2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA

Functional-Coverage Sampling in UVM RAL

*Use of 2 Obscure Method*s

Muneeb Ulla Shariff, Mirafra Technologies Pvt Ltd, Bangalore, Karnataka, India
(muneebullashariff@mirafra.com)

Ravi Reddy, Roche Sequencing Solutions, Santa Clara, California, USA
(ravi.reddy@roche.com)

***Abstract*—The Universal Verification Methodology (UVM) Register Abstraction Layer (RAL) is a very powerful feature to model the memory-mapped behavior of the registers and memories in the DUT. Based on the user's input the register-model-generators automatically generate the covergroup for the RAL functional-coverage. Depending on whether the covergroup needs to be sampled automatically on register access or as the result of an external call, two different methods need to be implemented; sample() and sample_values(). Due to the lack of information about these methods, they are rarely and improperly used. Thus, in this paper, the focus is to answer the following questions: which of the 2 methods to be used, when to be used and how to implement. Additionally, the methods are compared and contrasted, and there will be suggestions about which method could be used and their advantages in a given situation.**

***Keywords*—UVM; RAL; Functional Coverage; sample; sample_values**

I. Introduction

The UVM register model is used to mimic the design hardware register contents at the TestBench (TB) side and to abstract accesses to registers and memories. The register model is constructed from the classes that describe the memory regions or registers of the Design Under Test (DUT). These classes encapsulate the bit fields within registers and registers within blocks. The registers and memory blocks are allocated address offsets within an address map model inside the block. The UVM RAL provides tasks, read() and write(), which can be called from a UVM sequence to access the registers. The RAL model is kept up to date with the DUT state, either with the help of auto-prediction or explicit prediction, by using a register predictor component.

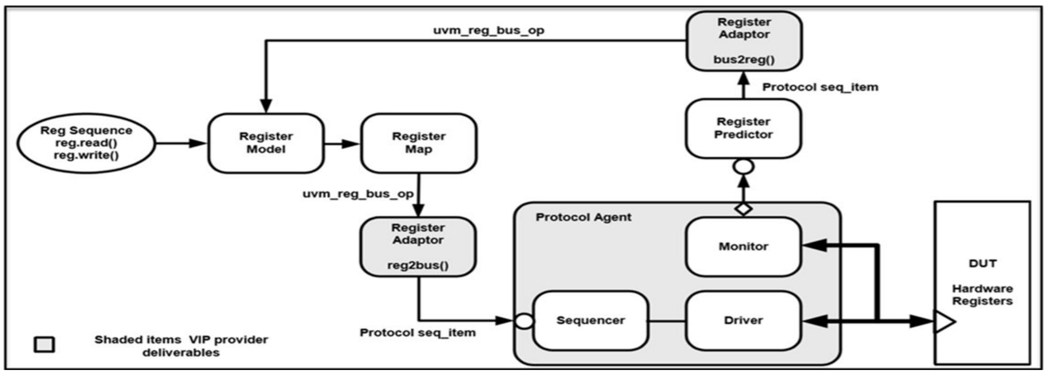


Figure 1. UVM Register Model Integration [1]

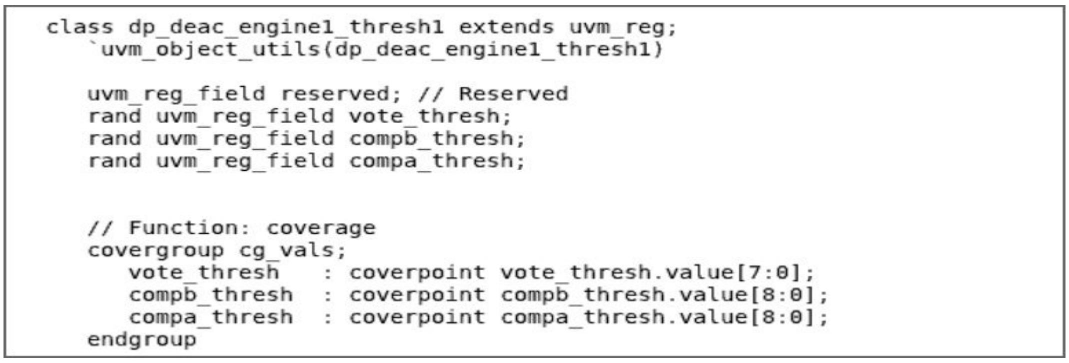
To integrate a register model into a UVM TB, we have to create the register model, implement and use the adapter class, a predictor and hook them into the TB structure. The detailed procedure is explained in the UVM Cookbook[2]., and in the UVM User's Guide[3][4]. Figure. 1 is a pictorial representation of the TB integration of the UVM register model.

The functional coverage of the RAL model is usually created by the register model generators. However, the sampling of the covergroup requires attentive work.

II. Procedure

In order to sample the RAL functional coverage the following steps have to be followed:

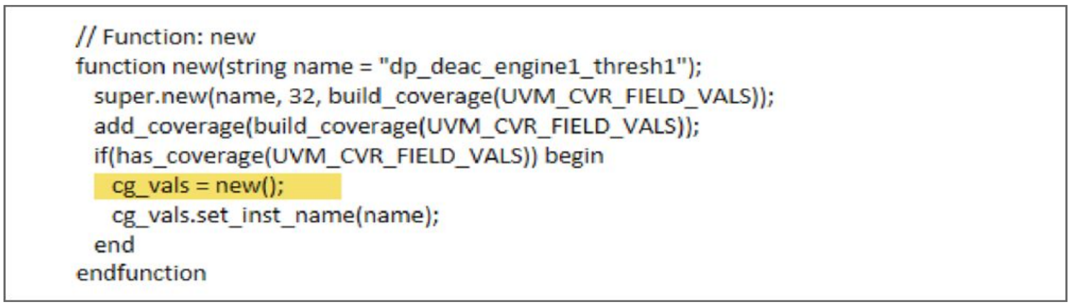
1. The covergroup and coverpoints must be defined. This is done using the register assistant tools, as shown in Figure 2.



```
class dp_deac_engine1_thresh1 extends uvm_reg;
   `uvm_object_utils(dp_deac_engine1_thresh1)

   uvm_reg_field reserved; // Reserved
   rand uvm_reg_field vote_thresh;
   rand uvm_reg_field compb_thresh;
   rand uvm_reg_field compa_thresh;


   // Function: coverage
   covergroup cg_vals;
      vote_thresh    : coverpoint vote_thresh.value[7:0];
      compb_thresh   : coverpoint compb_thresh.value[8:0];
      compa_thresh   : coverpoint compa_thresh.value[8:0];
   endgroup
```

Figure 2. Covergroup definition

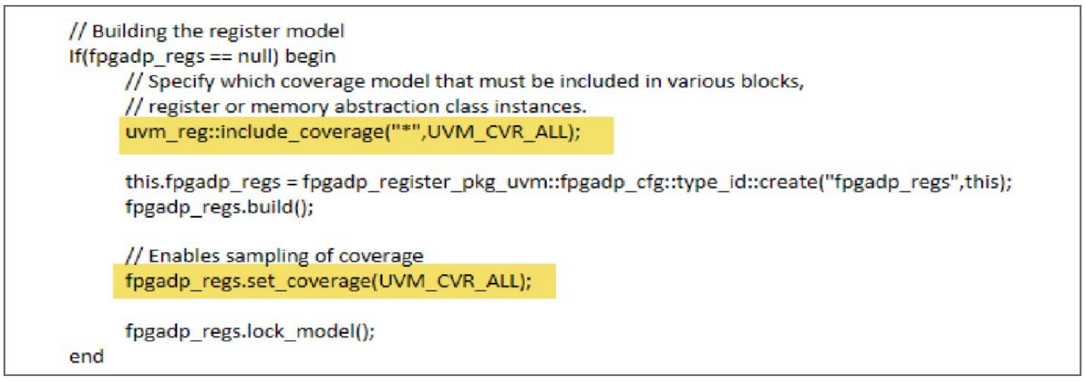
2. The coverage model needs to be constructed conditionally, as shown in Figure 3.



```
// Function: new
function new(string name = "dp_deac_engine1_thresh1");
  super.new(name, 32, build_coverage(UVM_CVR_FIELD_VALS));
  add_coverage(build_coverage(UVM_CVR_FIELD_VALS));
  if(has_coverage(UVM_CVR_FIELD_VALS)) begin
    cg_vals = new();
    cg_vals.set_inst_name(name);
  end
endfunction
```

Figure 3. Covergroup construction [5]

3. Before building the reg model you need to set uvm_reg::include_coverage(...) to indicate which models to be constructed, as depicted in Figure 4.

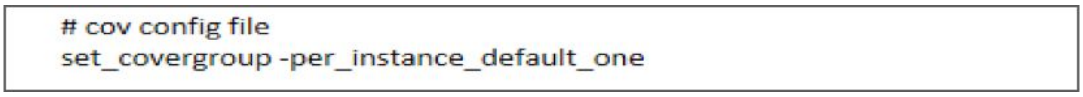


```
// Building the register model
If(fpgadp_regs == null) begin
        // Specify which coverage model that must be included in various blocks,
        // register or memory abstraction class instances.
        uvm_reg::include_coverage("*",UVM_CVR_ALL);

        this.fpgadp_regs = fpgadp_register_pkg_uvm::fpgadp_cfg::type_id::create("fpgadp_regs",this);
        fpgadp_regs.build();

        // Enables sampling of coverage
        fpgadp_regs.set_coverage(UVM_CVR_ALL);

        fpgadp_regs.lock_model();
end
```

Figure 4. Enabling building and sampling of coverage [5]

4. Eventually, you need to tell the compiler to enable coverage collection (The below options qualifies for Cadence Incisive Simulator)

   -uvm -write_metrics -covfile cov_config_file -coverage All



```
# cov config file
set_covergroup -per_instance_default_one
```

Figure 5. The contents of cov_config_file

5. Finally we need to sample the coverage using the 2 methods, uvm_reg::sample() and uvm_reg::sample_values(). We need prediction to update the RAL model and based on either auto-prediction mode or explicit-prediction mode, the uvm_reg::sample() or uvm_reg::sample_values() methods are used and implemented.

III. Prediction

In UVM Register Modelling, a prediction is an art of keeping the Register Model up-to-date with expected results for the design registers. This allows us to compare the expected results from the Register Model with actual register values from the DUT.

A. *Auto-Prediction Mode (implicit)*

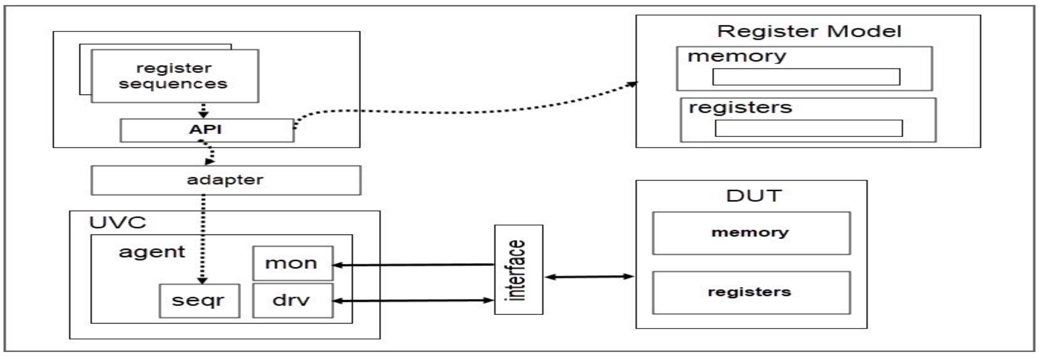
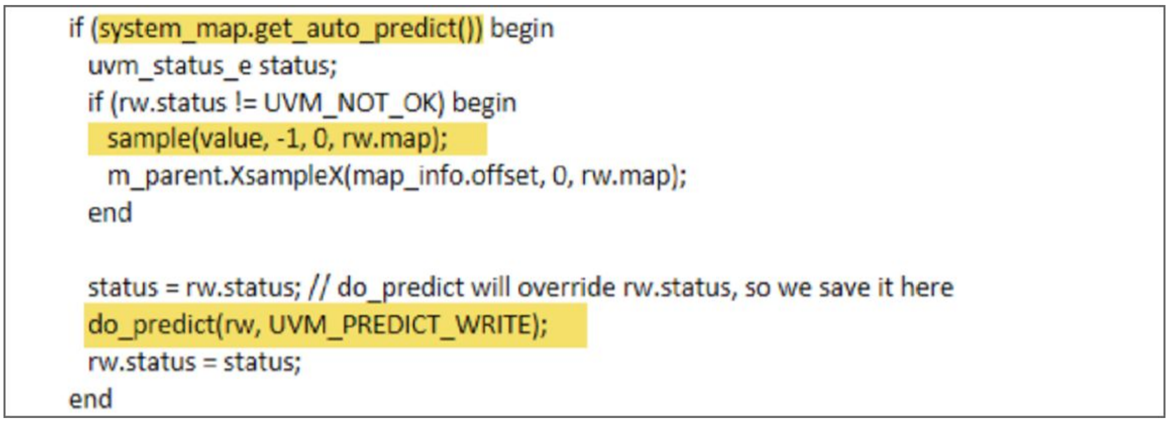


Figure 6. Auto-Prediction Model

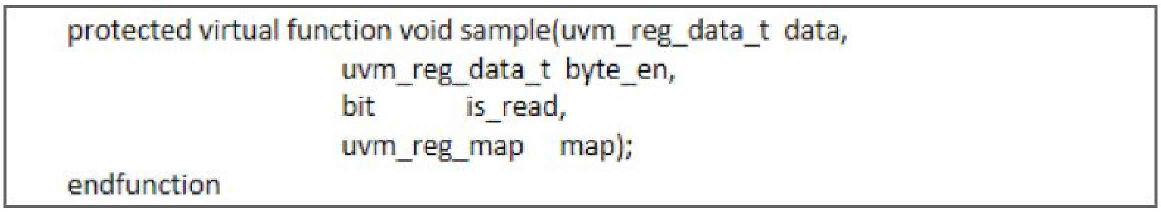
In this prediction mode, the sequences using the UVM register API update the RAL model automatically. On every register access, the uvm_reg::sample() method is called, as shown in Figure 7.



```
if (system_map.get_auto_predict()) begin
  uvm_status_e status;
  if (rw.status != UVM_NOT_OK) begin
    sample(value, -1, 0, rw.map);
    m_parent.XsampleX(map_info.offset, 0, rw.map);
  end

  status = rw.status; // do_predict will override rw.status, so we save it here
  do_predict(rw, UVM_PREDICT_WRITE);
  rw.status = status;
end
```

Figure 7. uvm_reg::sample() function call

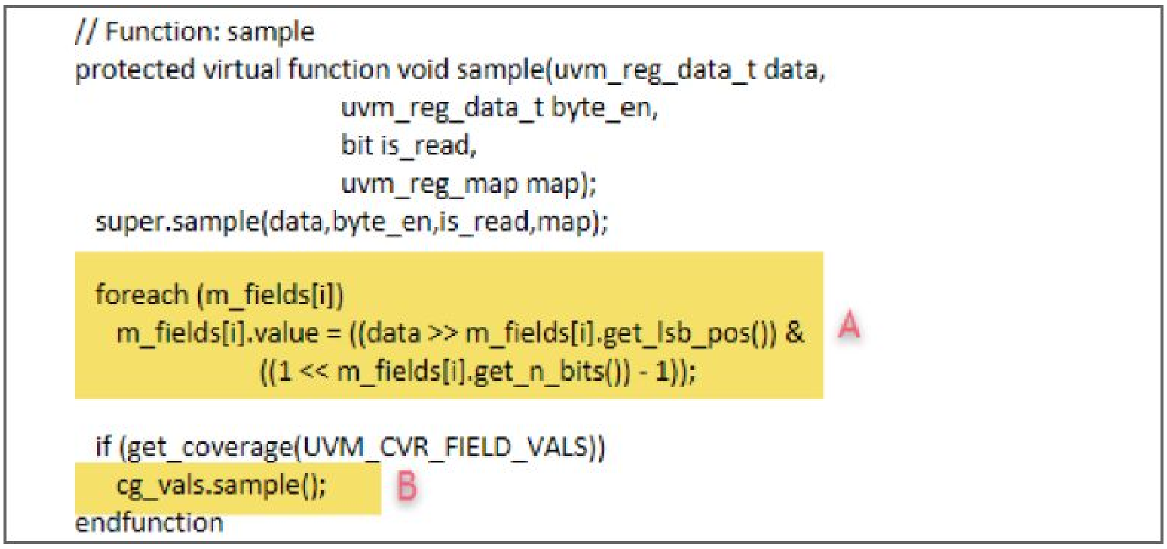
The default uvm_reg::sample() function is empty, as shown in Figure 8.



```
protected virtual function void sample(uvm_reg_data_t  data,
                                       uvm_reg_data_t  byte_en,
                                       bit             is_read,
                                       uvm_reg_map     map);
endfunction
```

Figure 8. uvm_reg::sample() function definition

Thus, to sample the coverage after each register access we need to implement the uvm_reg::sample() function, as depicted in Figure 9.



```
// Function: sample
protected virtual function void sample(uvm_reg_data_t data,
                                       uvm_reg_data_t byte_en,
                                       bit is_read,
                                       uvm_reg_map map);
  super.sample(data,byte_en,is_read,map);

  foreach (m_fields[i])
    m_fields[i].value = ((data >> m_fields[i].get_lsb_pos()) &
                        ((1 << m_fields[i].get_n_bits()) - 1));

  if (get_coverage(UVM_CVR_FIELD_VALS))
    cg_vals.sample();
endfunction
```

Figure 9. uvm_reg::sample() function implementation

As shown in Figure 7, the register-field values are updated **after** sampling the coverage because the uvm_reg::sample() is called before the uvm_reg::do_predict (which updates the register fields in RAL model).

Thus to make sure the register-field values are updated **before** sampling the coverage the register-fields are updated manually(Marker A) and then coverage sampling is done(Marker B), Figure 9.

B. Explicit prediction

This prediction mode updates the register model on all monitored transactions. It uses a predictor component and the UVC adapter.

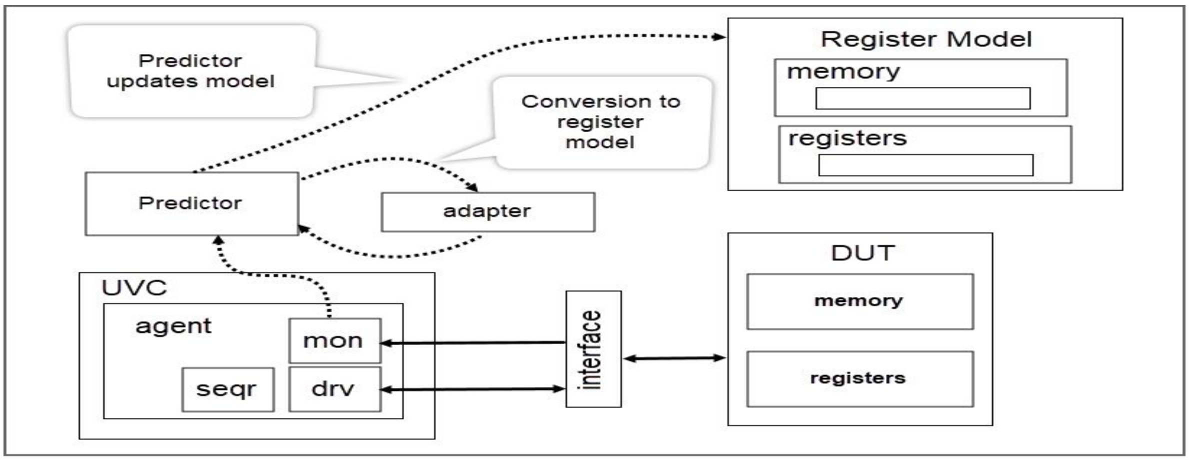
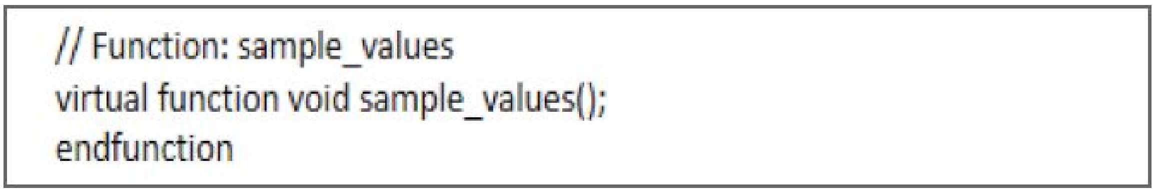


Figure 10. Explicit Prediction Model

With an explicit predictor, when register access is performed, the monitor sends out a transaction to the analysis port which is connected to uvm_reg_predictor and this triggers uvm_reg_predictor::write. This method updates the RAL model. After the update, we can explicitly call the uvm_reg::sample_values() method.

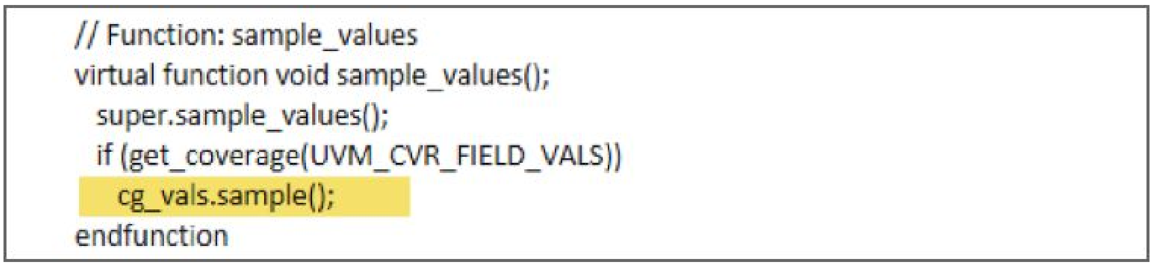
The default uvm_reg::sample_values() function is empty. (See Figure 11)



```
// Function: sample_values
virtual function void sample_values();
endfunction
```

Figure 11. uvm_reg::sample_values() function definition

Thus, in-order to sample the coverage we need to implement the uvm_reg::sample_values() function. (Refer Figure 12)

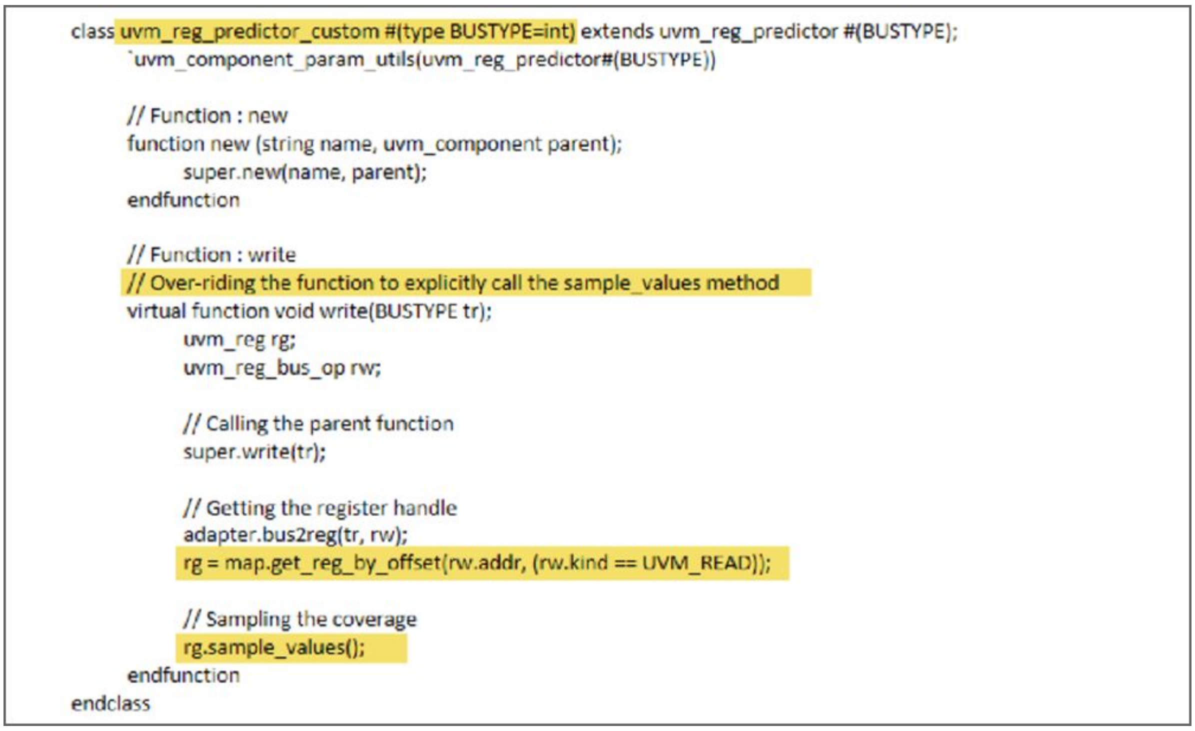


```
// Function: sample_values
virtual function void sample_values();
  super.sample_values();
  if (get_coverage(UVM_CVR_FIELD_VALS))
    cg_vals.sample();
endfunction
```

Figure 12. uvm_reg::sample_values() function implementation

B.1 Example

Let us consider an example of how to call the sample_values() method. The sample_values() method can be called when the user wants to capture the coverage. In this example, the custom predictor class is created in-order to override the *write()* method and to explicitly call the sample_values() method. We can explicitly call the uvm_reg::sample_values() after every register-access, as depicted in Figure 13.



```
class uvm_reg_predictor_custom #(type BUSTYPE=int) extends uvm_reg_predictor #(BUSTYPE);
        `uvm_component_param_utils(uvm_reg_predictor#(BUSTYPE))

        // Function : new
        function new (string name, uvm_component parent);
                super.new(name, parent);
        endfunction

        // Function : write
        // Over-riding the function to explicitly call the sample_values method
        virtual function void write(BUSTYPE tr);
                uvm_reg rg;
                uvm_reg_bus_op rw;

                // Calling the parent function
                super.write(tr);

                // Getting the register handle
                adapter.bus2reg(tr, rw);
                rg = map.get_reg_by_offset(rw.addr, (rw.kind == UVM_READ));

                // Sampling the coverage
                rg.sample_values();
        endfunction
endclass
```

Figure 13: Custom reg_predictor class with overridden write function.

With the use of uvm_reg::sample() and uvm_reg::sample_values() we will be able to sample the RAL functional coverage.

IV. Role Of Register Model Generators

The sample() and sample_values() method implementations, as depicted in Figure 9 and Figure 12, can be done by the register model generators. If the generator is unable to do so, the user can write a wrapper script to include the implementations.

Since the sample() is implicitly called, the user doesn't have to do anything. However, the sample_values() method has to be called explicitly by the user, as depicted in Figure 13. This is imperative because the place at which to call the sample_values() method is based on the user's need, hence this cannot be generalized and included by the register generators.

V. Comparison And Suggestions

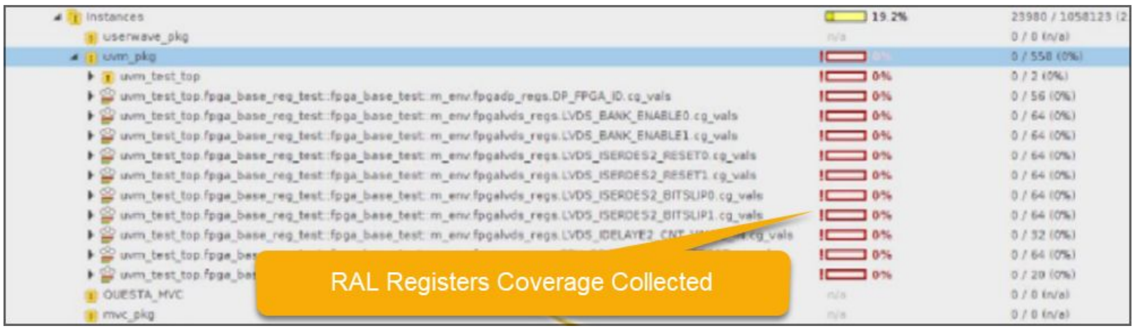
The sample() method is a protected virtual function, hence it cannot be called explicitly. On the other hand, the sample_values() method is just a virtual function and can be called by the user at the desired place, explicitly.

The sample() method is called implicitly on every register access, hence the user doesn't have to worry about calling the sample() method. However, the sample_values() task needs to be explicitly called.

Thus, when the auto-prediction scheme is used, the sample() method has to be used and in the explicit-prediction scheme, it is much more convenient and flexible to use sample_values() method.

VI. Results

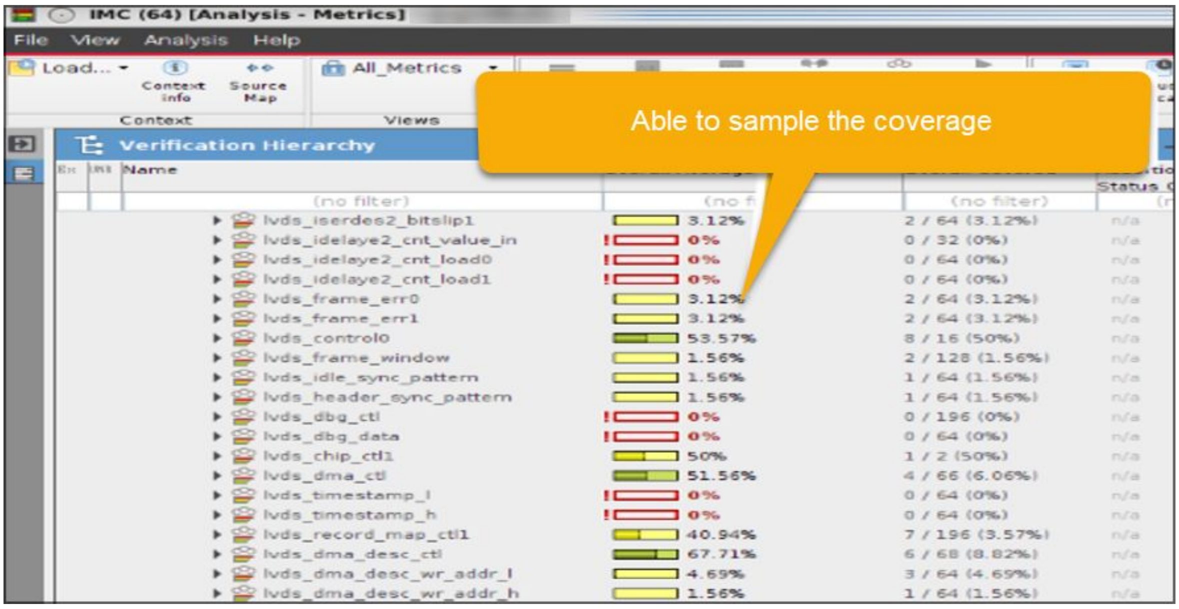
Without the implementation of either uvm_reg::sample() or uvm_reg::sample_values() the RAL functional coverage will only be created but not sampled. (See Figure 14)



```
Instances                                                                  19.2%   23980 / 1058123 (2
  userwave_pkg                                                             n/a     0 / 0 (n/a)
  uvm_pkg                                                                  0%      0 / 558 (0%)
    uvm_test_top                                                           0%      0 / 2 (0%)
    uvm_test_top.fpga_base_reg_test::fpga_base_test::m_env.fpgadp_regs.DP_FPGA_ID.cg_vals                0%   0 / 56 (0%)
    uvm_test_top.fpga_base_reg_test::fpga_base_test::m_env.fpgalvds_regs.LVDS_BANK_ENABLE0.cg_vals       0%   0 / 64 (0%)
    uvm_test_top.fpga_base_reg_test::fpga_base_test::m_env.fpgalvds_regs.LVDS_BANK_ENABLE1.cg_vals       0%   0 / 64 (0%)
    uvm_test_top.fpga_base_reg_test::fpga_base_test::m_env.fpgalvds_regs.LVDS_ISERDES2_RESET0.cg_vals    0%   0 / 64 (0%)
    uvm_test_top.fpga_base_reg_test::fpga_base_test::m_env.fpgalvds_regs.LVDS_ISERDES2_RESET1.cg_vals    0%   0 / 64 (0%)
    uvm_test_top.fpga_base_reg_test::fpga_base_test::m_env.fpgalvds_regs.LVDS_ISERDES2_BITSLIP0.cg_vals  0%   0 / 64 (0%)
    uvm_test_top.fpga_base_reg_test::fpga_base_test::m_env.fpgalvds_regs.LVDS_ISERDES2_BITSLIP1.cg_vals  0%   0 / 64 (0%)
    uvm_test_top.fpga_base_reg_test::fpga_base_test::m_env.fpgalvds_regs.LVDS_IDELAYE2_CNT_[illegible]cg_vals  0%   0 / 32 (0%)
    uvm_test_top.fpga_be[illegible]                                        0%      0 / 64 (0%)
    uvm_test_top.fpga_be[illegible]                                        0%      0 / 20 (0%)
  QUESTA_MVC                                                               n/a     0 / 0 (n/a)
  mvc_pkg                                                                  n/a     0 / 0 (n/a)
```

Figure 14**:** Coverage (0%) without the implementation of uvm_reg::sample() and uvm_reg::sample_values() methods

Thus, we need to implement the uvm_reg::sample() for auto-prediction and uvm_reg::sample_values() for explicit-prediction in order to sample the coverage successfully. (See Figure 15)



```
Name                              (no filter)       (no filter)       Status
lvds_iserdes2_bitslip1            3.12%     2 / 64 (3.12%)      n/a
lvds_idelaye2_cnt_value_in        0%        0 / 32 (0%)         n/a
lvds_idelaye2_cnt_load0           0%        0 / 64 (0%)         n/a
lvds_idelaye2_cnt_load1           0%        0 / 64 (0%)         n/a
lvds_frame_err0                   3.12%     2 / 64 (3.12%)      n/a
lvds_frame_err1                   3.12%     2 / 64 (3.12%)      n/a
lvds_control0                     53.57%    8 / 16 (50%)        n/a
lvds_frame_window                 1.56%     2 / 128 (1.56%)     n/a
lvds_idle_sync_pattern            1.56%     1 / 64 (1.56%)      n/a
lvds_header_sync_pattern          1.56%     1 / 64 (1.56%)      n/a
lvds_dbg_ctl                      0%        0 / 196 (0%)        n/a
lvds_dbg_data                     0%        0 / 64 (0%)         n/a
lvds_chip_ctl1                    50%       1 / 2 (50%)         n/a
lvds_dma_ctl                      51.56%    4 / 66 (6.06%)      n/a
lvds_timestamp_l                  0%        0 / 64 (0%)         n/a
lvds_timestamp_h                  0%        0 / 64 (0%)         n/a
lvds_record_map_ctl1              40.94%    7 / 196 (3.57%)     n/a
lvds_dma_desc_ctl                 67.71%    6 / 68 (8.82%)      n/a
lvds_dma_desc_wr_addr_l           4.69%     3 / 64 (4.69%)      n/a
lvds_dma_desc_wr_addr_h           1.56%     1 / 64 (1.56%)      n/a
```

Figure 15: Coverage after the uvm_reg::sample() implementation for auto-prediction and uvm_reg::sample_values() for explicit prediction

VII. Conclusion

Since the user is oblivious of the 2 obscure methods, uvm_reg::sample() and uvm_reg::sample_values(), they are rarely used. In this paper, we have shown as to how to use them, along with their implementations, when to use them and their effect on coverage sampling.

Acknowledgment

I would like to gratefully acknowledge the critical feedback and support that I received on the content of this paper from my colleagues, Mayukh Majumdar, Vishwanath Anathakrishnan, Manjunatha P.N, Ajay Sharma, Shveta Basavaraj Totad.

References

[1] M. Peryer, D. Aerne, "A New Class Of Registers," - DVCon US 2016
[2] Verification Academy – UVM Cookbook: https://verificationacademy.com/cookbook/registers/integrating
[3] Universal Verification Methodology (UVM) 1.1 Users Guide – Accellera, May 18, 2011
[4] Universal Verification Methodology (UVM) 1.2 Users Guide – Accellera, October 8, 2015
[5] Verification Academy Coverage Cookbook: https://verificationacademy.com/cookbook/coverage