David W. Smith

From: owner-sv-ec@eda.org on behalf of Arturo Salz [Arturo.Salz@Synopsys.COM]
Sent: Thursday, April 15, 2004 11:42 PM
To: Ryan, Ray; sv-ec@eda.org
Subject: Re: [sv-ec] Questions on Section 20 Coverage

Ray,

My comments are interspersed below in this color.

Arturo

----- Original Message -----
From: Ryan, Ray
To: sv-ec@eda.org
Sent: Thursday, March 11, 2004 1:30 PM
Subject: [sv-ec] Questions on Section 20 Coverage

Below are a few questions on Section 20 (Coverage) regarding the specification of transitions and about the computation of coverage statistics (i.e. how to compute the coverage values returned by the predefined coverage methods).

Regarding computation of coverage,

1) What is the difference between 'cumulative' coverage and 'type' coverage. These seem to be the same, however there are separate predefined methods to access the cumulative or type coverage. What is the difference between 'get_coverage()' and 'query()' or between 'get_inst_coverage()' and 'inst_query()' ?

Cummulative and type coverage are the same.
The **get_coverage()** method returns the cummulative coverage.
The **get_inst_coverage()** method returns the coverage for the instance on which that it is invoked.
The **query()** and **inst_query()** methods return detailed cummulative coverage information.
They are not defined in the LRM --- an omission that will be corrected by another erratum.

The LRM refers to the 'cumulative coverage', but does not really define how cumulative coverage is computed. Since the coverage numbers are returned (computed) by predefined methods, the LRM should be clear as to the computation of coverage numbers.

Here's a brief description of how cummulative coverage is computed.

Coverage of covergroup Cg is computed as:

$$C_g = \frac{\sum W_i * C_i}{\sum W_i}$$

$i \in$ Set of coverpoint and cross items defined in covergroup Cg

where:

W_i - Weight of item i

C_i - Coverage of item i (see below)

---

(1) Coverage of coverpoint item, Ci, depends on the coverpoint type: autobin or user-defined:

(a) user-defined coverpoint

$$C_i = \frac{\text{\# of bins covered}}{\text{\# of bins defined}}$$

(b) autobin coverpoint

$$C_i = \frac{\text{\# of bins covered}}{\mathbf{MIN}(\text{auto\_bin\_max}, 2^M)}$$

M - Number of bits needed to represent the coverpoint.

(2) Coverage of cross item, Ci:

$$C_i = \frac{\text{\# of bins covered}}{B_p + B_u}$$

$$B_p = \mathbf{MIN}(B_c, \text{cross\_auto\_bin\_max})$$

$$B_c = \left(\prod_{j \in \text{Set of coverpoints being crossed}} B_j\right) - B_b$$

Note: B_j - Number of bins defined in the j'th coverpoint being cross

where:

B_p - # of possible auto-cross bins

B_u - # of significant user-defined cross bins (bb below) --- excluding ignored and illegal bins

B_b - # of cross products that are included in all user-defined cross-bins, such as:

```
bins bb = binsof ...
```

```
ignore_bins bg = binsof ...
illegal_bins bi = binsof ...
```

---

2) At the end of 20.4 the last two paragraphs and the embedded example describe passing arguments to a constructor as a means to define a generic coverage group. Modifying the example slightly to specify a separate 'good' bin for each value yields the following example (the only difference is the [] following good) :

```
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

...
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
```

By default the covergroup (type) **gc** has the **per_instance** option set to false (0). In this case how is the type coverage for **gc** computed since each instance can have a different number of bins and the different sets of values associated with the bins.

The cummulative coverage includes the union of all significant bins, that is, the cummulative coverage includes the bins for all (possibly overlapping) bins of all instances. In this example, the bins [0 - 50] and [120 - 600].

---

3) When computing the cumulative coverage, are the bin counts from separate instances summed before comparison to the 'at_least' value to determine if a particular bin is covered for the covergroup type (or does the 'at_least' count need to be observed for each instance) ?

To determine if a particular bin of the covergroup type has been covererd, the computation uses the maximum of the 'at_least' values of all instances. The naximum represents the more conservative choice.

---

4) The last two paragraphs in 20.4 (mentioned in #2) belong in 20.2, since they discuss (introduces) writing a generic **coverage group**, whereas the topic of section 20.4 is **coverage points**.

Those two paragraphs describe how to define a coverpoint by reference - how arguments passed by reference become coverpoints. This feature does allow definition of a generic coverage-group. However, the intent of this section is to show how a coverpoint defintion is passed by reference, which is the reason why it is included in section 20.4 -- coverage points.

Related to specification of transitions,

5) As shown in the text and example in 20.4.1, non-consecutive repetition can be specified using [-> repeat_range]. For example,
  2 [-> 3]
is the same as

... 2 => ... => 2 ... => 2
where the dots (...) represent any transition that does not contain the value 2.

However, it appears that **repetition** is the **only** way to specify non-consecutive transitions values. Is there any way to specify non-consecutive transitions values that do not repeat? For example, is there any way to specify the transition:
... 1 => ... => 2 ... => 3
where the dots (...) represent any sequence of values (including a null sequence)?

That is correct. There is no syntax for specifying a negated wild-card transition value (the ... in the example).

Also it seems there isn't a way specify **zero** or more intervening transitions. That is, the sequence of sampled values
2 5 2 2
would not match the transition specification "2 [-> 3]" because between the 2nd and 3rd '2', there isn't a "transition that does not contain the value 2". The sequence
2 5 2 2 1 1 2
also does not match because the transitions between the required three '2' values cannot contain a '2'.

Why introduce non-consecutive transitions if only for limited repetitions of a value?

Your analysis is correct.
The non-consecutive transitions are introduced by the assertions language, the coverage specification simply makes use of the same syntax and semantics. We believe that using the same syntax and semantics avoids confusion.

---

6) In attempting to matching transition sequences can matched sequences overlap? Here are some examples:

Given the covergroup

```
 coverpoint val
{
   bins b2 = 2 [ -> 3:5] ;    // 3 to 5 non-consecutive 2s
   bins b3 = 3 [ -> 3:5] ;    // 3 to 5 non-consecutive 3s
   bins b4 = 4 [* 3];         // 3 consecutive 4s
 }
```

Consider the following sequence of 12 sampled values (for **val**)
1 2 3 2 3 2 3 2 3 2 3 1

On the 6th sample, I believe the transition sequence for **b2** is met and it's bit count is incremented. Will the transition sequence for **b3** be met on the 7th sample? Similarly, will **b2** be incremented on the 8th and 10th samples? Will **b3** be incremented on the 9th and 11th samples?

Matched sequences can overlap.
Your analysis is correct.

For the following sequence of 6 sampled values
4 4 4 4 4 4

will the bin **b4** be incremented on the 3rd, 4th, 5th and 6th sample (or just the 3rd and 6th)?

Yes. **b4** will be incremented on cycles 3, 4, 5, and 6.

- Ray