数 a は 16 ビットである為、auto_bin_max 値を遥かに超えます。この為、auto[0]~auto[63]は複数の値により共有される事になります。

尚、変数 a は符号なし 16 ビット整数です。従って、0~65535 の値を取り得ます。bit 型は符号なしである事に注意して下さい。この例に関しては、以下の様な集計を得ます。

```
COVERPOINT "a";
COVERAGE 100.00 64 64;
GOAL 100;
WEIGHT 1;
COMMENT "";
ATLEAST 1;
ABIN "auto" 0 5 "{[0:1023]}";
ABIN "auto" 1 1 "{[1024:2047]}";
ABIN "auto" 2 3 "{[2048:3071]}";
ABIN "auto" 3 5 "{[3072:4095]}";
ABIN "auto" 4 2 "{[4096:5119]}";
--- 省略 ---
ABIN "auto" 60 1 "{[61440:62463]}";
ABIN "auto" 61 1 "{[62464:63487]}";
ABIN "auto" 62 5 "{[63488:64511]}";
ABIN "auto" 63 3 "{[64512:65535]}";
ENDCOVERPOINT
```

3.6.5 default ビン

定義したビンに属しない値を受け取る為のビンです。このビンはカバレッジ計算の対象外になります。

例 3-5

```
class sample_t;
rand bit [3:0] a;
	covergroup cg;
		coverpoint a
		{
			bins zero = { 0 };
			bins low = { [1:3], 5 };
			bins high[] = { [8:$] };
			bins others = default;
		}
	endgroup
function new;
	cg = new;
endfunction
endclass

module test;
sample_t	sample;
	initial begin
		sample = new;
		for( int i = 0; i < 32; i++ ) begin
			sample.randomize();
			sample.cg.sample();
		end
	end
endmodule
```


Copyright 2019 © Artgraphics. All rights reserved.

a が 4、6、又は、7 であるとビン others に割り当てられます。ビン high の定義に$を使用している事に注意して下さい。上限値を想像する事が難しい場合、或いは、将来の拡張性を考慮する場合に使用すると便利です。この例に関しては、以下の様な集計を得ます。

```
COVERPOINT "a";
      COVERAGE 100.00 10 10;
      GOAL 100;
      WEIGHT 1;
      COMMENT "";
      ATLEAST 1;
      DEFAULTNAME "others";
      SBIN "zero" 2 "{0}";
      SBIN "low" 8 "{[1:3],5}";
      UBIN "high" 0 2 "{8}";
      UBIN "high" 1 1 "{9}";
      UBIN "high" 2 3 "{10}";
      UBIN "high" 3 3 "{11}";
      UBIN "high" 4 3 "{12}";
      UBIN "high" 5 2 "{13}";
      UBIN "high" 6 2 "{14}";
      UBIN "high" 7 2 "{15}";
      SBIN "others" 4 "{unmatched_values}";
ENDCOVERPOINT
```

3.6.6 illegal_bins

信号値としてあり得ない値が出現した場合の計測を illegal_bins 文で指定します。このビンに記録された値はカバレッジから除外されます。更に、実行時のエラー・メッセージの対象になります。即ち、illegal_bins に集計される値が出現した場合、エラー・メッセージが発行されます。例えば、次の様に定義します。

```
covergroup cg;
      coverpoint b
      {
            illegal_bins bad_value = {1,2,3};
            illegal_bins bad_transition = (4=>5=>6);
      }
endgroup
```

例 3-6

この例では、クラスを使用せずに covergroup をテストベンチ内に定義しています。更に、この例では信号値ではなく信号値の遷移を計測しています。集計する信号値の遷移

`(1=>2=>3=>4)`

は illegal_bins

`(2=>3)`

とオーバーラップしている事に注意して下さい。

Copyright 2019 © Artgraphics. All rights reserved.

```
module test;
logic [3:0] a;
bit         clk;

      covergroup cov @(posedge clk);
            coverpoint a
            {
                  bins fb[] = (1=>2=>3=>4), (6=>7);
                  illegal_bins ignore_vals = (2=>3);
                  bins others = default sequence;
            }
      endgroup

      initial begin
            cov   cg;
            cg = new;
      end

      initial
            repeat (13) #10 clk = ~clk;

      initial begin
            a = 1;
            #20 a = 2;
            #20 a = 3;
            #20 a = 4;
            #20 a = 5;
            #20 a = 6;
            #20 a = 7;
      end
endmodule
```

この場合、エラー・メッセージが発行されます。fb[0]は決して集計されないのでカバレッジが100%になる事はあり得ません。以下の様な結果を得ます。

```
COVERPOINT "a";
      COVERAGE 50.00 1 2;
      GOAL 100;
      WEIGHT 1;
      COMMENT "";
      ATLEAST 1;
      LBIN "fb" 0 0 "(1=>2=>3=>4)";
      LBIN "fb" 1 1 "(6=>7)";
ENDCOVERPOINT
ENDCOVERGROUPINSTANCE
```

3.6.7 ignore_bins

カバレッジの対象外の値をこの文で指定します。このビンに登録される値はカバレッジから除外されます。エラー・メッセージも発行されません。例えば、次の様に定義します。

Copyright 2019 © Artgraphics. All rights reserved.

```
covergroup cg;
      coverpoint a
      {
            ignore_bins ignore_vals = {7,8};
            ignore_bins ignore_trans = (1=>3=>5);
      }
endgroup
```

例 3-7

```
module test;
logic [3:0] a;
bit         clk;

      covergroup cov @(posedge clk);
            coverpoint a
            {
                  bins fb[] = { [1:10] };
                  ignore_bins ignore_vals = {7,8};
                  bins others = default;
            }
      endgroup

      initial begin
            cov   cg;
            cg = new;
      end

      initial
            repeat (10) #10 clk = ~clk;

      initial begin
            a = 1;
            #20 a = 3;
            #20 a = 5;
            #20 a = 7;
            #20 a = 9;
      end
endmodule
```

ignore_bins が{7,8}である為、実際に集計すべき値は、1~6、と 9~10 の 8 個の値です。テストベンチでは、a に 1、3、5、7、9 を割り当てています。7 は無視されるので、1、3、5、9 の 4 個の値が有効になります。従って、カバレッジは 50%となります。

3.6.8 信号値の遷移（transitions）

3.6.8.1 シンタックス

以下の様なシンタックスを持ちます（[1]）。複雑なシンタックスであると共に豊富な機能を備えています。

Copyright 2019 © Artgraphics. All rights reserved.

```
bins_or_options ::=
        coverage_option
        | [ wildcard ] bins_keyword bin_identifier
                [ [ [ covergroup_expression ] ] ] =
                { covergroup_range_list } [ with ( with_covergroup_expression ) ]
                [ iff ( expression ) ]
        | [ wildcard ] bins_keyword bin_identifier [ [ [ covergroup_expression ] ] ] =
                cover_point_identifier with ( with_covergroup_expression ) [ iff
                ( expression ) ]
        | [ wildcard ] bins_keyword bin_identifier [ [ [ covergroup_expression ] ] ] =
        set_covergroup_expression [ iff ( expression ) ]
        | [ wildcard] bins_keyword bin_identifier [ [ ] ] = trans_list [ iff
                ( expression ) ]
        ...
bins_keyword::= bins | illegal_bins | ignore_bins
covergroup_range_list ::= covergroup_value_range { , covergroup_value_range }
trans_list ::= ( trans_set ) { , ( trans_set ) }
trans_set ::= trans_range_list { => trans_range_list }
trans_range_list ::=
trans_item
         | trans_item [* repeat_range ]
         | trans_item [-> repeat_range ]
         | trans_item [= repeat_range ]
trans_item ::= covergroup_range_list
repeat_range ::=
covergroup_expression
        | covergroup_expression : covergroup_expression
```

3.6.8.2　記述例（[1]）

本節の内容は文献（[1]）からの抜粋です。信号値の状態の変化を=>で記します。例えば、

```
value1 => value2
```

は連続する二つのサンプリングで信号値が value1 から value2 に変化する事を意味します。変化する状態を幾つか連続する事が出来ます。例えば、

```
value1 => value2 => value3 => value4 => value5
```

の様に遷移を記述する事が出来ます。遷移のリストを両辺に指定する事が出来ます。例えば、

```
1, 2 => 3, 4
```

は、

```
(1=>3), (1=>4), (2=>3), (2=>4)
```

と同じです。繰り返し記号[*m]も使えます。例えば、

```
1 [*5]
```

は

```
1 => 1 => 1 => 1 => 1
```

と同じです。アサーションと同様に goto オペレータ[->m]が存在します。例えば、

```
1 [->3]
```

```
...=>1...=>1...=>1
```


Copyright 2019 © Artgraphics. All rights reserved.

と同じです。ここで、...には値 1 が現れない事とします。例えば、

```
1 => 3 [->3] => 5
```

は

```
1...=>3...=>3...=>3 =>5
```

と同じです。同様に、不連続な繰り返し[=m]も存在します。

```
1 [=3] => 2
```

は

```
...=>1...=>1...=>1...=>2
```

と同じです。

信号値の遷移をビンの定義に指定する事が出来ます。例えば、次の様に信号値の変化をビンに定義する事が出来ます。

```
bit   clk;
bit [3:0]   a;
covergroup cov @(posedge clk);
      coverpoint a
      {
            bins single_bin = (4 => 5 => 6), ([7:9],10 => 11,12);
            bins array_bin[] = (4 => 5 => 6), ([7:9],10 => 11,12);
      }
endgroup
```

single_bin には以下の遷移が含まれます。

```
4=>5=>6, 7=>11, 8=>11, 9=>11, 10=>11, 7=>12, 8=>12, 9=>12, 10=>12
```

array_bin は次の様に定義されます。

```
array_bin[4=>5=>6], array_bin[7=>11], ...,array_bin[10=>12]
```

例 3-8

この例では goto オペレータ[->m]を使用します。ビン tc で以下の様に使用しています。

```
bins tc = (12 => 3 [-> 1]);
```

これは、12 ... => 3 を意味します。即ち、a==12 となり、何時か、a==3 になれば tc に出現数が 1 だけ加算されます。

Copyright 2019 © Artgraphics. All rights reserved.

```
module test;
logic [3:0]       a, b;
logic signed [7:0]      sa, sb;
bit               clk;

      covergroup cov @(posedge clk);
            option.per_instance = 1;
            coverpoint a
            {
                  bins ta = (4 => 5 => 6), ([7:9],10 => 11,12);
                  bins uta[] = (4 => 5 => 6), ([7:9],10 => 11,12);
                  bins tc = (12 => 3 [-> 1]);
                  bins allother = default sequence;
            }
      endgroup

      initial cg = new;
      initial repeat (17) #10 clk = ~clk;
      initial begin
            a = 4;
            #20 a = 5; #20 a = 7; #20 a = 11; #20 a = 8;
            #20 a = 12;#20 a = 2; #20 a = 2; #20 a = 3;
      end
endmodule
```

結果は以下の様になります。

```
COVERPOINT "a";
      COVERAGE 36.36 4 11;
      GOAL 100;
      WEIGHT 1;
      COMMENT "";
      ATLEAST 1;
      TBIN "ta" 2;
      LBIN "uta" 0 0 "(4=>5=>6)";
      LBIN "uta" 1 1 "(7=>11)";
      LBIN "uta" 2 0 "(8=>11)";
      LBIN "uta" 3 0 "(9=>11)";
      LBIN "uta" 4 0 "(10=>11)";
      LBIN "uta" 5 0 "(7=>12)";
      LBIN "uta" 6 1 "(8=>12)";
      LBIN "uta" 7 0 "(9=>12)";
      LBIN "uta" 8 0 "(10=>12)";
      TBIN "tc" 1;
ENDCOVERPOINT
```

■

3.7 制約とカバレッジ・ビンの定義

通常は、クラスのプロパーティのカバレッジを求めます。しかも、プロパーティには制約が課されるのが普通です。その様な状況では、ビンの定義が必須になります。即ち、制約付きのプロパーティに対してカバレッジを求める場合、ビンの定義が無いと 100%カバレッジを達成する事は出来ません。

Copyright 2019 © Artgraphics. All rights reserved.

例 3-9

```
class simple_data_t;
rand bit [3:0]    data;

      function new;
            cg = new;
      endfunction
      constraint C { data[1:0] == 2'b00; }
      covergroup cg;
            coverpoint data;
      endgroup
endclass

module test;
simple_data_t     simple_data;

      initial begin
            simple_data = new;

            for( int i = 0; i < 32; i++ ) begin
                  simple_data.randomize();
                  simple_data.cg.sample();
            end
      end
endmodule
```

この例では、プロパーティ data に対して制約を与え、4 の倍数の値だけを取る様に設定しています。data のカバレッジを集計する様に設定されていますが、ビンの定義がありません。従って、auto ビンが auto[0]~auto[15]として作成されます。然し、data は 0、4、8、12 の値のみ取る為、auto ビンの 1/4 しかカバーされません。詰まり、カバレッジは最大 25%です。言い換えると、上記の記述は正しくありません。因みに、以下は集計結果です。

```
COVERPOINT "data";
      COVERAGE 25.00 4 16;
      GOAL 100;
      WEIGHT 1;
      COMMENT "";
      ATLEAST 1;
      ABIN "auto" 0 8 "{0}";
      ABIN "auto" 1 0 "{1}";
      ABIN "auto" 2 0 "{2}";
      ABIN "auto" 3 0 "{3}";
      ABIN "auto" 4 6 "{4}";
      ABIN "auto" 5 0 "{5}";
      ABIN "auto" 6 0 "{6}";
      ABIN "auto" 7 0 "{7}";
      ABIN "auto" 8 9 "{8}";
      ABIN "auto" 9 0 "{9}";
      ABIN "auto" 10 0 "{10}";
      ABIN "auto" 11 0 "{11}";
      ABIN "auto" 12 9 "{12}";
      ABIN "auto" 13 0 "{13}";
      ABIN "auto" 14 0 "{14}";
      ABIN "auto" 15 0 "{15}";
ENDCOVERPOINT
```

例えば、以下の様にビンの定義を追加すれば、100%カバレッジを達成する事が出来ます。

Copyright 2019 © Artgraphics. All rights reserved.

```
class simple_data_t;
rand bit [3:0]    data;

      function new;
            cg = new;
      endfunction
      constraint C { data[1:0] == 2'b00; }
      covergroup cg;
            coverpoint data
            {
                  bins value[] = { 0, 4, 8, 12 };
            }
      endgroup
endclass
```

Copyright 2019 © Artgraphics. All rights reserved.

4 クロス・カバレッジ

4.1 シンタックス

以下の様なシンタックスを持ちます（[1]）。非常に複雑なシンタックスなので、詳細な解説を避けて、例を説明します。

```
cover_cross ::=
        [ cross_identifier : ] cross list_of_cross_items [ iff ( expression ) ]
                cross_body
list_of_cross_items ::= cross_item , cross_item { , cross_item }
cross_item ::=
        cover_point_identifier
        | variable_identifier
cross_body ::=
        { { cross_body_item ; } }
        | ;
cross_body_item ::=
        function_declaraton
        | bins_selection_or_option ;
bins_selection_or_option ::=
        { attribute_instance } coverage_option
        | { attribute_instance } bins_selection
bins_selection ::= bins_keyword bin_identifier = select_expression [ iff
( expression ) ]
select_expression ::=
        select_condition
        | ! select_condition
        | select_expression && select_expression
        | select_expression || select_expression
        | ( select_expression )
        | select_expression with ( with_covergroup_expression ) [ matches
integer_covergroup_expression ]
        | cross_identifier
        | cross_set_expression [ matches integer_covergroup_expression ]
select_condition ::= binsof ( bins_expression ) [ intersect
{ covergroup_range_list } ]
bins_expression ::=
variable_identifier
        | cover_point_identifier [ . bin_identifier ]
covergroup_range_list ::= covergroup_value_range { , covergroup_value_range }
covergroup_value_range ::=
covergroup_expression
        | [ covergroup_expression : covergroup_expression ]25
with_covergroup_expression ::= covergroup_expression26
integer_covergroup_expression ::= covergroup_expression
cross_set_expression ::= covergroup_expression
```

4.2 クロス・カバレッジの記述

クロス・カバレッジはカバー・ポイントの組のカバレッジです。組になるカバー・ポイントの数に制限はありません。組（対）である為、ビンの数は大きくなる傾向があります。例えば、次の定義に於いて、

```
class sample_t;
bit [2:0]    a, b;
        covergroup cg;
                aBYb: cross a,b;
        endgroup
function new;
        cg = new;
endfunction
endclass
```


Copyright 2019 © Artgraphics. All rights reserved.

カバー・ポイント a 及び b のビン数はそれぞれ 8 ですが、クロス・カバレッジ aBYb のビン数は 64 になります。

クロス・カバレッジのシンタックスは多少複雑ですが、一般形式は以下の様になります。

```
[ cross_identifier : ]
      cross list_of_cross_items [ iff ( expression ) ] cross_body
cross_body ::= { { cross_body_item ; } } | ;
```

cross_body でクロス・カバレッジの定義をします。定義を省略すると、可能な全ての組み合わせの対が構成されます。cross_body のシンタックスは複雑ですが、簡単な定義法もあります。

binsof を使用すると比較的簡単にビンを定義する事が出来ます。binsof のシンタックスは以下の様になっています。

```
select_condition ::=
  binsof ( bins_expression ) [ intersect { covergroup_range_list } ]
```

例えば、次の様に定義します。

```
covergroup cov;
      coverpoint i { bins i[] = { [10:11] }; }
      coverpoint j { bins j[] = { [10:11] }; }
      x1: cross i, j;
      x2: cross i, j {
            bins i_ten = binsof(i) intersect {10};
      }
endgroup
```

クロス・プロダクト x1 は次の 4 つのビンから構成されます。

```
<i[0],j[0]>
<i[0],j[1]>
<i[1],j[0]>
<i[1],j[1]>
```

ここで、i[0] ::= {10}、i[1] ::= {11}です。

ビン i_ten は i==10 を含む i のビンから次の様に構成されます。

```
<i[0],j[0]>
<i[0],j[1]>
```

この他に自動的にビンを定義するので、x2 は次の 3 つのビンから構成されます。

```
i_ten
<i[1],j[0]>
<i[1],j[1]>
```


Copyright 2019 © Artgraphics. All rights reserved.

4.3 簡単な使用例

簡単な使用例を紹介します。

例 4-1

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

      function new();
            cg = new;
      endfunction

      covergroup cg;
            coverpoint        a;
            coverpoint        b;
            aBYb: cross a,b;
      endgroup
endclass

module test;
Sample      sample;

      initial begin
            sample = new;

            for( int i = 0; i < 8; i++ ) begin
                  sample.randomize();
                  sample.cg.sample();
            end
      end
endmodule
```

一般的に、制約を付けてビンを定義しないとクロス・カバレッジのカバレッジは低くなります。

```
CROSS "aBYb";
      CROSSITEM "a" "b";
      COVERAGE 12.50 8 64;
      GOAL 100;
      WEIGHT 1;
      COMMENT "";
      ATLEAST 1;
      CBIN "auto<auto[0],auto[3]>" 1;
      CBIN "auto<auto[0],auto[5]>" 1;
      CBIN "auto<auto[2],auto[5]>" 1;
      CBIN "auto<auto[5],auto[1]>" 1;
      CBIN "auto<auto[6],auto[5]>" 1;
      CBIN "auto<auto[6],auto[6]>" 1;
      CBIN "auto<auto[6],auto[7]>" 1;
      CBIN "auto<auto[7],auto[6]>" 1;
ENDCROSS
```

Copyright 2019 © Artgraphics. All rights reserved.

5　カバレッジ・オプション

カバレッジ・オプションはカバレッジ収集を制御する為の機能です。例えば、auto ビン数の標準値は 64 ですが、次の様にして変更する事が出来ます。

```
option.auto_bin_max = 4;
```

オプションを適切に設定する事により不必要な計測を回避する事が出来ます。同時に、カバレッジ率も向上します。例えば、32 ビットの変数のカバレッジを計測する際、カバレッジが 100%になる為には多くの時間が必要になります。一方、上記の設定変更をすると、32 ビット整数空間を 4 分割する為、瞬時に 100%カバレッジを達成する事ができます。ビン定義とオプション設定を組み合わせて使用すると適切なカバレッジを行なう事が出来ます。

オプションには covergroup のインスタンス毎に適用するオプションと、同じ covergroup のインスタンス全体に適用するオプションがあります。後者のオプションを type_option と呼びます。これは、クラスの static 属性と同じ概念です。オプションの詳細は、原書（[1]）を参照して下さい。

例 5-1

```
class Sample;
rand bit [7:0]     a, b;

      function new;
            cg = new;
      endfunction
      covergroup cg;
            a: coverpoint a { option.auto_bin_max = 4; }
            b: coverpoint b;
      endgroup
endclass

module test;
Sample      sample;

      initial begin
            sample = new;

            for( int i = 0; i < 16; i++ ) begin
                  sample.randomize();
                  sample.cg.sample();
            end
      end
endmodule
```

coverpoint a には auto ビンが 4 個しか作られません。一方、coverpoint b は 64 個の auto ビンを持ちます。それぞれのビンが計測に使用されます。明らかに、coverpoint a の方がカバレッジ率が高くなります。coverpoint a のカバレッジは以下の様になります。100%カバレッジを達成している事を確認する事が出来ます。

Copyright 2019 © Artgraphics. All rights reserved.

```
COVERPOINT "a";
      COVERAGE 100.00 4 4;
      GOAL 100;
      WEIGHT 1;
      COMMENT "";
      ATLEAST 1;
      ABIN "auto" 0 4 "{[0:63]}";
      ABIN "auto" 1 3 "{[64:127]}";
      ABIN "auto" 2 5 "{[128:191]}";
      ABIN "auto" 3 4 "{[192:255]}";
ENDCOVERPOINT
```

一方、coverpoint b のカバレッジは以下の様になります。カバレッジは僅か 23.44% です。

```
COVERPOINT "b";
      COVERAGE 23.44 15 64;
      GOAL 100;
      WEIGHT 1;
      COMMENT "";
      ATLEAST 1;
      ABIN "auto" 0 0 "{[0:3]}";
      ABIN "auto" 1 0 "{[4:7]}";
      ABIN "auto" 2 0 "{[8:11]}";
      ABIN "auto" 3 0 "{[12:15]}";
      --- 省略 ---
      ABIN "auto" 60 0 "{[240:243]}";
      ABIN "auto" 61 0 "{[244:247]}";
      ABIN "auto" 62 0 "{[248:251]}";
      ABIN "auto" 63 2 "{[252:255]}";
ENDCOVERPOINT
```

Copyright 2019 © Artgraphics. All rights reserved.

6 補足

6.1 シミュレーション領域（simulation regions）

時刻 T に於いてシミュレーションが開始されると、プロセス（あるいはスレッド）は、時系列的に分類された領域（regions）で実行します。ある領域のプロセスは他の領域のプロセスよりも先に実行します。この実行順序制約により、状態が安定した信号値を基にしてシミュレーションの論理が確立します。例えば、`a = b;` が実行する領域は `q <= d;` が実行する領域よりも早く実行します。その他の例として、program で記述されたテストベンチの論理は DUT の信号値が安定した状態になってから実行します。

SystemVerilog では以下の様に領域を分類しています。

① Preponed
② Pre-Active
③ Active
④ Inactive
⑤ Pre-NBA
⑥ NBA
⑦ Post-NBA
⑧ Pre-Observed
⑨ Observed
⑩ Post-Observed
⑪ Reactive
⑫ Re-Inactive
⑬ Pre-Re-NBA
⑭ Re-NBA
⑮ Post-Re-NBA
⑯ Pre-Postponed
⑰ Postponed

領域はこの順序で実行します。これらの全ての領域を理解する事が必要ですが、入門編の範囲を超える事は確かです。この節では、これらの領域の中の一部について記述します。

時系列的な領域の実行順序を原書から参照して図示しました。但し、明確さの為に一部を除外してあります。図の右側には、代表的な SystemVerilog 命令が実行する時期を明記してあります。

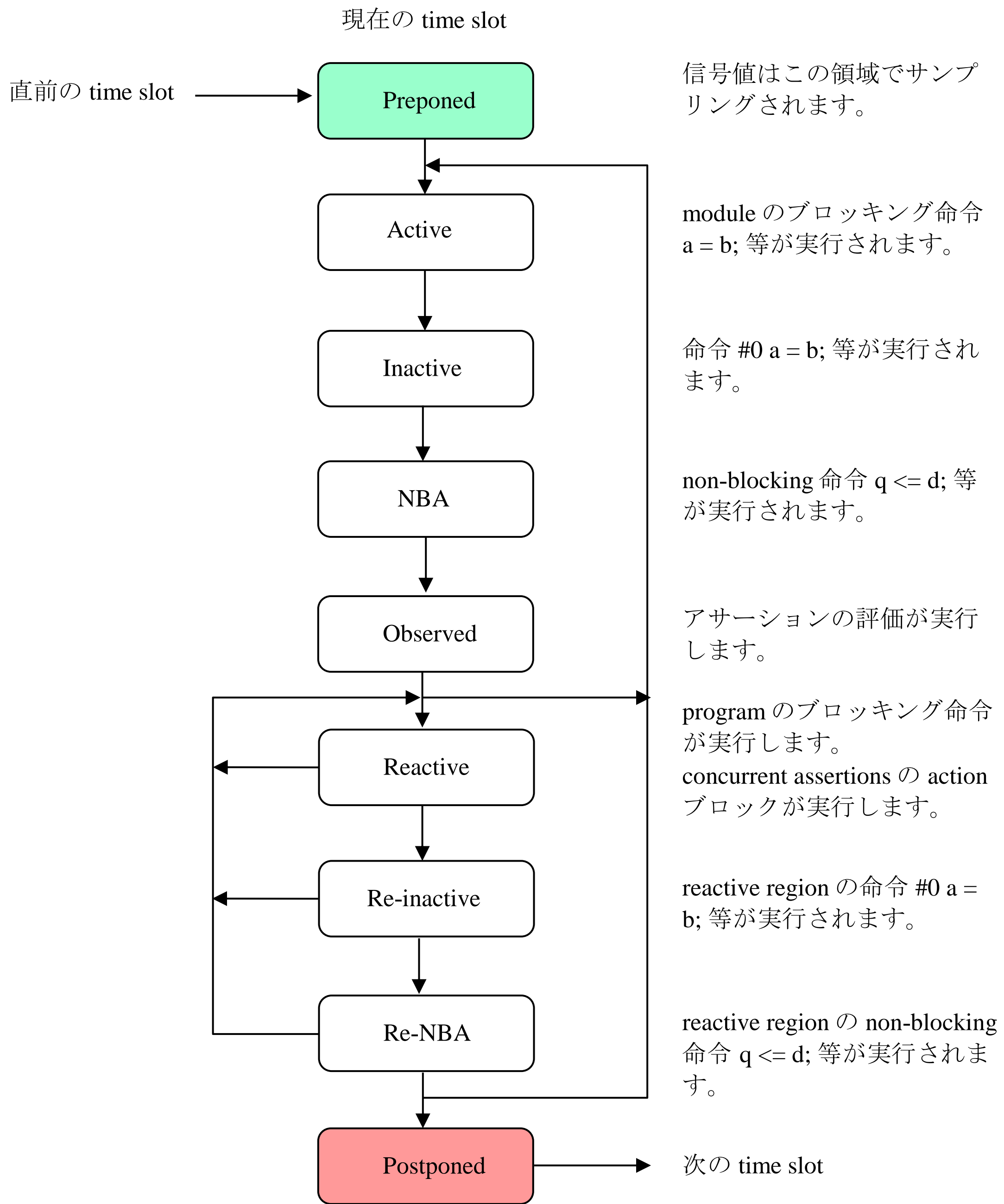
領域は類推し易くなっています。例えば、Re-Inactive 領域が存在しますが、それは Inactive 領域の対です。従って、Inactive 領域を見て下さい。それを見れば、Re-Inactive 領域では program で記述された

```
#0 a = b;
```

が実行する領域である事が分かります。

原書には全ての領域が記述されている為、理解する為には複雑過ぎると思えます。その為に、簡略化した図を準備しました。

Copyright 2019 © Artgraphics. All rights reserved.



Copyright 2019 © Artgraphics. All rights reserved.

7　参考文献

以下の文献を学習する事を薦めます。

[1] IEEE Std 1800-2017: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.
[2] Chris Spear: SystemVerilog for Verification, 2nd Edition, Springer 2008.

Copyright 2019 © Artgraphics. All rights reserved.