**Mantis 2506 - Non-trivial coverage space shapes and joint conditions are difficult to specify with covergroups**

**Motivation**

Coverage spaces often have non-trivial shapes that may be succinctly expressed by expressions or characteristic functions. Cross products often have non-trivial joint conditions on their components that again may be succinctly expressed by expressions or characteristic functions.

Expressing coverage spaces with such shapes and joint conditions is laborious with the existing covergroup features.

**In Section 19.5, syntax 19-2**

REPLACE

cover\_point ::=

 [ 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 [ **[** [ expression ] **]** ] = **{** open\_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**

open\_range\_list ::= open\_value\_range { **,** open\_value\_range }

open\_value\_range ::= value\_range

WITH

Note to the editor: some of the BNF changes below are additions to the BNF, even though they are not in blue (new items that should be colored red in the BNF are colored red not blue). This is true for all BNF changes in this proposal.

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\_constant\_expression ] **]** ] **=** **{** covergroup~~open~~\_range\_list **}** [ **with** **(** with\_expression **)** ] [ **iff** **(** expression **)** ]

 | [ **wildcard** ] bins\_keyword bin\_identifier [ **[** [ covergroup\_constant\_expression ] **]** ] = cover\_point\_identifier **with** **(** with\_expression **)** [ **iff** **(** expression **)** ]

 | [ **wildcard** ] bins\_keyword bin\_identifier [ **[** [ covergroup\_constant\_expression ] **]** ] = set\_expression [ **iff** **(** expression **)** ]

 | [ **wildcard**] bins\_keyword bin\_identifier [ **[ ]** ] **=** trans\_list [ **iff (** expression **)** ]

 | bins\_keyword bin\_identifier [ **[** [ covergroup\_constant\_expression ] **]** ] **= default** [ **iff (** expression **)** ]

 | bins\_keyword bin\_identifier **= default sequence** [ **iff (** expression **)** ]

bins\_keyword::= **bins** | **illegal\_bins** | **ignore\_bins**

covergroup~~open~~\_range\_list ::= covergroup~~open~~\_value\_range { **,** covergroup~~open~~\_value\_range }

~~open\_value\_range ::= value\_range~~

covergroup\_value\_range ::=

 covergroup\_constant\_expression

 | **[** covergroup\_constant\_expression **:** covergroup\_constant\_expression **]**

with\_expression ::= covergroup\_constant\_expression

set\_expression ::= covergroup\_constant\_expression

covergroup\_constant\_expression ::= expressionfn1

-------------------------

fn1) This expression is restricted as described in 19.5.

**In Section 19.5**

REPLACE:

A **coverpoint** coverage point creates a hierarchical scope and can be optionally labeled. If the label is specified, then it designates the name of the coverage point. This name can 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 can generate a name for the coverage point only for the purposes of coverage reporting, that is, generated names cannot be used within the language.

A coverpoint name has limited visibility. An identifier can only refer to a coverpoint in the following contexts:

— In the coverpoint list of a **cross** declaration (see 19.6),

— In a hierarchical name where the prefix specifies the name of a covergroup variable. For example, cov1.cp.option.weight where cov1 is the name of a covergroup variable and cp is the name of a coverpoint declared within the covergroup.

— Following ::, where the left operand of the scope resolution operator refers to a covergroup. For example, covtype :: cp :: type\_option.weight.

For example:

**covergroup** cg ( **ref int** x , **ref int** y, **input int** c);

 **coverpoint** x; // creates coverpoint "x" covering the formal "x"

 x: **coverpoint** x; // INVALID: coverpoint label "x" already exists

 b: **coverpoint** y; // creates coverpoint "b" covering the formal "y"

 c: **coverpoint** x; // creates coverpoint "c" covering the formal "x"

 option.weight = c; // set weight of "cg" to value of formal "c"

 d: **coverpoint** x {

 option.weight = 2; // set the weight of coverpoint "d"

 }

 d.option.weight = 2; // INVALID use of "d", also syntax error

 **cross** x, y { // Creates implicit coverpoint "y" covering

 // the formal "y". Then creates a cross of

 // coverpoints "x", "y"

 option.weight = c; // set weight of cross to value of formal "c"

 }

 b: **cross** y, x; // INVALID: coverpoint label "b" already exists

**endgroup**

WITH:

A **coverpoint** coverage point creates a hierarchical scope and can be optionally labeled. If the label is specified, then it designates the name of the coverage point. This name can 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 can generate a name for the coverage point only for the purposes of coverage reporting, that is, generated names cannot be used within the language.

A data type for the coverpoint may be specified explicitly or implicitly in *data\_type\_or\_implicit*. In either case, it shall be understood that a data type is specified for the coverpoint. The data type shall be an integral type.

If a data type is specified, then the coverpoint expression shall be assignment compatible with the data type. Values for the coverpoint shall be of the specified data type and shall be determined as though the coverpoint expression were assigned to a variable of the specified data type.

If no data type is specified, then the inferred data type for the coverpoint shall be the self-determined type of the coverpoint expression.

A coverpoint name has limited visibility. An identifier can only refer to a coverpoint in the following contexts:

— In the coverpoint list of a **cross** declaration (see 19.6),

— In a hierarchical name where the prefix specifies the name of a covergroup variable. For example, cov1.cp.option.weight where cov1 is the name of a covergroup variable and cp is the name of a coverpoint declared within the covergroup.

— Following ::, where the left operand of the scope resolution operator refers to a covergroup. For example, covtype :: cp :: type\_option.weight.

Only constant expressions (see 11.2.1), instance constants (for an embedded covergroup), or non-ref arguments to the covergroup are allowed to be used in the following coverpoint constructs:

* *with\_expression*
* *select\_expression*
* *open\_range\_list*, or
* an expression specifying a fixed number of bins

Instance constants referenced from a covergroup shall be members of the enclosing class. The initializers for such instance constants shall appear before the referring covergroup constructor call in the class constructor. These initializers shall not appear with the covergroup constructor call in the body of any looping statement (see 12.7) or fork-join\_none, either before or after.

Function calls may participate in expressions within the coverpoint, but the following semantic restrictions are imposed:

* Functions shall not contain **output** or non-**const** **ref** arguments (**const ref** is allowed).
* Functions shall be automatic, preserve no state information, and have no side effects.
* Functions shall not reference non-constant variables outside the local scope of the function.
* User-defined system task or function calls are restricted to constant system function calls (see 11.2.1)

For example:

**covergroup** cg ( **ref int** x , **ref int** y, **input int** c);

 **coverpoint** x; // creates coverpoint "x" covering the formal "x"

 x: **coverpoint** ~~x~~z; // INVALID: coverpoint label "x" already exists

 b: **coverpoint** y; // creates coverpoint "b" covering the formal "y"

 c: **coverpoint** x; // creates coverpoint "c" covering the formal "x"

 option.weight = c; // set weight of "cg" to value of formal "c"

 bit [7:0] **coverpoint** y[31:24];// creates coverpoint “y” covering the high

 // order 8 bits of the formal “y”

 d: **coverpoint** x {

 option.weight = 2; // set the weight of coverpoint "d"

 }

 d.option.weight = 2; // INVALID use of "d", also syntax error

 **cross** x, y { // Creates implicit coverpoint "y" covering

 // the formal "y". Then creates a cross of

 // coverpoints "x", "y"

 option.weight = c; // set weight of cross to value of formal "c"

 }

 b: **cross** y, x; // INVALID: coverpoint label "b" already exists

**endgroup**

**In Section 19.5**

ADD and RENUMBER 19.5.1 to 19.5.2, 19.5.2 to 19.5.3, etc.

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.

**19.5.1 Specifying bins for values**

**In Section 19.5**

REPLACE:

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, **[]**, shall follow the bin name. To create a fixed number of bins for a set of values, a number can be specified inside the square brackets. The *open\_range\_list* used to specify the set of values associated with a bin shall be constant expressions (see 11.2.1), instance constants (for classes only), or non-**ref** arguments to the coverage group. It shall be legal to use the $ primary in an *open\_value\_range* of the form [ expression : $ ] or [ $ : expression ].

WITH:

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, **[]**, shall follow the bin name. To create a fixed number of bins for a set of values, a single positive integral expression ~~number~~ can be specified inside the square brackets. ~~The~~ *~~open\_range\_list~~* ~~used to specify the set of values associated with a bin shall be constant expressions (see 11.2.1), instance constants (for classes only), or non-~~**~~ref~~** ~~arguments to the coverage group.~~ It shall be legal to use the $ primary in an *open\_value\_range* of the form [ expression : $ ] or [ $ : expression ].

**In Section 19.5**

ADD

The example above defines a coverage group, cg, in which the signal to be sampled and 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.

**19.5.1.1 Coverpoint bin with clause**

The **with** clause specifies that only those values in the *open\_range\_list* that satisfy the given expression (i.e., for which the expression evaluates to true) are included in the bin. The truth value of the **with** clause expression is interpreted in the same way an expression is interpreted in the condition of a procedural **if** statement (see 12.4). In the expression, the name item shall be used to represent the candidate value. The candidate value is of the same type as the coverpoint.

The name of the coverpoint itself may be used in place of the *open\_range\_list* to denote all values of the coverpoint. Only the name of the coverpoint containing the bin being defined shall be allowed; no other coverpoint names shall be permitted.

Consider the following example:

 a: **coverpoint** x

 {

 **bins** mod3[] = {[0:255]} **with** (item % 3 == 0);

 }

This bin definition selects all values from 0 to 255 that are evenly divisible by 3.

 **coverpoint** b

 {

 **bins** func[] = b **with** (myfunc(item));

 }

Note the use of the coverpoint name b to denote that the *with\_expression* will be applied to all values of the coverpoint.

As with array manipulation methods involving **with** (see 7.12), if the expression has side effects, the results are unpredictable.

The **with** clause behaves as if the expression were evaluated for every value in the *open\_range\_list* at the time the covergroup instance is constructed. By default the *with\_expression* is applied to the set of values in the *open\_range\_list* prior to distribution of values to the bins. If the distribution of values is desired before *with\_expression* application the distribute\_first covergroup option (see 19.7.1) can be used to achieve this ordering. The result of applying a *with\_expression* shall preserve multiplicity of bin items as well as bin order. The intent of these rules is to allow for the use of non-simulation analysis techniques to calculate the bin (for example, formal symbolic analysis), or for caching of previously calculated results.

**19.5.1.2 Coverpoint bin set expression**

The *set\_expression* syntax allows specifying an expression yielding an array of values that define the bin. Any array whose element type is assignment compatible with the coverpoint type is permitted, with the exception that associative arrays are not permitted.

Identifiers declared within the covergroup (such as coverpoint identifiers and bin identifiers) are not visible. The expression is evaluated when the covergroup instance is constructed.

**In Section 19.5.6**

REPLACE

A coverpoint expression and the expressions in a **bins** construct are involved in comparison operations in order to determine into which bins a particular value falls. Let e be the coverpoint expression and b be an expression in a **bins** *open\_range\_list*. The following rules shall apply when evaluating e and b: For **wildcard** bins, x and z values in b shall be treated as all possible 0 and 1 values prior to applying these rules.

a) e shall be self-determined

b) b shall be evaluated as though it were the right-hand side of an assignment to a variable whose type is **type**(e). Enumeration values in expressions b and e shall first be treated as being in an expression context. This implies that the type of an enumeration value is the base type of the enumeration and not the enumeration type itself. An implementation shall issue a warning under the following conditions:

1) If e is unsigned and b is signed with a negative value.

2) If assigning b to a variable of type **type**(e) would yield a value that is not equal to b under normal comparison rules for ==.

3) If b yields a value with any x or z bits. This rule does not apply to **wildcard** bins because x and z shall be treated as 0 and 1 as described above.

WITH

A coverpoint expression, ~~and~~ the expressions in a **bins** construct, and the coverpoint type, if present, are involved in comparison operations in order to determine into which bins a particular value falls. Let e be the coverpoint expression and b be an expression in a **bins** *open\_range\_list*. The following rules shall apply when evaluating e and b: For **wildcard** bins, x and z values in b shall be treated as all possible ~~0~~**0** and ~~1~~**1** values prior to applying these rules.

a) e shall be self-determined

b) If there is no coverpoint type, then b shall be evaluated as though it were the right-hand side of an assignment to a variable whose type is **type**(e). In the presence of a coverpoint type, e and b shall be evaluated as though it were the right-hand side of an assignment to a variable whose type is the coverpoint type. Enumeration values in expressions b and e shall first be treated as being in an expression context. This implies that the type of an enumeration value is the base type of the enumeration and not the enumeration type itself. An implementation shall issue a warning under the following conditions if a coverpoint type is not present:

1) If e is unsigned and b is signed with a negative value.

2) If assigning b to a variable of type **type**(e) would yield a value that is not equal to b under normal comparison rules for ==.

3) If b yields a value with any ~~x~~x or ~~z~~z bits. This rule does not apply to **wildcard** bins because ~~x~~x and ~~z~~z shall be treated as ~~0~~0 and ~~1~~1 as described above.

An implementation shall issue a warning under the following conditions if a coverpoint type is present.

1) If the coverpoint type is unsigned and b is signed with a negative value.

2) If assigning b to a variable of coverpoint type would yield a value that is not equal to b under normal comparison rules for ==.

3) If b yields a value with any x or z bits. This rule does not apply to **wildcard** bins because x and z shall be treated as 0 and 1 as described above.

**In Section 19.6, syntax 19-4**

REPLACE

cover\_cross ::=

 [ cross\_identifier **:** ] **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 ::=

 **{** { 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\_condition ::= **binsof (** bins\_expression **)** [ **intersect {** open\_range\_list **}** ]

bins\_expression ::=

 variable\_identifier

 | cover\_point\_identifier [ **.** bin\_identifier ]

open\_range\_list ::= open\_value\_range { **,** open\_value\_range }

open\_value\_range ::= value\_range

WITH

cover\_cross ::=

 [ cross\_identifier **:** ] **cross** list\_of\_coverpoints [ **iff (** expression **)** ] ~~select\_bins\_or\_empty~~cross\_body

list\_of\_coverpoints ::= cross\_item **,** cross\_item { **,** cross\_item }

cross\_item ::=

 cover\_point\_identifier

 | variable\_identifier

~~select\_bins\_or\_empty~~cross\_body ::=

 **{** { bins\_selection\_or\_option **;** } **}**

 | function\_declaration

 | **;**

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\_expression **)** [ **matches** integer\_expression ]

 | cross\_identifier

 | cross\_set\_expression [ **matches** integer \_expression ]

select\_condition ::= **binsof (** bins\_expression **)** [ **intersect {** covergroup~~open~~\_range\_list **}** ]

bins\_expression ::=

 variable\_identifier

 | cover\_point\_identifier [ **.** bin\_identifier ]

covergroup\_open\_range\_list ::= covergroup~~open~~\_value\_range { **,** covergroup~~open~~\_value\_range }

~~open\_value\_range ::= value\_range~~

covergroup\_value\_range ::=

 covergroup\_constant\_expression

 | **[** covergroup\_constant\_expression **:** covergroup\_constant\_expression **]**

with\_expression ::= covergroup\_constant\_expression

integer\_expression ::=

 covergroup\_constant\_expression

 | **$**

cross\_set\_expression ::= covergroup\_constant\_expression

**In Section 19.6**

REPLACE

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

A **cross** name has limited visibility. An identifier can only refer to a **cross** in the following contexts:

— In a hierarchical name where the prefix specifies the name of a covergroup variable. For example, cov1.crs.option.weight where cov1 is the name of a covergroup variable and crs is the name of a **cross** declared within the covergroup.

— Following :: where the left operand of the scope resolution operator refers to a covergroup. For example, covtype :: crs :: type\_option.weight.

WITH

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

A **cross** name has limited visibility. An identifier can only refer to a **cross** in the following contexts:

— In a hierarchical name where the prefix specifies the name of a covergroup variable. For example, cov1.crs.option.weight where cov1 is the name of a covergroup variable and crs is the name of a **cross** declared within the covergroup.

— Following :: where the left operand of the scope resolution operator refers to a covergroup. For example, covtype :: crs :: type\_option.weight.

Identifiers and function calls within the cross are restricted in the same way as identifiers and function calls within coverpoints (see 19.5). Functions declared within the cross shall not be visible outside of that scope.

**In Section 19.6**

ADD and RENUMBER 19.6.1 to 19.6.2, 19.6.2 to 19.6.3, etc.

**19.6.1: Defining cross coverage bins**

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.

**In Section 19.6**

ADD

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

**19.6.1.1 Cross bin with clause**

The **with** clause in a *select\_expression* specifies that only those bin tuples in the subordinate *select\_expression* for which sufficiently many value tuples satisfy the given *with\_expression* (i.e., for which the expression evaluates to true) are selected. The truth value of the **with** clause expression is interpreted in the same way an expression is interpreted in the condition of a procedural **if** statement (see 12.4).

In the expression, occurrences of *cross\_items* (i.e., those *coverpoint\_identifiers* or *variable\_identifiers* occurring in the *list\_of\_coverpoints* for the cross) represent corresponding values in the value tuples of the candidate bin tuples.

When a *cross\_identifier* is used as a *select\_expression*, it selects all possible bin tuples. When used with a **with** clause, the cross bin can be completely described using a *with\_expression*. Only the *cross\_identifier* of the enclosing cross may be used; other *cross\_identifiers* shall be disallowed.

The optional **matches** clause specifies the selection policy. The *integer\_expression* shall evaluate to a positive integer or $, representing the minimum number of satisfying value tuples required to select the candidate bin tuple. The $ symbol specifies that all value tuples must satisfy the expression to select the candidate bin tuple. When the **matches** clause is omitted, the selection policy defaults to one.

Consider the following example:

 **logic** [0:7] a, b;

 **parameter** [0:7] mask;

 ...

 **covergroup** cg;

 **coverpoint** a

 {

 **bins** low[] = {[0:127]};

 **bins** high = {[128:255]};

 }

 **coverpoint** b

 {

 **bins** two[] = b **with** (item % 2 == 0)

 **bins** three[] = b **with** (item % 3 == 0)

 }

 X: **cross** a,b

 {

 **bins** apple = X **with** (a+b < 257) **matches** 127;

 **bins** cherry = (**binsof**(b) **intersect** [0:50] && **binsof**(a.low) **intersect** [0:50]) **with** (a==b);

 **bins** plum = **binsof**(b.two) **with** (b > 12) || **binsof**(a.low) **with** (a & b & mask);

 }

 **endgroup**

The bin structure for coverpoint a is straightforward - bin array low contains 128 single-element bins for each value between 0 and 127, and bin high contains all values from 128 to 255. The bins of coverpoint b are specified using the **with** clause; bin array two contains a bin for each even value, and three contains a bin for each value divisible by 3.

The cross X crosses coverpoints a and b. Three cross bins are defined, apple, cherry, and plum. apple consists of all bin tuples for which a+b < 257 for at least 127 value tuples. In this example apple would consist of three coverpoint bin tuples: <high, two[0]>, <high, two[1]>, and <high, three[0]>.

The cross bin cherry demonstrates using the **with** clause on a complex *select\_expression*. First, those bin tuples consisting of a bin from b containing a value between 0 and 50 are selected; then, the && operator selects from those bin tuples the bin tuples with a bin from a.low containing a value between 0 and 50. The **with** clause then selects from those only the bin tuples containing at least one value tuple where a==b.

The cross bin plum demonstrates a *select\_expression* composed of **with** expressions. The first **with** expression selects those bin tuples containing bins in the b.two bin array whose values are greater than 12. The || operator then adds the bin tuples selected by the second **with** expression – namely those containing a bin from a.low and for which the bitwise-AND of the a-value, b-value and a mask is non-zero for some values a and b in the bins of the bin tuple.

As with array manipulation methods involving **with** (see 7.12), if the expression has side effects, the results are unpredictable.

The **with** clause behaves as if the expression were evaluated for every value tuple of every bin tuple selected by the subordinate *select\_expression* at the time the covergroup instance is constructed. However, implementations are not required to schedule the evaluation events when calculating the bin tuples in the cross bin; all, some, or none of the events may be scheduled. This allows for the use of non-simulation analysis techniques to calculate the cross bin (for example, formal symbolic analysis), or for caching of previously calculated results.

**19.6.1.2 Cross bin automatically-defined types**

A cross defines a coverage space composed of tuples of values. To aid in describing the structure of that space, SystemVerilog provides automatically-defined types for these tuples and queues of tuples in each cross. The types are named CrossValType and CrossQueueType. The scope of the type names is the cross itself and the types are not accessible outside of this scope.

The definition of CrossValType is a SystemVerilog struct consisting of one member for each coverpoint in the cross. The name and type of each field are the name and type of the corresponding coverpoint. If range bounds for the coverpoint type are not evident (e.g., the coverpoint expression is a concatenation and no other type is specified), the bounds are assumed to be [$bits(coverpoint\_expression)-1:0]. The definition of CrossQueueType is an unbounded queue of CrossValType elements.

The cross types shall be considered as implicit typedefs in the body of the cross, even though the syntax does not allow typedefs to appear there explicitly. Consider the following example:

 **covergroup** cg (**ref logic** [0:3] x, **ref logic** [0:7] y, **ref logic** [0:2] a);

 xy: **coverpoint** {x,y};

 **coverpoint** y;

 XYA: **cross** xy, a

 {

 // the cross types are as if defined here as follows:

 // typedef struct {logic [11:0] xy;logic [0:2] a;} CrossValType;

 // typedef CrossValType CrossQueueType[$];

 };

 **endgroup**

Section 19.6.1.3 shows how CrossValType and CrossQueueType can be used to compute explicit enumerations of cross bins.

**19.6.1.3 Cross bin set expression**

The *cross\_set\_expression* syntax allows specifying an expression yielding a queue of elements that define the cross bin, similarly to the *set\_expression* for coverpoint bins. However, for cross bins the type of the queue shall be the cross's CrossQueueType, whose elements are of type CrossValType (see Sec 19.6.1.2).

The selection of bin tuples for the cross bin by the elements of the *cross\_set\_expression* is subject to the same policy specification as the cross bin **with** expression (see 19.6.1.1). The optional **matches** expression specifies the number of value tuples in a bin tuple that must be present in the *cross\_set\_expression* for that bin tuple to be selected. The default policy is one, denoting the policy where a single value tuple from a bin tuple must exist in the *cross\_set\_expression* to select the bin tuple.

For example,

 **int** a;

 **logic** [7:0] b;

 **covergroup** cg;

 **coverpoint** a { **bins** x[] = {[0:10]}; }

 **coverpoint** b { **bins** y[] = {[0:20]}; }

 aXb : **cross** a, b

 {

 **bins** one = '{ '{1,2}, '{3,4}, '{5,6} };

 }

 endgroup

The cross bin definition uses an array literal to define the bin tuples in cross bin one as <a.x[1], b.y[2]>, <a.x[3], b.y[4]>, and <a.x[5], b.y[6]>. Here, the cross bin provides the context required to determine the type of the literal (in this case, the cross's CrossQueueType). In general, literal arrays are not required; any expression may be used as long as it evaluates to the cross's CrossQueueType. A cast is required if the type is assignment-incompatible with the cross type.

Below is a more involved example:

 **module** mod\_m;

 **logic** [31:0] a, b;

 **covergroup** cg(**int** cg\_lim);

 **coverpoint** a;

 **coverpoint** b;

 aXb : **cross** a, b

 {

 **function** CrossQueueType myFunc1(**int** f\_lim);

 **for** (**int** i = 0; i < f\_lim; ++i)

 myFunc1.push\_back(CrossValType'('{i,i}));

 **endfunction**

 **bins** one = myFunc1(cg\_lim);

 **bins** two = myFunc2(cg\_lim);

 **function** CrossQueueType myFunc2(**logic** [31:0] f\_lim);

 **for** (**logic** [31:0] i = 0; i < f\_lim; ++i)

 myFunc2.push\_back('{2\*i,2\*i});

 **endfunction**

 }

 **endgroup**

 cg cg\_inst = **new**(3);

 **endmodule**

Here we use functions to create the queues that define the cross bins. Note that the coverpoints a and b are 32 bits wide; iterating over all value tuples using a **with** expression would be computationally expensive. By using functions, the user is able to restrict the bin computation to a reasonable subset of value tuples; the entire cross space need not be considered.

The function myFunc1 requires a cast when calling push\_back since the array item’s type is defined using a type which does not match the automatically-defined cross type, but is cast-compatible. Cross bin two, however, does not require a cast, since myFunc2 is defined using explicit references to the cross bin types. The call to myFunc2 takes advantage of SystemVerilog's rule allowing forward references to functions.

As shown, the bins for cg\_inst are as follows:

 cg\_inst.aXb.one = <0,0>, <1,1>, <2,2>

 cg\_inst.aXb.two = <0,0>, <2,2>, <4,4>

**In 19.7.1, table 19-3**

ADD to the table

Option name: **distribute\_first**=*boolean*

Default: 0

Description: When true, instructs the tool to perform value distribution to the bins prior to application of the *with\_expression*.

**In A.2.11**

REPLACE

cover\_point ::=

 [ 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 [ **[** [ expression ] **]** ] = **{** open\_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 }

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 ::= range\_list

repeat\_range ::=

expression

| expression **:** expression

cover\_cross ::=

 [ cross\_identifier **:** ] **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 ::=

 **{** { 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\_condition ::= **binsof (** bins\_expression **)** [ **intersect {** open\_range\_list **}** ]

bins\_expression ::=

 variable\_identifier

 | cover\_point\_identifier [ **.** bin\_identifier ]

open\_range\_list ::= open\_value\_range { **,** open\_value\_range }

open\_value\_range ::= value\_range

WITH

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\_constant\_expression ] **]** ] **=** **{** covergroup~~open~~\_range\_list **}** [ **with** **(** with\_expression **)** ] [ **iff** **(** expression **)** ]

 | [ **wildcard** ] bins\_keyword bin\_identifier [ **[** [covergroup\_constant\_expression ] **]** ] = cover\_point\_identifier **with** **(** with\_expression **)** [ **iff** **(** expression **)** ]

 | [ **wildcard** ] bins\_keyword bin\_identifier [ **[** [covergroup\_constant\_expression ] **]** ] = set\_expression [ **iff** **(** expression **)** ]

 | [ **wildcard**] bins\_keyword bin\_identifier [ **[ ]** ] **=** trans\_list [ **iff (** expression **)** ]

 | bins\_keyword bin\_identifier [ **[** [covergroup\_constant\_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 }

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 ::= range\_list

repeat\_range ::=

expression

| expression **:** expression

cover\_cross ::=

 [ cross\_identifier **:** ] **cross** list\_of\_coverpoints [ **iff (** expression **)** ] ~~select\_bins\_or\_empty~~cross\_body

list\_of\_coverpoints ::= cross\_item **,** cross\_item { **,** cross\_item }

cross\_item ::=

 cover\_point\_identifier

 | variable\_identifier

~~select\_bins\_or\_empty~~cross\_body ::=

 **{** { bins\_selection\_or\_option **;** } **}**

 | function\_declaration

 | **;**

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\_expression **)** [ **matches** integer\_expression ]

 | cross\_identifier

 | cross\_set\_expression [ **matches** integer \_expression ]

select\_condition ::= **binsof (** bins\_expression **)** [ **intersect {**covergroup~~open~~\_range\_list **}** ]

bins\_expression ::=

 variable\_identifier

 | cover\_point\_identifier [ **.** bin\_identifier ]

~~open\_range\_list ::= open\_value\_range {~~ **~~,~~** ~~open\_value\_range }~~

~~open\_value\_range ::= value\_range~~

covergroup\_value\_range ::=

 covergroup\_constant\_expression

 | **[** covergroup\_constant\_expression **:** covergroup\_constant\_expression **]**

with\_expression ::= covergroup\_constant\_expression

set\_expression ::= covergroup\_constant\_expression

integer\_expression ::=

 covergroup\_constant\_expression

 | **$**

cross\_set\_expression ::= covergroup\_constant\_expression

covergroup\_constant\_expression ::= expression

**In A.2.11**

ADD

fn1) This expression is restricted as described in 19.5.