20.1 Introduction

Functional verification comprises a large portion of the resources required to design and validate a complex system. Often, the validation must be comprehensive without redundant effort. To minimize wasted effort, coverage is used as a guide for directing verification resources by identifying tested and untested portions of the design.

Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project in order to reduce the number of simulation cycles spent in verifying a design.

Broadly speaking, there are two types of coverage metrics. Those that can be automatically extracted from the design code, such as code coverage, and those that are user-specified in order to tie the verification environment to the design intent or functionality. This latter form is referred to as Functional Coverage, and is the topic of this section.

Functional coverage is a user-defined metric that measures how much of the design specification, as enumerated by features in the test plan, has been exercised. It can be used to measure whether interesting scenarios, corner cases, specification invariants, or other applicable design conditions—captured as features of the test plan—have been observed, validated and tested.

The key aspects of functional coverage are:

- It is user-specified, and is not automatically inferred from the design
- It is based on the design specification (i.e., its intent) and is thus independent of the actual design code or its structure.

Since it is fully specified by the user, functional coverage requires more upfront effort (someone has to write the coverage model). Functional coverage also requires a more structured approach to verification. Although functional coverage can shorten the overall verification effort and yield higher quality designs, these shortcomings can impede its adoption.

The SystemVerilog functional coverage extensions address these shortcomings by providing language constructs for easy specification of functional coverage models. This specification can be efficiently executed by the SystemVerilog simulation engine, thus, enabling coverage data manipulation and analysis tools that speed up the development of high quality tests. The improved set of tests can exercise more corner cases and required scenarios, without redundant work.

The SystemVerilog functional coverage constructs enable:

- Coverage of variables and expressions, as well as cross coverage between them.
- Automatic as well as user-defined coverage bins.
- Associate bins with sets of values, transitions, or cross products.
- Filtering conditions at multiple levels.
- Events and sequences to automatically trigger coverage sampling.
- Procedural activation and query of coverage.
- Optional directives to control and regulate coverage.
20.2 Defining the coverage model: covergroup

The covergroup construct encapsulates the specification of a coverage model. Each covergroup specification may include the following components:

- A clocking event that synchronizes the sampling of coverage points
- A set of coverage points
- Cross coverage between coverage points
- Optional formal arguments
- Coverage options

The covergroup construct is a user-defined type. The type definition is written once, and multiple instances of that type can be created in different contexts. Similar to a class, once defined, a covergroup instance can be created via the new() operator. A covergroup may be defined in a module, program, interface, or class.

```
covergroup_declaration ::= covergroup covergroup_identifier [ ( list_of_task_proto ) ] [ clocking_event ];
                      { coverage_spec_or_option ; }
endgroup [ : covergroup_identifier ]

list_of_task_proto ::= task_proto_formal { , task_proto_formal }

coverage_spec_or_option ::= {attribute_instance} coverage_spec
                      | {attribute_instance} coverage_option

coverage_option ::= option.option_name = expression
               | type_option.option_name = expression

coverage_spec ::= cover_point
               | cover_cross

variable_decl_assignment ::= // covergroup creation
                          ...
                          | covergroup_identifier variable_identifier = new [ ( list_of_arguments ) ]

Syntax 20-1—Covergroup syntax (excerpt from Annex A)
```

The identifier associated with the covergroup declaration defines the name of the coverage model. Using this name, an arbitrary number of coverage model instances can be created. For example:

```
covergroup cg; ... endgroup;
cg cg_inst = new;
```

The above example defines a covergroup named cg. An instance of cg is declared as cg_inst and created using the new operator.

A covergroup may specify an optional list of arguments. When the covergroup specifies a list of formal arguments, its instances must provide to the new operator all the actual arguments that are not defaulted. Actual arguments are evaluated when the new operator is executed. A ref argument allows a different variable to be sampled by each instance of a covergroup. Input arguments will not track value of their arguments; they will use the value passed to the new operator.
If a clocking event is specified, it defines the event at which coverage points are sampled. If the clocking event
is omitted, users must procedurally trigger the coverage sampling. This is done via a built-in method (see Sec-
tion 20.7). Optionally, the strobe option can be used to modify the sampling behavior. When the strobe option
is not set (the default), a coverage point is sampled as soon as the clocking event takes place. If the clocking
event occurs multiple times in a time step, the coverage point will also be sampled multiple times. The strobe
option (see Section 20.6.1) can be used to specify that coverage points are sampled at the end of the time slot,
thereby filtering multiple clocking events so that only sample per time slot is taken.

A covergroup may contain one or more coverage points. A coverage point can be a variable or an expression.
Each coverage point includes a set of bins associated with its sampled values or its value-transitions. The bins
may be explicitly defined by the user or automatically created by the tool. Coverage points are discussed in
detail in Section 20.4.

```systemverilog
enum { red, green, blue } color;

covergroup g1 @(posedge clk);
  c: coverpoint color;
endgroup;
```

The above example defines coverage group g1 with a single coverage point associated with variable color. The
value of the variable color is sampled at the indicated clocking event: the positive edge of signal clk. Since the
coverage point does not explicitly define any bins, the tool automatically creates 3 bins, one for each possible
value of the enumerated type. Automatic bins are described in Section 20.4.2.

A coverage group may also specify cross coverage between two or more coverage points or variables. Any
combination of more than two variables or previously declared coverage points is allowed. For example:

```systemverilog
enum { red, green, blue } color;
bit [3:0] pixel_adr, pixel_offset, pixel_hue;

covergroup g2 @(posedge clk);
  Hue: coverpoint pixel_hue;
  Offset: coverpoint pixel_offset;
  AxC: cross color, pixel_adr; // cross 2 variables (implicitly declared coverpoints)
  all: cross color, Hue, Offset; // cross 1 variable and 2 coverpoints
endgroup;
```

The example above creates coverage group g2 that includes 2 coverage points and two cross coverage items.
Explicit coverage points labeled Offset and Hue are defined for variables pixel_offset and pixel_hue. System-
Verilog implicitly declares coverage points for variables color and pixel_adr in order to track their cross cov-
erage. Implicitly declared cover points are described in Section 20.5.

A coverage group may also specify one or more options to control and regulate how coverage data is struc-
tured and collected. Coverage options may be specified for the coverage group as a whole, or for specific
items within the coverage group, that is, any of its coverage points or crosses. In general, a coverage option
specified at the covergroup level applies to all of its items unless overridden by them. Coverage options are
described in Section 20.6.

### 20.3 Using covergroup in classes

By embedding a coverage group within a class definition, the covergroup provides a simple way to cover a
subset of the class properties. This integration of coverage with classes provides an intuitive and expressive
mechanism for defining the coverage model associated with a class. For example,

In class xyz, defined below, members m_x and m_y are covered using an embedded covergroup:

```systemverilog
In class xyz, defined below, members m_x and m_y are covered using an embedded covergroup:
```
class xyz;
    bit [3:0] m_x;
    int m_y;
    bit m_z;

    covergroup cv1 @m_z;    // embedded covergroup
        coverpoint m_x;
        coverpoint m_y;
    endgroup

    function new(); cv1 = new; endfunction
endclass

In this example, data members m_x and m_y of class xyz are sampled on every change of data member m_z.

When a covergroup is defined within a class, and no explicit instances of that covergroup are declared in the class then an instance with the same name as the coverage group is implicitly defined, e.g. in the above example, an instance named cv1 of the embedded coverage group is implicitly defined. Whether the coverage group variable is implicitly or explicitly declared, each class contains exactly one instance of each embedded coverage group. Each embedded coverage group thus becomes part of the class, tightly binding the class properties to the coverage definition. Declaring multiple instances of an embedded coverage group shall result in a compiler error.

An embedded covergroup can define a coverage model for protected and local class properties without any changes to the class data encapsulation. Class members can become coverage points or can be used in other coverage constructs, such as conditional guards or option initialization.

A class can have more than one covergroup. The following example shows two cover groups in class MC.

class MC;
        logic [3:0] m_x;
        local logic m_z;
        bit m_e;
        covergroup cv1 @(posedge clk); coverpoint m_x; endgroup
        covergroup cv2 @m_e ; coverpoint m_z; endgroup
endclass

In covergroup cv1, public class member variable m_x is sampled at every positive edge of signal clk. Local class member m_z is covered by another covergroup cv2. Each coverage groups is sampled by a different clocking event.

An embedded coverage group must be explicitly instantiated in the new method. If it is not then the coverage group is not created and no data will be sampled.

Below is an example of an embedded coverage_group that does not have any passed-in arguments, and uses explicit instantiation to synchronize with another object:

class Helper;
    int m_ev;
endclass

class MyClass;
        Helper m_obj;
        int m_a;

covergroup Cov @(m_obj.m_ev);
  coverpoint m_a;
endgroup

function new();
  m_obj = new;

  Cov = new; // Create embedded covergroup after creating m_obj
endfunction
endclass

In this example, covergroup Cov is embedded within class MyClass, which contains an object of type Helper class, called m_obj. The clocking event for the embedded coverage group refers to data member m_ev of m_obj. Because the coverage group Cov uses m_obj, m_obj must be instantiated before Cov. Therefore, the coverage group Cov is instantiated after instantiating m_obj in the class constructor. As shown above, the instantiation of an embedded coverage group is done by assigning the result of the new operator to the coverage group identifier.

The following example shows how arguments passed in to an embedded coverage group can be used to set a coverage option of the coverage group.

class C1;
  bit [7:0] x;

  covergroup cv (int arg) @ (posedge clk);
    option.at_least = arg;
    coverpoint x;
  endgroup

  function new(int p1);
    cv = new(p1);
  endfunction
endclass

initial begin
  C1 obj = new(4);
end

20.4 Defining coverage points

A covergroup may contain one or more coverage points. A coverage point can be an integral variable or an integral expression. Each coverage point includes a set of bins associated with its sampled values or its value-transitions. The bins may be explicitly defined by the user or automatically created by SystemVerilog. The syntax for specifying coverage points is given below.

```plaintext
cover_point ::= [label :] 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 [ [expression]] = { range_list } [iff (expression)]
```
A coverage point may be optionally labeled. If the label is specified then it designates the name of the coverage point. This name may be used to add this coverage point to a cross coverage specification, or to access the methods of the coverage point. If the label is omitted and the coverage point is associated with a single variable then the variable name becomes the name of the coverage point. Otherwise, an implementation may generate a name for the coverage point only for the purposes of coverage reporting, that is, generated names may not be used within the language.

A coverage point may sample the values that correspond to a particular scheduling region (see Section 14) by specifying a clocking block signal. Thus, a coverage point that denotes a clocking block signal will sample the values made available by the clocking block. If the clocking block specifies a skew of #1step, the coverage point will sample the signal values from the Preponed region. If the clocking block specifies a skew of #0, the coverage point will sample the signal values from the Observe region.

The expression within the `iff` construct specifies an optional condition that disables coverage for that cover point. If the guard expression evaluates to false at a sampling point, the coverage point is ignored. For example:

```verbatim
covergroup g4;
  coverpoint s0 iff(!reset);
endgroup
```

In the preceding example, cover point `s0` is covered only if the value `reset` is false.

A coverage-point bin associates a name and a count with a set of values or a sequence of value transitions. If the bin designates a set of values, the count is incremented every time the coverage point matches one of the values in the set. If the bin designates a sequence of value transitions, the count is incremented every time the coverage point matches the entire sequence of value transitions.

The bins for a coverage point can be automatically created by SystemVerilog or explicitly defined using the `bins` construct to name each bin. If the bins are not explicitly defined, they are automatically created by SystemVerilog. The number of automatically created bins can be controlled using the `auto_bin_max` coverage option. Coverage options are described in Section 20.6.

The `bins` construct allows creating a separate bin for each value in the given range-list, or a single bin for the entire range of values. To create a separate bin for each value (an array of bins) the square brackets, [], must follow the bin name. To create a fixed number of bins for a set of values, a number may be specified inside the square brackets. The `range_list` used to specify the set of values associated with a bin shall be constant expressions, instance constants (for classes only) or non-ref arguments to the coverage group.

If a fixed number of bins is specified, and that number is smaller than the number of values in the bin then the possible bin values are uniformly distributed among the specified bins. If the number of values is not divisible by the number of bins then the last bin will include the remaining items. For example:

```verbatim
bins fixed [3] = {1:10};
```
The 11 possible values are distributed as follows: <1,2,3>, <4,5,6>, <7,8,9,10>. If the number of bins exceeds the number of values then some of the bins will be empty.

The expression within the `iff` construct at the end of a bin definition provides a per-bin guard condition. If the expression is false at a sampling point, the count for the bin is not incremented.

The `default` specification defines a bin that is associated with none of the defined value bins. The `default` bin catches the values of the coverage point that do not lie within any of the defined bins. However, the coverage calculation for a coverage point shall not take into account the coverage captured by the `default` bin. The `default` is useful for catching unplanned or invalid values. The `default` specification does not accept multiple transition bins (the `[]` notation is not allowed).

```c
int v_a;

covergroup cg @ (posedge clk);

coverpoint v_a {
    bins a = { [0:63], 65 };
    bins b[] = { [127:150], [148:191] }; // note overlapping values
    bins c[] = { 200, 201, 202 };
    bins others[] = default;
}
endgroup
```

In the example above, the first `bins` construct associates bin `a` with the values of variable `v_a` between 0 and 63, and the value 65. The second `bins` construct creates a set of 65 bins `b[127]`, `b[128]`, ..., `b[191]`. Likewise, the last `bins` construct creates 3 bins: `c[200]`, `c[201]`, and `c[202]`. Every value that does not match bins `a`, `b[]`, or `c[]` is added into its own distinct bin.

A `default` or `default sequence` bin specification cannot be explicitly ignored (see Section 20.4.4). It shall be an error for bins designated as `ignore_bins` to also specify a `default` or `default sequence`.

Generic coverage groups can be written by passing their traits as arguments to the constructor. For example:

```c
covergroup gc (ref int ra, int low, int high ) @ (posedge clk);

coverpoint ra // sample variable passed by reference
{
    bins good = { [low : high] };
    bins bad[] = default;
}
endgroup
```

```c
int va, vb;

cg c1 = new ( va, 0, 50 ); // cover variable va in the range 0 to 50

cg c2 = new ( vb, 120, 600 ); // cover variable vb in the range 120 to 600
```

The example above defines a coverage group, `gc`, in which the signal to be sampled as well as the extent of the coverage bins are specified as arguments. Later, two instances of the coverage group are created; each instance samples a different signal and covers a different range of values.

### 20.4.1 Specifying bins for transitions

The syntax for specifying transition bins accepts a subset of the sequence syntax described in Section 17:
bins_or_options ::= 
  ... 
  | [wildcard] bins_keyword bin_identifier [[ ] ] = ( trans_list ) [iff ( expression ) ]

bins_keyword ::= bins | illegal_bins | ignore_bins

trans_list ::= trans_set { , trans_set }

trans_set ::= trans_range_list -> trans_range_list { -> trans_range_list }

trans_range_list ::= trans_item

  | trans_item [ [* value_range ] ] // consecutive repetition 
  | trans_item [ [*-> value_range ] ] // goto repetition 
  | trans_item [ [*= value_range ] ] // non-consecutive repetition

trans_item ::= { range_list } | value_range

Syntax 20-3 — Transition bin syntax (excerpt from Annex A)

A trans_list specifies one or more sets of ordered value transitions of the coverage point. A single value transition is thus specified as:
  value1 -> value2
It represents the value of coverage point at two successive sample points, that is, value1 followed by value2 at the next sample point.

A sequence of transitions is represented as:
  value1 -> value3 -> value4 -> value5
In this case, value1 is followed by value3, followed by value4 and followed by value5. A sequence may be of any arbitrary length.

A set of transitions can be specified as:
  {range_list1} -> {range_list2}
This specification expands to transitions between each value in range_list1 and each value in range_list2. For example,
  {1,5} -> {6, 7}
specifies the following four transitions:
  1->6, 1->7, 5->6, 5->7

Consecutive repetitions of transitions are specified using (see Annex G):
  trans_item [* value_range ]
Here, trans_item is repeated for value_range times. For example,
  3 [* 5]
is same as
  3->3->3->3->3

An example of a range of repetition is
  3 [* 3:5]
is same as
  3->3->3, 3->3->3->3, 3->3->3->3->3

The repetition with non-consecutive occurrence of a value is specified using: trans_item [*-> value_range ]. Here, the occurrence of a value is specified with an arbitrary number of sample points where the value does not occur. For example,
  3 [*-> 3]
is the same as
  ...3->...->3...->3
where the dots ( . . . ) represent any transition that does not contain the value 3.
Non-consecutive repetition is where a sequence of transitions continues until the next transition. For example,
\[3 \ast = 2\]
is same as the transitions below excluding the last transition.
\[3 \rightarrow \ldots \rightarrow 3 \ldots \rightarrow 3\]

A \textit{trans_list} specifies one or more sets of ordered value transitions of the coverage point. If the sequence of value transitions of the coverage point matches any complete sequence in the \textit{trans_list}, the coverage count of the corresponding bin is incremented. For example:

```verilog
bit [4:1] v_a;
covergroup cg (@posedge clk);
coverpoint v_a {
    bins sa = (4 \rightarrow 5 \rightarrow 6, {{7:9},10} \rightarrow {11,12});
    bins sb[] = (4 \rightarrow 5 \rightarrow 6, {{7:9},10} \rightarrow {11,12});
}
endgroup
```

The example above defines two transition coverage bins. The first \texttt{bins} construct associates the following sequences with bin \texttt{sa}: 4\rightarrow5\rightarrow6, or 7\rightarrow11, 8\rightarrow11, 9\rightarrow11, 10\rightarrow11, 7\rightarrow12, 8\rightarrow12, 9\rightarrow12, 10\rightarrow12. The second \texttt{bins} construct associates an individual bin with each of the above sequences: \texttt{sb[4\rightarrow5\rightarrow6]}, \ldots, \texttt{sb[10\rightarrow12]}.

Transitions that specify sequences of unbounded or undetermined varying length cannot be used with the multiple bins construct (the \texttt{[]} notation). For example, the length of the transition: \texttt{3\ast = 2}, which uses non-consecutive repetition, is unbounded and can vary during simulation. An attempt to specify multiple bins with such sequences shall result in an error.

### 20.4.2 Automatic bin creation for coverage points

If a coverage point does not define any bins, SystemVerilog automatically creates state bins. This provides an easy-to-use mechanism for binning different values of a coverage point. Users can either let the tool automatically create state bins for coverage points or explicitly define named bins for each coverage point.

When the automatic bin creation mechanism is used, SystemVerilog creates \(N\) bins to collect the sampled values of a coverage point. The value \(N\) is determined as follows:
- For an \texttt{enum} coverage point, \(N\) is the cardinality of the enumeration.
- For any other integral coverage point, \(N\) is the minimum of \(2^M\) and the value of the \texttt{auto_bin_max} option, where \(M\) is the number of bits needed to represent the coverage point.

If the number of automatic bins is smaller than the number of possible values (\(N < 2^M\)) then the \(2^M\) values are uniformly distributed in the \(N\) bins. If the number of values, \(2^M\), is not divisible by \(N\), then the last bin will include the additional (up to \(N-1\)) remaining items. For example, if \(M\) is 3, and \(N\) is 3 then the 8 possible values are distributed as follows: \(<0:1>, <2:3>, <4,5,6,7>\).

Automatically created bins only consider 2-state values; sampled values containing \texttt{X} or \texttt{Z} are excluded.

SystemVerilog implementations may impose a limit on the number of automatic bins. See the Section 20.6 for the default value of \texttt{auto_bin_max}.

Each automatically created bin will have a name of the form: \texttt{auto[value]}, where value is either a single coverage point value, or the range of coverage point values included in the bin — in the form \texttt{low:high}. For enumerated types, value is the label associated with a particular enumerated value.

### 20.4.3 Wildcard specification of coverage point bins
By default, a value or transition bin definition can specify 4-state values. When a bin definition includes an X or Z, it indicates that the bin count should only be incremented when the sampled value has an X or Z in the same bit positions, i.e., the comparison is done using ===. The wildcard bins definition causes all X, Z, or ? to be treated as wildcards for 0 or 1 (similar to the =?= operator). For example:

```
wildcard bins g12_16 = { 4'b11?? };
```

The count of bin g12_16 is incremented when the sampled variable is between 12 and 16:

```
1100 1101 1110 1111
```

Similarly, transition bins can define wildcard bins. For example:

```
wildcard bins T0_3 = (2'b0x -> 2'b1x);
```

The count of transition bin T0_3 is incremented for transitions (as if by {0,1}->{2,3}):

```
00 -> 10 00 -> 11 01 -> 10 01 -> 11
```

A wildcard bin definition only consider 2-state values; sampled values containing X or Z are excluded. Thus, the range of values covered by a wildcard bin is established by replacing every wildcard digit by 0 to compute the low bound and 1 to compute the high bound.

### 20.4.4 Excluding coverage point values or transitions

A set of values or transitions associated with a coverage-point can be explicitly excluded from coverage by specifying them as ignore_bins. For example:

```
covergroup cg23;
    coverpoint a
    {
        ignore_bins ignore_vals = {7,8};
        ignore_bins ignore_trans = (1->3->5);
    }
endgroup
```

All values or transitions associated with ignored bins are excluded from coverage. Ignored values or transitions are excluded even if they are also included in another bin.

### 20.4.5 Specifying Illegal coverage point values or transitions

A set of values or transitions associated with a coverage-point can be marked as illegal by specifying them as illegal_bins. For example:

```
covergroup cg3;
    coverpoint b
    {
        illegal_bins bad_vals = {1,2,3};
        illegal_bins bad_trans = {4->5->6};
    }
endgroup
```

All values or transitions associated with illegal bins are excluded from coverage. If they occur, a run-time error is issued. Illegal bins take precedence over any other bins, that is, they will result in a run-time error even if they are also included in another bin.

### 20.5 Defining cross coverage


A coverage group may specify cross coverage between two or more coverage points or variables. Cross coverage is specified using the cross construct. When a variable \( V \) is part of a cross coverage, SystemVerilog implicitly creates a coverage point for the variable, as if it had been created by the statement "coverpoint \( V \);". Thus, a cross involves only coverage points. Expressions may not be used directly in a cross; a coverage point must be explicitly defined first.

The syntax for specifying cross coverage is given below.

```plaintext
cover_cross ::= [label:] cross list_of_coverpoints [ iff ( expression ) ] select_bins_or_empty

list_of_coverpoints ::= cross_item , cross_item { , cross_item }

cross_item ::= cover_point_identifier
  | variable_identifier

select_bins_or_empty ::= { { bin_selection_or_option ; } }
  |

bin_selection_or_option ::= {attribute_instance} coverage_option
  | {attribute_instance} bins_selection

bins_selection ::= bins_keyword bin_identifier = select_expression

select_expression ::= select_condition
  | ! select_condition
  | select_expression && select_expression
  | select_expression || select_expression
  | ( select_expression )

select_condition ::= binsof ( bins_expression ) [ , intersect open_range_list ]

bins_expression ::= variable_identifier
  | cover_point_identifier[ , bins_identifier ]

open_range_list ::= { open_value_range , open_value_range }

open_value_range ::= expression
  | [ expression : expression ]
  | [ expression : $ ]
  | [$ : expression ]
```

Syntax 20-4 —Cross coverage syntax (excerpt from Annex A)

The label for a cross declaration provides an optional name. The label also creates a hierarchical scope for the bins defined within the cross.

The expression within the optional iff provides a conditional guard for the cross coverage. If at any sample point, the condition evaluates to false, the cross coverage is ignored.
Cross coverage of a set of N coverage points is defined as the coverage of all combinations of all bins associated with the N coverage points, that is, the Cartesian product of the N sets of coverage-point bins. For example:

```systemverilog
bit [3:0] a, b;

covergroup cov @ (posedge clk);
    aXb : cross a, b;
endgroup
```

The coverage group `cov` in the example above specifies the cross coverage of two 4-bit variables, `a` and `b`. SystemVerilog implicitly creates a coverage point for each variable. Each coverage point has 16 bins, namely `auto[0]...auto[15]`. The cross of `a` and `b` (labeled `aXb`), therefore, has 64 cross products, and each cross product is a bin of `aXb`.

Cross coverage between expressions previously defined as coverage points is also allowed. For example:

```systemverilog
bit [3:0] a, b, c;

covergroup cov2 @ (posedge clk);
    BC: coverpoint b+c;
    aXb : cross a, BC;
endgroup
```

The coverage group `cov2` has the same number of cross products as the previous example, but in this case, one of the coverage points is the expression `b+c`, which is labeled `BC`.

```systemverilog
bit [31:0] a_var;
bit [3:0] b_var;

covergroup cov3 @ (posedge clk);
    A: coverpoint a_var { bins yy[] = { [0:9] }; } 
    CC: cross b_var, A;
endgroup
```

The coverage group `cov3` crosses variable `b_var` with coverage point `A` (labeled `CC`). Variable `b_var` automatically creates 16 bins (`auto[0]...auto[15]`). Coverage point `A` explicitly creates 10 bins `yy[0]..yy[9]`. The cross of two coverage points creates `16 * 10 = 160` cross product bins, namely the pairs shown below:

```
<auto[0], yy[0]>
<auto[0], yy[1]>
...
<auto[0], yy[9]>
<auto[1], yy[0]>
...
<auto[15], yy[9]>
```

Cross coverage is allowed only between coverage points defined within the same coverage group. Coverage points defined in a coverage group other than the one enclosing the cross may not participate in a cross. Attempts to cross items from different coverage groups shall result in a compiler error.

In addition to specifying the coverage points that are crossed, SystemVerilog includes a powerful set of operators that allow defining cross coverage bins. Cross coverage bins can be specified in order to group together a set of cross products. A cross-coverage bin associates a name and a count with a set of cross products. The count of the bin is incremented every time any of the cross products match, i.e., every coverage point in the cross matches its corresponding bin in the cross product.

User-defined bins for cross coverage are defined using bins select-expressions. The syntax for defining these bin selection expressions is given in Syntax 20-4.
The `binsof` construct yields the bins of its expression, which can be either a coverage point (explicitly defined or implicitly defined for a single variable) or a coverage-point bin. The resulting bins can be further selected by including (or excluding) only the bins whose associated values intersect a desired set of values. The desired set of values can be specified using a comma-separated list of `open_value_range` as shown in Syntax 20-4. For example, the following select expression:

```plaintext
binsof( x ).intersect { y }
```

denotes the bins of coverage point x whose values intersect the range given by y. Its negated form:

```plaintext
! binsof( x ).intersect { y }
```

denotes the bins of coverage point x whose values do not intersect the range given by y.

The `open_value_range` syntax can specify a single value, a range of values, or an open range, which denotes the following:

- `[ $ : value ]`  => The set of values less than or equal to value
- `[ value : $ ]`  => The set of values greater or equal to value

The bins selected can be combined with other selected bins using the logical operators `&&` and `||`.

### 20.5.1 Example of user-defined cross coverage and select expressions

```plaintext
bit [7:0] v_a, v_b;

covergroup cg @ (posedge clk);

a: coverpoint v_a
{
    bins a1 = [0:63];
    bins a2 = [64:127];
    bins a3 = [128:191];
    bins a4 = [192:255];
}

b: coverpoint v_b
{
    bins b1 = {0};
    bins b2 = [1:84];
    bins b3 = [85:169];
    bins b4 = [170:255];
}

c : cross v_a, v_b
{
    bins c1 = ! binsof(a).intersect ([100:200]); // 4 cross products
    bins c2 = binsof(a.a2) || binsof(b.b2); // 7 cross products
    bins c3 = binsof(a.a1) && binsof(b.b4); // 1 cross product
}

digroup
```

The example above defines a coverage-group named cg that samples its cover-points on the positive edge of signal clk (not shown). The coverage-group includes two cover-points, one for each of the two 8-bit variables, v_a and v_b. The coverage-point labeled 'a' associated with variable v_a, defines four equal-sized bins for each possible value of variable v_a. Likewise, the coverage-point labeled 'b' associated with variable v_b, defines four bins for each possible value of variable v_b. The cross definition labeled 'c', specifies the cross coverage of the two cover-points v_a and v_b. If the cross coverage of cover-points a and b were defined without any additional cross-bins (select expressions) the then cross coverage of a and b would include 16 cross products corresponding to all permutations of bins a1 through a4 with bins b1 through b4, that is, cross products <a1, b1>, <a1,b2>, <a1,b3>, <a1,b4>... <a4, b1>, <a4,b2>, <a4,b3>, <a4,b4>.
The first user-defined cross bin, c1, specifies that all bin c1 should include only cross products of cover-point a that do not intersect the value range 100-200. This select expression excludes bins a2, a3, and a4. Thus, bin c1 will cover only four cross-products of <a1,b1>, <a1,b2>, <a1,b3>, and <a1,b4>.

The second user-defined cross bin, c2, specifies that bin c2 should include only cross products whose values are covered by bin a2 of cover-point a or cross products whose values are covered by bin b2 of cover-point b. This select expression includes the following 7 cross products: <a2, b1>, <a2,b2>, <a2,b3>, <a2,b4>, <a1, b2>, <a3,b2>, and <a4,b2>.

The final user-defined cross bin, c3, specifies that bin c3 should include only cross products whose values are covered by bin a1 of cover-point a and cross products whose values are covered by bin b4 of cover-point b. This select expression includes only one cross-product <a1, b4>.

When select expressions are specified on transition bins, the binsof operator uses the last value of the transition.

**20.5.2 Excluding cross products**

A group of bins can be excluded from coverage by specifying a select expression using ignore_bins. For example:

```vseq
covergroup yy;
cross a, b {
    ignore_bins foo = binsof(a).intersect { 5, [1:3] };
}
endgroup
```

All cross products that satisfy the select expression are excluded from coverage. Ignored cross products are excluded even if they are included in other cross-coverage bins of the enclosing cross.

**20.5.3 Specifying Illegal cross products**

A group of bins can be marked as illegal by specifying a select expression using illegal_bins. For example:

```vseq
covergroup zz(int bad);
cross x, y {
    illegal_bins foo = binsof(y).intersect {bad};
}
endgroup
```

All cross products that satisfy the select expression are excluded from coverage, and a run-time error is issued. Illegal cross products take precedence over any other cross products, that is, they will result in a run-time error even if they are also explicitly ignored (using an ignore_bins) or included in another cross bin.

**20.6 Specifying coverage options**

Options control the behavior of the a covergroup, coverpoint and cross. There are two types of options: those that are specific to an instance of a covergroup, and those that specify an option for the covergroup type as a whole.

The following table lists instance specific covergroup options and their description. Each instance of a covergroup can initialize an instance specific option to a different value. The initialize option value affects only that instance.
weight= number | 1 | If set at the covergroup syntactic level, it specifies the weight of this covergroup instance for computing the overall instance coverage of the simulation. If set at the coverpoint (or cross) syntactic level, it specifies the weight of a coverpoint (or cross) for computing the instance coverage of the enclosing covergroup.

goal=number | 90 | Specifies the target goal for a covergroup instance, or a coverpoint or a cross of an instance.

name=string | unique name | Specify a name for the covergroup instance. If unspecified, a unique name for each instance is automatically generated by the tool.

comment=string | ”” | A comment that appears with the instance of a covergroup, or a coverpoint or cross of the covergroup instance. The comment is saved in the coverage database and included in the coverage report.

at_least=number | 1 | Minimum number of hits for each bin. A bin with a hit count that is less than num is not considered covered.

detect_overlap=Boolean | 0 | When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a coverpoint.

auto_bin_max=number | 64 | Maximum number of automatically created bins when no bins are explicitly defined for a coverpoint.

cross_auto_bin_max=number | unbounded | Maximum number of automatically created cross product bins for a cross.

cross_num_print_missing=number | 0 | Number of missing (not covered) cross product bins that must be saved to the coverage database and printed in the coverage report.

per_instance=Boolean | 0 | Each instance contributes to the overall coverage information for the covergroup type. When true, coverage information for this covergroup instance is tracked as well.

Table 20-1: Instance specific coverage options

The instance specific options mentioned above can be set in the covergroup definition. The syntax for setting these options in the covergroup definition is:

```plaintext
option.option_name = expression ;
```

The identifier option is a built-in member of any coverage group.

An example is shown below.

```plaintext
covergroup gl (int w, string instComment) @(posedge clk) ;
   // track coverage information for each instance of gl in addition
   // to the cumulative coverage information for covergroup type gl
   option.per_instance = 1;

option.comment = instComment;   // comment for each instance of this covergroup

a : coverpoint a_var
{
   // Create 128 automatic bins for coverpoint “a” of each instance of gl
   option.auto_bin_max = 128;
}

b : coverpoint b_var;
{
   // This coverpoint contributes w times as much to the coverage of an instance of gl than
```
// coverpoints “a” and “c1”
option.weight = w;
}
c1 : cross a_var, b_var;
endgroup

Option assignment statements in the covergroup definition are evaluated at the time that the covergroup is instantiated. The per_instance option is instance constant. It can only be set in the covergroup definition. Other instance specific options can be set procedurally after a covergroup has been instantiated. The syntax is:

coverage_option_assignment ::= // Not in Annex A
instance_name.option.option_name = expression;

| instance_name.covergroup_item_identifier.option.option_name = expression;

Here is an example:

covergroup gc @(posedge clk) ;
a : coverpoint a_var;
b : coverpoint b_var;
endgroup

gc g1 = new;
g1.option.comment = “Here is a comment set for the instance g1”;
g1.a.option.weight = 3; // Set weight for coverpoint “a” of instance g1

The following table summarizes the syntactical level (covergroup, coverpoint, or cross) at which instance options can be specified. All instance options can be specified at the covergroup level. Except for the weight, goal, comment, and per_instance options, all other options set at the covergroup syntactic level act as a default value for the corresponding option of all coverpoint(s) and cross(es) in the covergroup. Individual coverpoint or crosses may overwrite these default values. When set at the covergroup level, the weight, goal, comment, and per_instance options do not act as default values to the lower syntactic levels.

<table>
<thead>
<tr>
<th>Option name</th>
<th>Allowed in Syntactic Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>covergroup</td>
</tr>
<tr>
<td>weight</td>
<td>Yes</td>
</tr>
<tr>
<td>goal</td>
<td>Yes</td>
</tr>
<tr>
<td>comment</td>
<td>Yes</td>
</tr>
<tr>
<td>at_least</td>
<td>Yes (default for coverpoints &amp; crosses)</td>
</tr>
<tr>
<td>detect_overlap</td>
<td>Yes (default for coverpoints)</td>
</tr>
<tr>
<td>auto_bin_max</td>
<td>Yes (default for coverpoints)</td>
</tr>
<tr>
<td>cross_auto_bin_max</td>
<td>Yes (default for crosses)</td>
</tr>
<tr>
<td>cross_num_print_missing</td>
<td>Yes (default for crosses)</td>
</tr>
<tr>
<td>per_instance</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 20-2: Coverage options per-syntactic level

20.6.1 Covergroup Type Options

The following table lists options that describe a particular feature (or property) of the covergroup type as a whole. They are analogous to static data members of classes.

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>

16
Table 20-3: Coverage group type (static) options

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight=constant_number</td>
<td>1</td>
<td>If set at the covergroup syntactic level, it specifies the weight of this covergroup for computing the overall cumulative (or type) coverage of the saved database. If set at the coverpoint (or cross) syntactic level, it specifies the weight of a coverpoint (or cross) for computing the cumulative (or type) coverage of the enclosing covergroup.</td>
</tr>
<tr>
<td>goal=constant_number</td>
<td>90</td>
<td>Specifies the target goal for a covergroup type, or a coverpoint or cross of a covergroup type.</td>
</tr>
<tr>
<td>comment=string_literal</td>
<td>&quot;&quot;</td>
<td>A comment that appears with the covergroup type, or a coverpoint or cross of the covergroup type. The comment is saved in the coverage database and included in the coverage report.</td>
</tr>
<tr>
<td>strobe=constant_number</td>
<td>0</td>
<td>If set to 1, all samples happen at the end of the time slot, like the $strobe system task.</td>
</tr>
</tbody>
</table>

The covergroup type options mentioned above can be set in the covergroup definition. The syntax for setting these options in the covergroup definition is:

```vhdl
    type_option.option_name = expression ;
```

The identifier type_option is a built-in member of any coverage group.

Different instances of a covergroup cannot assign different values to type options. This is syntactically disallowed, since these options can only be initialized via constant expressions. Here is an example:

```vhdl
    covergroup gl (int w, string instComment) @ (posedge clk) ;
    // track coverage information for each instance of gl in addition
    // to the cumulative coverage information for covergroup type gl
    option.per_instance = 1;
    type_option.comment = "Coverage model for features foo and bar";
    type_option.strobe = 1; // sample at the end of the time slot
    // comment for each instance of this covergroup
    option.comment = instComment;

    a : coverpoint a_var
    {
        // Use weight 2 to compute the coverage of each instance
        option.weight = 2;
        // Use weight 3 to compute the cumulative (type) coverage for gl
        type_option.weight = 3;
        // NOTE: type_option.weight = w would cause syntax error.
    }
    b : coverpoint b_var;
    {
        // Use weight w to compute the coverage of each instance
        option.weight = w;
        // Use weight 5 to compute the cumulative (type) coverage of gl
        type_option.weight = 5;
    }
endgroup
```

In the above example the coverage for each instance of gl is computed as:

$$(((\text{instance coverage of } "a") \times 2) + (\text{instance coverage of } "b") \times w) / (2 + w).$$

On the other hand the coverage for covergroup type “gl” is computed as:

$$(((\text{instance coverage of } "a") \times 2) + (\text{instance coverage of } "b") \times w) / (2 + w).$$
Type options can be set procedurally at any time during simulation. The syntax is:

```
coverage_type_option_assignment ::= // Not in Annex A
    covergroup_name::type_option.option_name = expression;
    | covergroup_name::covergroup_item_identifier::type_option.option_name = expression;
```

Here is an example:

```verilog
covergroup gc @(posedge clk) ;
    a : coverpoint a_var;
    b : coverpoint b_var;
endgroup
...
gc::type_option.comment = "Here is a comment for covergroup g1";
gc::a::type_option.weight = 3; // Set the weight for coverpoint “a” of covergroup g1
gc g1 = new;
```

The following table summarizes the syntactical level (covergroup, coverpoint, or cross) in which type options can be specified. When set at the covergroup level, the type options do not act as defaults for lower syntactic levels.

<table>
<thead>
<tr>
<th>Option name</th>
<th>Allowed Syntactic Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>covergroup</td>
</tr>
<tr>
<td>weight</td>
<td>Yes</td>
</tr>
<tr>
<td>goal</td>
<td>Yes</td>
</tr>
<tr>
<td>comment</td>
<td>Yes</td>
</tr>
<tr>
<td>strobe</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Table 20-4: Coverage type-options*

### 20.7 Predefined coverage methods

The following coverage methods are provided for the covergroup. These methods can be invoked procedurally at any time.

<table>
<thead>
<tr>
<th>Method (function)</th>
<th>Can be called on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>covergroup</td>
<td>coverpoint</td>
</tr>
<tr>
<td>void sample()</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>real get_coverage()</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>real get_inst_coverage()</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>void set_inst_name(string)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>void start()</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>void stop()</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>real query()</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>real inst_query()</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Table 20-5: Predefined coverage methods*

### 20.8 Predefined coverage system tasks

SystemVerilog provides the following system tasks to help manage coverage data collection.
**$set_coverage_db_name**( name ) – Sets the filename of the coverage database into which coverage information is saved at the end of a simulation run.

**$load_coverage_db** ( name ) – Load from the given filename the cumulative coverage information for all coverage group types.

**$get_coverage()** – Returns as a real number in the range 0 to 100 the overall coverage of all coverage group types. This number is computed as described above.

**$get_inst_coverage()** – Returns as a real number in the range 0 to 100 the overall per-instance coverage of all instances of all coverage group. This number is computed as described above.

### 20.9 Querying coverage at run time

There are two methods that allow querying of coverage information during simulation. These are described in Section 20.7.

### 20.10 Functional coverage BNF

```
covergroup_declaration ::= covergroup covergroup_identifier [ ( list_of_task_proto ) ] [ clocking_event ]; { coverage_spec_or_option ; } endgroup [ : covergroup_identifier ]

list_of_task_proto ::= task_proto_formal , task_proto_formal

coverage_spec_or_option ::= {attribute_instance} coverage_spec | {attribute_instance} coverage_option

coverage_option ::= option.option_name = expression | type_option.option_name = expression

coverage_spec ::= cover_point | cover_cross

variable_decl_assignment ::= // covergroup creation

| covergroup_identifier variable_identifier = new [ ( list_of_arguments ) ]

cover_point ::= [label :] 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 [ [[expression]] ] = { range_list } [ iff (expression) ]

| [ wildcard ] bins_keyword bin_identifier [ [ ] ] = { trans_list } [ iff (expression) ]

| bins_keyword bin_identifier [ [[expression]] ] default [ iff (expression) ]

| bins_keyword bin_identifier default_sequence [ iff (expression) ]
```
bins_keyword::= bins | illegal_bins | ignore_bins

range_list ::= value_range { , value_range }

value_range ::= // from Annex A.8.3
    expression
    | [ expression : expression ]

trans_list ::= trans_set { , trans_set }

trans_set ::= trans_range_list -> trans_range_list { -> trans_range_list }

trans_range_list ::= trans_item
    | trans_item [ [* value_range ] ] // consecutive repetition
    | trans_item [ *[-> value_range ] ] // goto repetition
    | trans_item [ [*= value_range ] ] // non-consecutive repetition

trans_item ::= { range_list } | value_range

cover_cross ::= [label:] cross list_of_coverpoints [ iff ( expression ) ] select_bins_or_empty

list_of_coverpoints ::= cross_item , cross_item { , cross_item }

cross_item ::= cover_point_identifier
    | variable_identifier

select_bins_or_empty ::= { { bin_selection_or_option ; } }
    | ;

bin_selection_or_option ::= {attribute_instance} coverage_option
    | {attribute_instance} bins_selection

bins_selection ::= bins_keyword bin_identifier = select_expression

select_expression ::= select_condition
    | ! select_condition
    | select_expression && select_expression
    | select_expression || select_expression
    | ( select_expression )

select_condition ::= binsof ( bins_expression ) [ . intersect open_range_list ]

bins_expression ::= variable_identifier
    | cover_point_identifier [ , bins_identifier ]

open_range_list ::= { open_value_range { , open_value_range }

open_value_range ::= expression
| expression : expression |
| expression : $ |
| $ : expression |
Modify Syntax of Section 8.10 (Event control) as shown

```
delay_or_event_control ::= // from Annex A.6.5
    delay_control
  | event_control
  | repeat ( expression ) event_control

delay_control ::=
    # delay_value
  | # ( mintypmax_expression )

event_control ::=
    @ event_identifier
  | @ ( event_expression )
  | @*
  | @ (*)

event_expression ::= [ edge_identifier ] expression [ iff expression ]
  | event_expression or event_expression
  | event_expression , event_expression
  | begin hierarchical_btf_identifier
  | end hierarchical_btf_identifier

hierarchical_btf_identifier ::= hierarchical_task_identifier
  | hierarchical_function_identifier
  | hierarchical_block_identifier
  | hierarchical_identifier { class_identifier :: } method_identifier

edge_identifier ::= posedge | negedge // from Annex A.7.4
```

Syntax 8-8—Delay and event control syntax (excerpt from Annex A)

Add to the end of Section 8.10 (Event control)

SystemVerilog event expressions can be triggered by the start or the end of execution of a given named block, task, function, or class method. Event expressions that specify the begin keyword followed by a hierarchical identifier denoting a named block, task, function, or class method shall be triggered immediately before the corresponding block, task, function, or method begins executing its first statement. Event expressions that specify the end keyword followed by a hierarchical identifier denoting a named block, task, function, or class method shall be triggered immediately after the corresponding block, task, function, or method executes its last statement. Event expressions that specify the end of execution shall not be triggered if the block, task, function, or method is disabled.

For example:

```verbatim
task send_receive(inout byte b);
    bus <= b;
    # 5
    b = bus;
endtask
```
task check_sr();
    @( begin send_receive ) $	ext{display}( "sent some data" );
    @( end send_receive ) $	ext{display}( "received some data" );
endtask

When task check_sr is called, it will block until task send_receive is called. The first line of task check_sr unblocks when a call to send_receive takes place. Likewise, the second line of task_sr will wait until the task send_receive terminates (i.e., the task returns).