Unraveling the Complexities of Functional Coverage: An advanced guide to simplify your use model

Thomas Ellis
Product Engineer
Mentor, Wilsonville, OR

Rohit Jain
Principal Engineer
Mentor, Fremont, CA

***Abstract*- More than likely, you first dipped your foot into the pool of coverage with code coverage. Code coverage is a very valuable metric which helps you see whether or not the code you have written is being effectively exercised, and can be a good gauge of verification progress and completeness. One major advantage of code coverage is that you do not need to write any extra code to collect it, you simply enable it in your simulator and it is collected automatically. Code coverage however, doesn't ensure the design works correctly, it merely shows that the code you've written was executed. To further improve the quality and robustness of coverage, we need functional coverage. With functional coverage, the goal is to verify that the design meets its requirements, and functions as intended. Unlike code coverage however, which can be derived from the design, functional coverage is not automatic, and requires that a user explicitly create the functional coverage model themselves. For users both experienced to the usage of functional coverage, and those implementing it for this first time, this can be a complicated task. By knowing some of the potential pitfalls ahead of time, you can save yourself anguish down the road.**

I. Introduction

SystemVerilog was not the first language to include the concept of functional coverage, nor was it the first to be difficult to fully understand. There are many intricacies to the language which can greatly affect the results you observe from simulation. Defining simple covergroups is fairly straightforward, however, the complexity greatly increases once you introduce the concept of trying to merge multiple instances of the same covergroup, which can often lead to vastly different outcomes depending on the implementation approach. Understanding them becomes more complicated when you have embedded covergroups and then trying to merge them across various instances in the design. You could potentially end up with covergroups which simply cannot be merged, which makes it impossible to see where you truly are with regards to your coverage goals. You can potentially introduce unnecessary performance overhead, or restrict your debug visibility into the covergroups themselves if they are not created correctly. These types of decisions are often made early on in a project, and in some cases, there is no other recourse to solving the problem then to make extensive changes to the testbench environment. Proper understanding of the way covergroups are created and function is needed to avoid this scenario before it can become a problem.

In this paper we will outline a set of guidelines for writing an unambiguous coverage model. What to do, what not to do, and how.

II. Understanding Covergroup Options

The SV syntax provides a slew of options for customizing covergroups and covergroup instances. Some are obvious, some perhaps not as clear. Furthermore, there are certain interactions between these option sets which can affect how coverage is collected, merged and reported. Here we will distill down how these options work, and how to best use them to get the results you want.

The main options we are going to analyze, as well as an example covergroup which employs them follows:

```
module top;
  covergroup cg (ref int v);
    option.per_instance = 0;
    type_option.merge_instances = 0;
    option.get_inst_coverage = 0;
```

```
    coverpoint v { bins val[] = { [0:1] }; }
  endgroup
  int a, b;
  cg cva = new(a);
  cg cvb = new(b);
  initial begin
    #1; a = 0; cva.sample();
    #1; b = 1; cvb.sample();
    #1; $display("cva=%.2f cvb=%.2f cva+cvb=%.2f",
                 cva.get_inst_coverage(), cvb.get_inst_coverage(),
                 cg::get_coverage());
  end
endmodule
```

Example 1. Covergroup with Default Options

Note in this case all three options are set to their default values of zero, but are defined here explicitly for clarity. If you were to simulate this code as it is written, and then look at the makeup of the covergroups created, you would see what is pictured in Figure 1.

There are two key points to notice in this image. The first is that while two instances of the covergroup were created, there are no instances displayed in the covergroup listed. This is a result of the `option.per_instance` value set above. This option is used to determine whether or not instance specific information will be reported for covergroups.

Figure 1. Covergroup with Default Options

If we were to run the same example, but set `option.per_instance=1`, we would instead get a slightly different image, as seen in figure 2.

With this change, we can now see that all instances are shown. It is important to note that this option ONLY effect reporting, it has no effect on how coverage is collected or calculated. You might think of it as more of a debugging option, it provides you with additional instance level information on your covergroup. The resultant coverage at the covergroup type level however, will not be effected. Furthermore, it should also be noted that while specifying `option.per_instance=1` does require the tool to save instance information into the database, `option.per_instance=0` does not preclude the tool from still saving instance information, however it is not required to do so in that case.

Figure 2. Covergroup with option.per_instance=1

The second key takeaway here is more easily seen via the last screenshot (did we mention using `option.per_instance=1` is a good debug aid?), and that is the way in which the coverage was calculated. The means by which the type, or cumulative coverage is calculated is controlled by the `type_option.merge_instances`. The calculation for each instance is simply, 1/2 bins hit for each instance, results in a coverage percentage of 50%. You will notice the cumulative coverage is also 50% in this case. That is because when `type_option.merge_instances=0`, the LRM dictates that cumulative coverage will be the weighted average of all instances, which in this case will also be 50%.

Another way you can tell that `type_option.merge_instances=0` is being used, is by virtue of the lack of bins under the `cg::v coverpoint` at the type level (notice that it is greyed out). This is because in a weighted average merge, there is not a concept of bins under the coverpoint, this is something that is well illustrated in our example.

If you look closely, you will notice that both instances of the covergroup hit 1 bin in its constituent coverpoint. Furthermore, you can see that they hit different bins; `cva` hits `val[0]` while `cvb` hits `val[1]`. If you were to try and display the merged result under the cumulative coverpoint, what would you list? Obviously, there is no way to represent the weighted average coverage via two bins, which is why bins are not listed under the type coverage when `type_option.merge_instances=0`.

Now, let's take a look at the difference if we change `type_option.merge_instances=1`. Note that in this example `option.per_instance=1` for clarity.

You should notice one big difference here, and that is that the cumulative coverage of the covergroup is now 100%. In addition, you will also see that there are bins under the cumulative coverpoint. With the change of `type_option.merge_instances=1`, the tool now calculates the cumulative coverage as a union of coverage of all instances. Even though both coverpoints only had 50% coverage, combined, they hit both bins, resulting in 100% cumulative coverage. Determining what the correct value for `type_option.merge_instances` is will vary from case to case, however, the crucial takeaway here is that it can have a drastic effect on how cumulative coverage is calculated for your covergroups.

Covergroups

Name	Class Type	Coverage	Goal
/top		100.0%	
TYPE cg		100.0%	100
CVP cg::v		100.0%	100
bin val[0]		1	1
bin val[1]		1	1
INST \/top/cva		50.0%	100
CVP v		50.0%	100
bin val[0]		1	1
bin val[1]		0	1
INST \/top/cvb		50.0%	100
CVP v		50.0%	100
bin val[0]		0	1
bin val[1]		1	1

Figure 3. Covergroup with type_option.merge_instances=1

The final option we will take a look at here is `option.get_inst_coverage`. You probably noticed it in the original example, but the reason we've waited till now to bring it up is because it is directly linked with `type_option.merge_instances`, in that `get_inst_coverage` only applies when `merge_instances` is set. Let's take a look at how this options works.

SystemVerilog provides two methods for printing out coverage, `get_coverage()` and `get_inst_coverage()`. In the following scenario, we are going to look at the differences between the two when calling them on a covergroup (though they can be used with coverpoints and crosses as well). When `option.get_inst_coverage=0` (default), then both of these functions will return the same value, as can be seen here in Figure 4.

```
# cva=100.00 cvb=100.00 cva+cvb=100.00
```

Figure 4. option.get_inst_coverage=0

Since we are using the same options as the last example, you would probably expect that the instance coverage returned would be 50%, however, you can see here that unless `get_inst_coverage` is set (and `merge_instances` is set), you will not enable instance data through this built-in method.

If we change the code such that `option.get_inst_coverage=1`, you can see that we get the instance values, as seen in Figure 5.

```
# cva=50.00 cvb=50.00 cva+cvb=100.00
```

Figure 5. option.get_inst_coverage=1

Here you can see the instance coverage reported as the expected 50%. A complete listing of how these options and methods interact with one another can be found in the following table.

Table 1. Summary of Covergroup Options

`type_option.merge_instances`	`option.get_inst_coverage`	`get_inst_coverage()`	`get_coverage()`
0	0 | 1	Average of all instances	Average of all instances
1	0	Merge of all instances	Merge of all instances

1	1	Individual instance	Merge of all instances

Use this table to ensure you are using the proper settings to achieve your desired outcome from these methods.

When it comes to what is the best set of options to use, it can be difficult to make a blanket statement for all scenarios. However, speaking generally, the following will typically be the most resourceful settings:

```
option.per_instance=0

type_option.merge_instances=1
```

The reason for this is that it limits the amount of data you need to carry forward through merging, reporting and analysis. As mentioned, previously, there are times when you really will need per-instance coverage, whether it be for debugging purposes, or to answer a particular question about your coverage, and in those situations you will need to change the settings accordingly.

One other point to keep in mind here, is that some simulations offer the ability to globally set the value of `option.per_instance` and `type_option.merge_instances`, for any covergroup which did not have it explicitly set in the code. This can be very helpful for times when you want to quickly enable per-instance coverage (remember it is off by default) for a debugging task, but don't want to modify your source in multiple areas.

Another aspect to touch upon here is performance, both runtime and memory consumption, while using these options. For embedded covergroups (covergroups defined inside a class object), using `option.per_instance=1` or `type_option.merge_instances=0` may cause simulation to retain all covergroup objects being created during the life of simulation to produce the expected behavior of these options, even though its parent class object may have been used and then deleted. In such cases, class objects will be garbage collected and recreated when user asks to create one again, but embedded covergroup objects are always retained and continue to add up as and when the user creates new class objects. This behavior may be obvious to the user, but it can cause not only substantial memory and performance overhead during simulation, but also performance and disk overhead in maintaining and processing large amounts of data post-simulation. Hence, in order to keep your flow efficient, we recommend using these options only when you really need them.

III. Naming Your Covergroups

One bit that often gets users tripped up when it comes to embedded covergroups, is that of instance names. When you are not using embedded covergroups, the name of the instance is supplied when you construct (or new) the covergroup. This has been the case in previous examples. However, when that covergroup is embedded in a class, you now have to do a little extra work to name that covergroup instance.

Why do you care? Well, honestly in some cases it may not be a big deal, particularly if you will only ever have one instance of a class instantiated. However, if you intend to have more than one instance of that class (and by association that covergroup), then you start to run into some issues. Regardless of that, it is just a good practice to get in the habit of. You see, covergroup instances are required to have a unique identifier to differentiate them, which is the covergroup name. Furthermore, if you do not specify one, the LRM dictates that a unique name will be automatically generated for you by the tool [3].

Take the following covergroup as an example:

```
package p;
  class c;
    int i;
    covergroup cg;
      coverpoint i { bins ival[] = { [0:1] }; }
    endgroup
    function new();
      cg = new;
```

```
    endfunction
    function void sample(int val);
      i = val;
      cg.sample();
    endfunction
  endclass
endpackage
module top;
  p::c cv = new;
  p::c cv2 = new;
  initial begin
    ...
  end
endmodule
```

Example 2. Multiple Covergroup Instances

In this example you can see we have two instantiations of the class c. When the first instance of the class in created, `cv`, a covergroup cg is created via the new function. When the second instance of the class is created, `cv2`, the same will happen, only this time the covergroup cannot also be named cg, so the tool will add a unique string to ensure a unique covergroup instance. The following Figure shows this in the Questa GUI:

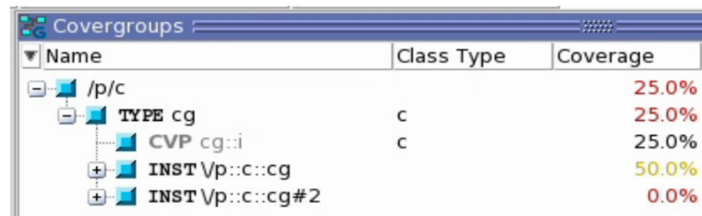

Figure 6. Automatically Generated Covergroup Instance Names

Notice a `#<val>` is inserted for subsequent instances of the covergroup. While this accomplishes the need of unique names, if you need to do any analysis at the instance level, it can make it difficult to differentiate the two instances. The solution is to ensure that the covergroup instances each get unique names, which can easily be done by passing in a name when creating the class instance. Here is a modified example of the above code to do just that:

```
package p;
  class c;
    int i;
    covergroup cg(string o_name);
      option.name = o_name;
      coverpoint i { bins ival[] = { [0:1] }; }
    endgroup
    function new(string cvg_name);
      cg = new(cvg_name);
    endfunction
    function void sample(int val);
      i = val;
      cg.sample();
    endfunction
  endclass
endpackage
module top;
  p::c cv = new(“foo”);
  p::c cv2 = new(“bar”);
  initial begin
    cv.sample(0);
    $display($get_coverage());
  end
endmodule
```

Example 3. Explicitly Named Covergroup Instances

Here you can see that we name the two instances `foo` and `bar` respectively in our code. Furthermore, you can see how those names get applied and displayed in the GUI below:

Covergroups

Name	Class Type	Coverage
/p/c		25.0%
TYPE cg	c	25.0%
CVP cg::i	c	25.0%
INST foo		50.0%
INST bar		0.0%

Figure 7. User Named Covergroup Instances

While it isn't a requirement to name each instance, it is something that is very easy to do, and can offer a lot of clarity that auto generate tool names do not provide. In general, it is a good habit to get into, you will surely thank yourself later.

IV. Cross Bin Expansion

Another bit which is important to understand when it comes to covergroups, is that of default cross-bin expansion. Without care, the size of these crosses can quickly get out of control. When a user creates a cross, but doesn't specify explicitly what bins need to be created, tools usually have the capability to auto-expand all possible combinations of cross-bins. This is desirable in many cases as the user can avoid creating explicit list of all combinations.

However, it can be problematic in some cases where auto-expansion results in thousands or even millions of cross-bins. The user most likely did not really require such a large expansion, but it is created anyways due to tool's auto-expand feature. This makes post-processing of coverage results difficult, as a user must deal with such a huge number of cross-bins. Additionally, this can create a large impact in terms of performance and memory overhead. Behavior that is typically not desired by the user. In these cases, users should be aware of the potential for a large expansion of bins and provide explicit list of cross-bins they would like to measure, which is usually a smaller and more manageable list of cross-bins.

So, we have seen both cases, where auto-expansion is desirable and where auto-expansion is not desirable. Tools also provide control settings to say whether the user wants auto-expansion or not. This provides the ability to turn off/on auto-expansion completely on all crosses, again something user may not desire.

A balanced approach could be used here. User can ask tool to auto-expand in all cases, except when they are explicitly providing a list of user-defined cross-bins. This lets user control both scenarios in the context of the same simulation, where they would like to have auto-expansion and where they would not want auto-expansion. An example of this approach can be seen in Example 4.

```
covergroup cvg;
  a_cov: coverpoint A iff (start) {
    bins a_1 = {1};
    bins a_2 = {2};
    bins a_3 = {3};
  }
  b_cov: coverpoint B iff (start) {
    bins b_1 = {1};
    bins b_2 = {2};
    bins b_3 = {3};
  }
  c_cov: coverpoint C iff (start) {
    bins c_1 = {1};
    bins c_2 = {2};
    bins c_3 = {3};
  }
```

```
  a_b_cross :cross a_cov, b_cov{
    bins a_1_b_1 = binsof(a_cov.a_1) && binsof(b_cov.b_1);
    // don't auto-expand cross-bins here
  }
  a_c_cross :cross a_cov, b_cov;
    // auto-expand cross-bins here
endgroup
```

Example 4. Cross-bin Expansion

Industry standard simulators usually provide different controls for the behavior here. Please check with your tool provider for any specifics.

V. SUMMARY OF GUIDELINES

As we have eluded to several times in this paper, it is difficult to give across the board recommendations on what the best approach is, as users verification goals may differ slightly, however, the following list of guidelines should generally apply to anyone employing the use of SV covergroups.

1. For covergroup options, the most efficient settings will be:
    a. `option.per_instance=0`
    b. `type_option.merge_instances=1`
2. Get in the habit of always naming your covergroup instances.
3. Explicitly define a list of cross bins whenever possible, rather than relying on a tools auto-expansion features.

VI. SUMMARY

Covergroups are an incredibly powerful component of verification, however, they can be a bit difficult to set up properly. By following these recommendations, and armed with a better understanding of the effect covergroup options have on the results of verification, you can be sure to get the most out of your verification.

REFERENCES

[1] SystemVerilog Coding Guidelines: Package import versus `include, https://blogs.mentor.com/verificationhorizons/blog/2010/07/13/package-import-versus-include/

[2] Using SystemVerilog Packages in Real Verification Projects, https://www.mentor.com/products/fv/verificationhorizons/horizons-jun-10

[3] ANSI/IEEE 1800-2012 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, https://standards.ieee.org/findstds/standard/1800-2012.html