A Simplified Approach to Generating Functional Coverage

Aditya Pagonda
Neil Bulman

Broadcom
Cambridge, UK

www.broadcom.com

ABSTRACT

In the process of verifying a design, SystemVerilog functional coverage is commonly used to define target metrics that must be covered. Functional coverage may be defined for register accesses, transaction members of a class, interface scenarios, or combinations among them. The UVM register model generated by RALgen has some functional coverage that addresses the register space. The more interesting functional coverage items are those that have to be manually coded and can take days to get right. This paper discusses a solution to automatically generate functional coverage items from user requirements.
The paper provides a solution that will:

- *Use RALgen-generated and transaction-class definitions.*
- *Generate coverage items on register fields within a design block or between design blocks.*
- *Generate coverage items on transaction members of a class.*
- *Make it easy to include and exclude fields, bins, etc.*

Table of Contents

List of Figures

1. Introduction

Functional coverage is a key metric in determining device verification progress and completeness. Unlike code coverage, which effectively comes for free from the simulation tools, functional coverage must be defined.

Since code coverage requires no effort, some may conclude that functional coverage is not required. There are two major problems with this conclusion. First, code coverage only indicates what was implemented, not what is needed. Code coverage won't indicate whether a required functionality was implemented. Second, code coverage indicates only what code was exercised, not that anything useful was done or that functionality was checked. For example, someone may choose to toggle a reset signal at the end of a test. In this case, code coverage shows that the reset signal was toggled, and any code dependent on the reset signal state-change was exercised. It won't show whether the desired effect was achieved, or whether the design does anything useful in response to the signal that toggled.

Functional coverage, on the other hand, is tied to the expected design functionality, and not to how the design is implemented. In functional coverage, users control the sampling points. As a result, it is possible to sample the coverage only when a check has taken place or, depending on strategy, after the check has passed.

The approach discussed in this document can be applied equally to both block-level and top-level test benches.

2. Functional Coverage Challenges

Although the value of functional coverage is clear, it has one major issue. Implementing a coverage model requires human labor and has its challenges.

The first challenge: defining the coverage model as well as the language and syntax of covergroups, coverpoints, and crosses can be daunting. For the less experienced, a lot of time can be spent performing multiple compilations to clean up syntax issues. For the more experienced, defining the coverage may not be too daunting, but remembering what constructs are available over time is often difficult.

After creating a coverage model, the second challenge is determining whether enough coverpoints have been defined and whether the defined bins are sensible. Although a coverage model that yields 100% coverage in time for tape-out may feel like progress, it doesn't indicate that anything useful is being covered. The simplest way to achieve this is to review the defined coverage. This is where the challenge really lies. Providing a bunch of SystemVerilog files with coverage definitions is typically not helpful to those, other than verification engineers, involved in a project. As a result, providing the SystemVerilog files leads to resistance or a poorly done job.

Additionally, it can be difficult to find where all the coverage definitions exist. Making a scary situation worse, the definitions can be included in files with thousands of lines, possibly leading to confusion and oversights.

It is possible to ask the question: "I've generated my register model using RALgen, doesn't its coverage model address my needs?" Unfortunately, it probably doesn't. Although parts of the register model may be sufficient for some verification needs, it is not sufficient to address the entire coverage model. While RALgen does appear to support defining coverage information in the RAL file, it does not define coverage for nonregister items. It still leaves you with your planned coverage across several places. Additional issues with RALgen are:

- It restricts crosses to fields within the same register, and sampling is done only on a per-register basis.
- It does not allow sampling a single field within a register.
- It does not allow sampling all fields in the register model at once.
- It does not allow the crossing of fields that are across different registers, or even blocks.
- There may be aspects of the design functionality that are not completely captured in the register model; for example, what the data traffic looks like.

From the previously mentioned issues, there are two major stumbling blocks:

- The technical challenges in defining an appropriate coverage model.
- The motivational challenges:
  - If it's too difficult to create, people will not bother or give up.
  - If it's too difficult to review and understand, people won't bother reviewing, bringing into doubt the quality of the coverage model being used for sign-off.

3. Simplifying Functional Coverage

The starting point to address the functional coverage challenges was looking into the register-based coverage. For the devices being verified, device control largely comes from the registers, making the registers prime candidates to consider for simplifying functional coverage.

Within Broadcom, an internal format is used to define register maps, and tools exist to generate the required design-flow files. This includes HTML pages for documentation, RTL files for the hardware design, and C files for the software development. As the internal format is used to generate the other design-related files, it makes sense to use it as the source of the register model. Using the same format keeps the register model in line with the other views of the design.

```
regtype32         Type_OUTPUT_CTRL
   desc           output control register.
      field       reserved                   31:16
      field       MAX_BURST                  15:13 unsigned
         desc     Maximum burst length
                  default  3h
      field       reserved                   12:01
      field       CLEAR                      00:00 unsigned
         desc     This bit clears all states
                  in output block.
         default  0h
endtype

regtype32         Type_OUTPUT_STATUS
   desc           output status register.
      field       reserved                   31:14
      field       DONE                       13:13 unsigned
         desc     When 1 it indicates that there are no
                  more outstanding transactions.
         default  1h
      field       reserved                   12:08
      field       OUTSTANDING_WRITES         07:00 unsigned
         desc     This field shows the number of
                  outstanding write transactions.
         default  0h
endtype

regset            Regs
   private        ENGINEERING_ONLY
   title          Registers for interface block

   Type_OUTPUT_CTRL        OUTPUT_CTRL      +000h RW
      title                  OUTPUT Control

   Type_OUTPUT_STATUS      OUTPUT_STATUS    +004h RO
      title                  OUTPUT Status
endregset


blockdef TOP_BLOCK
   regset Regs       OUTPUT_REGS       +00100000h
endblockdef
```

Figure 1: Register Definition Example

The decision was made to use RALgen to generate the register model. Using RALgen simplifies the task because RAL is a very simple format that is easy to generate. The first step was to extend this automation to generate a RAL file that allowed RALgen to generate the register model. This raised the challenge of determining how to manage crosses across different registers inside the register model.

The approach taken was to create a flattened view of the register model. The key aim of the flattened view was to provide a live view of what the register model currently looked like, without the need for any synchronization. To achieve this, a class extending uvm_object was created. The members of the uvm_object are:

- A handle for the register model.
- A handle for all of the fields contained within the register model.

The handles to the register-model fields must be unique. As expected, this is generally not possible to achieve using just the field name, so the names were extended to also include the block and register. Doing this helps remove the ambiguity about which field is actually being referred to. The drawback of creating a flattened view of the register model is duplicating the functional coverage definition for multiple instances of the same register or block. This drawback can be handled by updating the scripts that generate the functional coverage class.

Inside the new function, each of these handles is set to point to its relevant uvm_reg_field in the register model.

```
class output_regs extends uvm_object;

  ral_block_OUTPUT output_reg_mdl;

  uvm_reg_field Regs_OUTPUT_CTRL_MAX_BURST; // #num_bits#3#0#
  uvm_reg_field Regs_OUTPUT_CTRL_CLEAR; // #num_bits#1#0#

  uvm_reg_field OUTPUT_STATUS_ DONE; // #num_bits#1#0#
  uvm_reg_field OUTPUT_STATUS_OUTSTANDING_WRITES; // #num_bits#8#0#

  function new(string name = "output_regs");
    super.new(name);

    if(!uvm_config_db#(ral_block_OUTPUT)::get(uvm_root::get(), "*",
                                    "output_reg_mdl", output_reg_mdl))
      `uvm_fatal(get_type_name(), "reg_model not found")

    //Regs;
    this.Regs_OUTPUT_CTRL_MAX_BURST = output_reg_mdl.Regs_OUTPUT_CTRL.MAX_BURST;
    this.Regs_OUTPUT_CTRL_CLEAR = output_reg_mdl.Regs_OUTPUT_CTRL.CLEAR;

    this.Regs_OUTPUT_STATUS_DONE = output_reg_mdl.Regs_OUTPUT_STATUS.DONE;
    this.Regs_OUTPUT_STATUS_OUTSTANDING_WRITES =
                      output_reg_mdl.Regs_OUTPUT_STATUS.OUTSTANDING_WRITES;
  endfunction : new
endclass : output_regs
```

Figure 2: Example of a Generated Register Wrapper

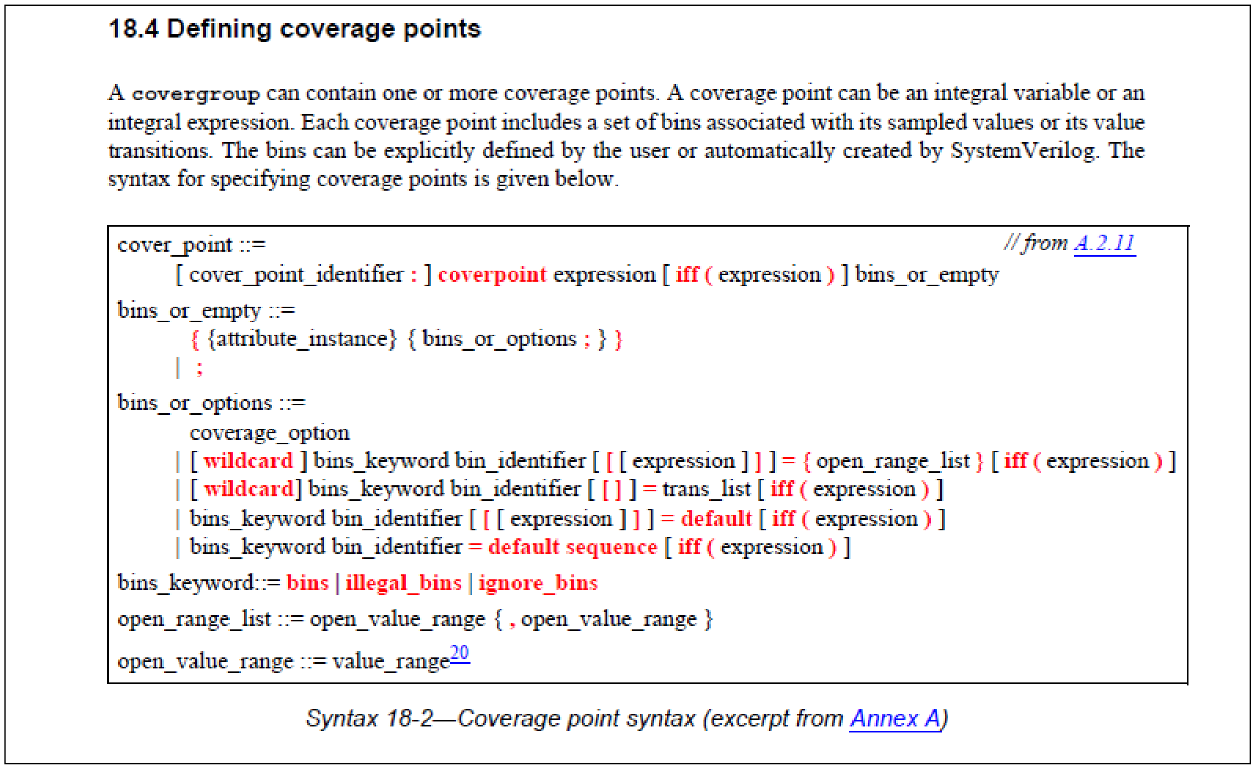
An additional file must be created. Fortunately, the additional file can be generated in the same way as the RAL file is generated from the register-model definition.

Having addressed the issue of being able to view the register model fields in one place, it's now possible to start considering the coverage. The next step is to extend the generated class with its field pointers.

The starting point to this is defining an input format for identifying the required coverage. An Excel® spreadsheet was used to accomplish this. Using an Excel spreadsheet has the following advantages:

- It's a user-friendly interface for defining and reviewing the coverage definition.
- It's a program that all members of staff have access to.
- An Excel parser is readily available for Perl.
- It supports additional formatting to assist data entry and review.

Having selected a format for entering the coverage, the next step was to define a data-entry template. Section 18.4 of the SystemVerilog standard was used to develop the data-entry template.



18.4 Defining coverage points

A **covergroup** can 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 can be explicitly defined by the user or automatically created by SystemVerilog. The syntax for specifying coverage points is given below.

```
cover_point ::=                                                              // from A.2.11
       [ 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[20]
```

Syntax 18-2—Coverage point syntax (excerpt from Annex A)

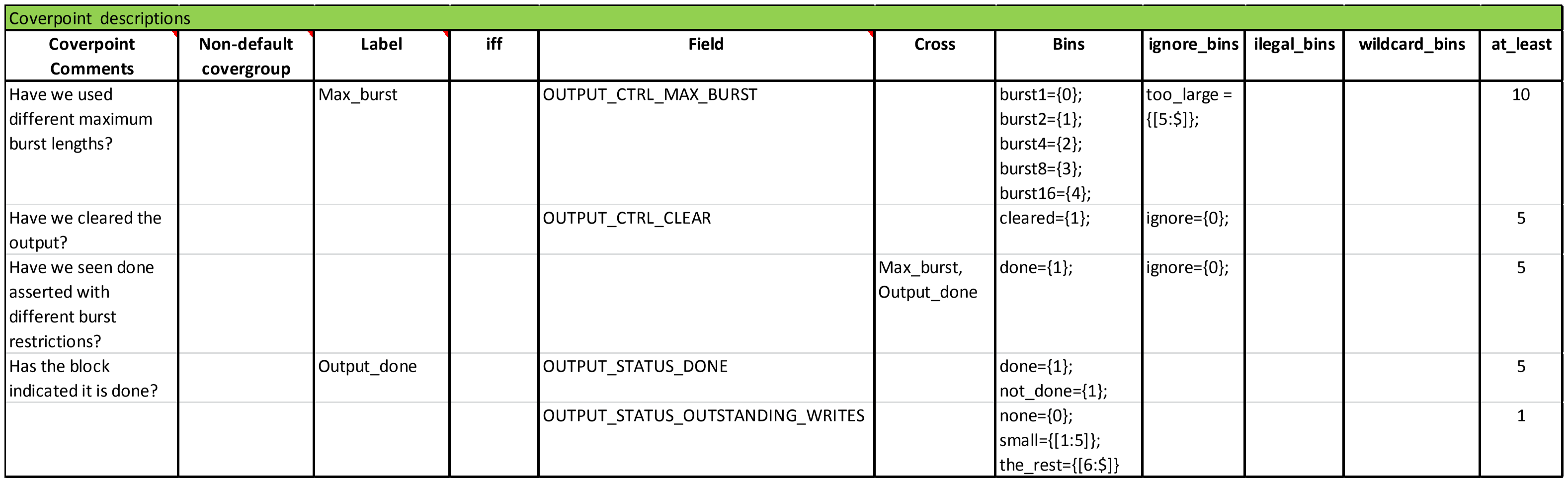
Figure 3: SystemVerilog Coverpoint Definition

Using Section 18.4 of the SystemVerilog standard as a guide, a separate column was defined for each bin type, plus an additional column was defined per bin type to ease the definition of wildcard bins. Additional columns were added for:

- Iff.
- The field to be covered.
- The label to be used for the coverpoint.
- Determining which fields are to be used if the item is referring to a cross, rather than just a coverpoint.
- The covergroup to which the coverpoint/cross belongs.
- Any options to be used for the coverpoint, such as:
  - At_least
  - Comment

The crosses were defined seperately from the coverpoints to make it easier for designers to understand during their planned functional coverage reviews.

Figure 4 shows an example of the spreadsheet used to enter the coverage.



Coverpoint descriptions										
Coverpoint Comments	Non-default covergroup	Label	iff	Field	Cross	Bins	ignore_bins	ilegal_bins	wildcard_bins	at_least
Have we used different maximum burst lengths?		Max_burst		OUTPUT_CTRL_MAX_BURST		burst1={0}; burst2={1}; burst4={2}; burst8={3}; burst16={4};	too_large = {[5:$]};			10
Have we cleared the output?				OUTPUT_CTRL_CLEAR		cleared={1};	ignore={0};			5
Have we seen done asserted with different burst restrictions?					Max_burst, Output_done	done={1};	ignore={0};			5
Has the block indicated it is done?		Output_done		OUTPUT_STATUS_DONE		done={1}; not_done={1};				5
				OUTPUT_STATUS_OUTSTANDING_WRITES		none={0}; small={[1:5]}; the_rest={[6:$]}				1

Figure 4: Functional Coverage Definition Example

Having created the format for defining the coverage model, the next step was to generate the model.

The steps to do this are:

1. Parse the list of fields and registers. For simplicity, this parses the wrapper previously generated rather than reparsing the register definition.
2. Parse the spreadsheet that defines the coverage.
3. Loop through crosses and check that any referenced fields exist as coverpoints. If not, add them to the list of coverpoints to be created.
4. Check that all the fields referred to in coverpoints actually exist in the register model.
5. Loop through all the referenced covergroups. When no covergroup is defined, it is inferred to be the default covergroup.
   a. Write out the coverpoints.
   b. Write out the crosses.

To document where coverpoints/crosses were defined, a SystemVerilog comment was added to each coverpoint/cross while writing out the functional coverage model. Each comment was simply a reference to the cell in the entry spreadsheet that caused the coverpoint to be created. The reasoning behind doing this was to provide an audit trail, making code review easier.

```
import vip_axi_trans_pkg::vip_axi_trans;

class top_vip_axi_trans_cov extends vip_axi_trans;

  covergroup cg_vals;
    option.per_instance = 1;
    option.at_least = 10;
    PRIVILEGED : coverpoint is_privileged;
  endgroup

  function new(string name = "top_vip_axi_trans_cov");
    cg_vals = new();
  endfunction

  `uvm_object_utils(top_vip_axi_trans_cov)

  function void sample_values(vip_axi_trans axi_trans);
    this.copy(axi_trans);
    if (cg_vals != null) cg_vals.sample();
  endfunction

endclass
```

Figure 5: Generated-Transaction Functional Coverage Class Example

During initial trials, it was determined that more information was required when generating the coverage model. This additional information included:

- Whether the value stored in the field was signed or unsigned.
- The number of bits used by a field.

Unfortunately, the generated register model doesn't appear to discriminate between signed and unsigned values. When reviewing coverage and defining bins, the sign information helps to avoid confusion.

Regarding the number of bits, the default UVM register is 64 bits but can be overriden by UVM_REG_DATA_WIDTH. For many registers, this is much larger than required, which can lead to the following undesirable outcomes:

- All interesting values can get placed in the same bin, making it difficult to see whether all necessary conditions are being met.
- Some bins will be empty.

4. Extending to Use Transaction Classes

The problem with the approach described so far is that it only helps to cover items contained within the register model. Unfortunately, not everything that needs to be covered is in the register model. Some of the items to be covered may be in transaction classes; for example, data used in the scoreboard.

As a format for defining coverage had already been defined for register-model-based coverage, it was decided that the same format should also be used to describe any other coverage. This keeps things simple in terms of definition and review because only a single format needs to be understood.

The challenges now are:

- Where does the data to be covered actually exist. We can no longer rely on a pointer to the register model because the data we want isn't there.
- Where to put the functional coverage.

If the coverage had been defined by hand, it could have been included in the transaction class. Unfortunately, this doesn't help address the challenges intially identified:

- When defining the coverage.
- When reviewing the defined functional coverage.

The approach taken was to generate a new class that extends the transaction class. The advantages of doing this are:

- It‘s easier to identify what coverage is defined; it’s in a file with functional coverage and nothing else. There are no issues with the coverage being defined in different places in different files.
- The original class doesn’t need to be modified.
- The extended classes containing the coverage definitions can be put in a single location.

Defining the coverage in a new class leads to the following questions:

- How is the coverage sampled?
- How do we ensure we are covering the values we expect?

One approach is to use the factory overrides to change the original instances of the class to the extended version containing the functional coverage definition. This still leaves the issue of sampling the coverage. It also leaves the functional coverage spread across different parts of the test-bench hierarchy.

The approach taken was:

1. To create a class that contains instances of all the required extended classes containing the coverage.
2. To add analysis ports that allow the covered transactions to be passed to the new class.
3. Create an analysis port in the relevant test-bench component (for example, the scoreboard) that writes the transaction to be covered at the appropriate point in time (for example, after a successful check).
4. Connect the analysis ports.

As the coverage class is not connected to the transaction being covered, we still need to ensure that we cover the correct values. To address this, the sample function used in the coverage class does two things. The first is to copy the incoming transaction to itself. This addresses the issue of covering the correct values. The second is to actually do the sampling. An example of the scheme used is shown in Figure 6.

```
`ifndef TOP_COV
`define TOP_COV

//analyis ports to get objects to collect coverage for
// AXI
`uvm_analysis_imp_decl(_axi)

class hevd_top_cov extends uvm_component;

  top_vip_axi_trans_cov       axi_cov;
  hevd_cov                         reg_cov;

  `uvm_object_utils(top_cov)

  uvm_analysis_imp_axi #(vip_axi_trans, top_cov) axi_cov_port;

  function new(string name = "top_cov", uvm_component parent = null);
    super.new(name, parent);

    axi_cov = new("axi_cov");
    axi_cov_port = new("axi_cov_port", this);

    reg_cov = new("reg_cov");
  endfunction: new

  function void write_axi(input vip_axi_trans axi_recd);
    axi_cov.sample_values(axi_recd);
  endfunction

  function void sample_reg_cov();
    reg_cov.sample_values();
  endfunction

endclass : top_cov

`endif
```

Figure 6: Top-Level Coverage Class Example

The original approach used for generating the coverage class for the register model can be used again, but with some minor modifications. It is no longer possible to check whether each entry in the spreadsheet refers to a valid item because we no longer have the register definitions to conclusively define what is valid to cover. Secondly, information regarding the class being extended needs to be passed to the generation script. As a result, it was very easy to extend the original script to generate both coverage groups. The results are a consistent format for entry and review, and the generated coverage looks consistent.

5. Conclusions

The approach described in this paper successfully lowers some functional coverage barriers. Three major challenges were addressed to more easily get full value from SystemVerilog functional coverage. The first challenge was how to cover fields spread across different registers. The second challenge was how to increase the effectiveness of reviewing the functional coverage. The final aspect was how to lower the barriers involved in generating the functional coverage.

It can be easily seen that the first challenge has been addressed; coverage of fields is no longer restricted to just fields within the same register. We now have an approach that gives a lot more flexibility to what can be covered within the register model. In addition, the register model structure no longer causes confusion as to what can be defined as a cross.

The other two challenges are a little more subjective in defining their success, but it can be seen that the approach described removes some of the barriers used to get the most from functional coverage. The coverage definition is no longer in a SystemVerilog file, so the common argument that designers don't understand SystemVerilog and, therefore, can't review the planned coverage points is no longer valid. In addition, by having a reusable template, it's no longer pushed to those with SystemVerilog knowledge to define the coverage. For those with knowledge of SystemVerilog, the template provides an easy reminder of the options for defining coverage.

While the approach described has been successful so far, additional improvements are possible.

It would be desirable to parse the transaction classes to allow the functional coverage definition to be checked before creating the coverage class. This would remove the need to compile the test bench in order to discover the functional coverage definitions that refer to nonexistent class members.

Another improvement would be to add support to allow a covergroup and, more importantly, a cross to include items both in the register model and in a transaction class.

A final improvement would be to look at integrating the functional coverage definitions with the test plan. This would have the benefit of adding the direct link between the test plan item and the functional coverage item that indicates that this has been met.

References

[1] IEEE 1800 standard

[2] Synopsys RALgen user guide