Functional Coverage of Register Access via Serial Bus Interface using UVM

D. M. Tomušilović

***Abstract*-While the UVM Register Abstraction Layer provides support for functional coverage collection, the available elements are not adequate for the coverage of complex register access scenarios. The usage of the Serial Peripheral Interface for register access brings a whole new set of challenges, as bit-resolution access level and low-level communication parameters become essential parts of functional coverage. The External Functional Coverage Subscriber facilitates the handling of these items, while also offering the option of including the additional factors which affect the register access, such as power management, into the coverage metric.**

I. Introduction

One of the critical tasks in the process of verification is the verification of register space. Any bug within the register implementation will manifest itself through a faulty device operation. Complexity of the verification process rises in case that the Serial Peripheral Interface (SPI) is used for register access. The main challenge comes from the fact that it is necessary to break the byte-resolution, thoroughly covered in UVM documentation, in order to support bit-by-bit access. Additional parameters, such as low-level communication timing, need to be taken into consideration. Therefore, the functional coverage of register access via a serial bus interface contains a set of very specific items not applicable to parallel interfaces.

UVM Register Abstraction Layer (UVM_REG) with certain modifications described in [1] proved very suitable for the modeling of the device register space. However, even though the author had initially planned to use the UVM_REG built-in coverage collection elements, it turned out that they have certain limitations that are not straightforward to overcome. Furthermore, the register model would have become aware of some low-level protocol attributes, breaking its modularity.

Much more appropriate is the usage of a separate coverage collection component, which observes the SPI communication as a subscriber, while also referencing the register model and other components of interest, such as the power monitor.

II. Verification Environment

A. SPI Monitor

According to the Universal Verification Methodology, the interface monitor has a passive role of observing the bus interface lines, extracting information from a bus and creating a transaction that can be processed by the rest of the environment.

A typical monitor is a class extended from *uvm_monitor,* which is a part of UVM library. It contains a virtual interface through which the bus information is accessed. Data collection is performed within the predefined *run_phase* task and the collected transaction is published via a TLM analysis port, as represented in Figure 1.

The monitor can also perform checks and basic coverage collection at low-level protocol layer, as well as be used to handle protocol specific requirements, such as frequency measurement.

B. Layer Monitor

In order to overcome UVM_REG limitations regarding partial register access, an intermediate component is placed between the interface monitor and the register model predictor. While the UVM documentation does provide support of partial register access through the field *supports_byte_enable* of *uvm_adapter* class, it recommends maintaining byte-level granularity. To handle bit-level access, the layer monitor subscribes to the SPI monitor analysis port and references the register model. The prediction mechanism is shown in Figure 2. The calculated value is provided to the predictor for further processing. The solution is applicable regardless of the register size and can be extended to support other requirements, such as bit-level or byte-level strobe.

```
class dvcon_spi_monitor extends uvm_monitor;

    `uvm_component_utils(dvcon_spi_monitor)

    virtual dvcon_spi_if                spi_if;
    dvcon_spi_item                      spi_item;
    uvm_analysis_port #(dvcon_spi_item) spi_analysis_port;
    dvcon_spi_cfg                       spi_cfg;

    ...
endclass

task  dvcon_spi_monitor::run_phase(uvm_phase phase);
    ...
    spi_item = dvcon_spi_item::type_id::create("spi_item", this);
    forever begin
      sample_if(spi_item);
      spi_analysis_port.write(spi_item);
    end

    ...
endtask
```

Figure 1. SPI monitor

```
temp_reg  = dvcon_rm.default_map.get_reg_by_offset(address);
temp_data = temp_reg.get_mirrored_value();

for (int i=0; i<DATA_SIZE; i++)
  if (direction == UVM_WRITE)
  begin
    temp_data[DATA_UPDATE_MSB-i] = tr.transmit_data[DATA_LSB+i];
  end
  else
  begin
    temp_data[DATA_UPDATE_MSB-i] = tr.receive_data[DATA_LSB+i];
  end
```

Figure 2. Partial access prediction mechanism in the layer monitor

C. *Predictor*

The primary role of the predictor component is to perform updates of the register model. When the predictor gets an observed transaction via predefined *bus_in* analysis export, it invokes *bus2reg* method of the corresponding adapter, which converts the transaction into a suitable generic format. If check on read is enabled, the read value is compared against the mirrored value of the register being accessed. The predefined *sample* method is called upon register and register block level. Finally, the prediction of the new mirrored value is done, taking the defined register access policy into account.

III. UVM_REG COVERAGE API

D. *Overview*

As the register coverage model is very dependent upon the implemented registers and fields, UVM does not provide any implicit coverage. It does, however, propose usage of functional coverage type identifiers, in order to determine whether certain covergroups are to be instantiated or not [2].

There are several predefined identifiers, presented in Table 1. User-defined and vendor-defined identifiers may be added to handle coverage models which are not natively supported by the UVM documentation.

To include coverage models, a testcase writer may use the static method *include_coverage* of the *uvm_reg* class. Multiple coverage models can be included, as the symbolic values use a one-hot encoding. Additionally, a concrete path specifying the register or the block to which the command applies can be provided as the first argument to the method. Various options are demonstrated in Figure 3.

The register model developer lists all supported coverage models using *build_coverage* as an argument to the register or register block constructor *new()*. Finally, the *has_coverage* method checks whether a coverage model is supported and included, in which case a dedicated covergroup is instantiated.

Table 1. Functional coverage identifiers

Identifier	Coverage model
UVM_NO_COVERAGE	No covergroups built
UVM_CVR_REG_BITS	Read and written data covergroups built
UVM_CVR_ADDR_MAP	Accessed addresses covergroups built
UVM_CVR_FIELD_VALS	Field values covergroups built
UVM_CVR_ALL	All covergroups built

```
function void dvcon_spi_default_test::build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Include one coverage model
    // uvm_reg::include_coverage("*", UVM_CVR_REG_BITS);

    // Include multiple coverage models
    uvm_reg::include_coverage("*", UVM_CVR_REG_BITS   +
                                   UVM_CVR_FIELD_VALS +
                                   UVM_CVR_ADDR_MAP);
    ...
endfunction : build_phase
```

Figure 3. Include coverage models

On the register level, UVM documentation proposes usage of two covergroups dedicated to field values coverage and the coverage of register access in terms of read and written data bits. Their instantiation is shown in Figure 4, while the definition of implemented cover items is given in Figure 5.

```
class dvcon_reg1 extends uvm_reg;
  `uvm_object_utils(dvcon_reg1)

  uvm_reg_field dvcon_field1;
  uvm_reg_field dvcon_field2;

  function new (string name = "dvcon_reg1");
    super.new(name, 8, build_coverage(UVM_CVR_REG_BITS +
                                      UVM_CVR_FIELD_VALS));

    if (has_coverage(UVM_CVR_REG_BITS))
      cg_bits = new();
    if (has_coverage(UVM_CVR_FIELD_VALS))
      cg_vals = new();
  endfunction

  ...
```

Figure 4. Register level covergroups instantiation

```
covergroup cg_bits with function sample(uvm_reg_data_t data,
                                        bit           is_read);
  DATA_0: coverpoint data[0];
  ...
  RW    : coverpoint is_read;
  //cross coverage
endgroup

covergroup cg_vals;
  coverpoint dvcon_field1.get_mirrored_value()
  {
    bins ALL[] = {[0:15]};
  }
  coverpoint dvcon_field2.get_mirrored_value()
  {
    bins ZERO  = {0};
    bins OTHER = {[1:15]};
  }
endgroup
```

Figure 5. Register level covergroups definition

On the register block level, UVM documentation also proposes usage of two covergroups dedicated to field values coverage and the coverage of register access in terms of accessed addresses. The former is quite similar to a covergroup implemented on the register level, but on the block level it facilitates cross coverage between values of fields in different registers. The instantiation is shown in Figure 6, while the definition of implemented cover items is given in Figure 7.

```
class dvcon_block extends uvm_reg_block;
  `uvm_object_utils(dvcon_block)

  dvcon_reg1 my_reg1;
  dvcon_reg2 my_reg2;

  function new(string name = "dvcon_block");
    super.new(name, build_coverage(UVM_CVR_ADDR_MAP +
                                   UVM_CVR_FIELD_VALS));

    if (has_coverage(UVM_CVR_ADDR_MAP))
      cg_addr = new();
    if (has_coverage(UVM_CVR_FIELD_VALS))
      cg_vals = new();
  endfunction

  ...
```

Figure 6. Register block level covergroups instantiation

```
covergroup cg_addr with function sample (uvm_reg_addr_t offset,
                                         bit            is_read);
  OFFSET: coverpoint offset
  {
    bins REG1={`REG1_O};
    bins REG2={`REG2_O};
  }
  RW:     coverpoint is_read;
  OFFSET_x_RW: cross OFFSET, RW;
endgroup

covergroup cg_vals;
  R1F1: coverpoint my_reg1.dvcon_field1.get_mirrored_value()
  {
    bins ZERO={0};
    bins ONE ={1};
  }
  R2F2: coverpoint my_reg2.dvcon_field2.get_mirrored_value()
  {
    bins ZERO={0};
    bins TWO ={2};
  }
  CROSS: cross R1F1, R2F2;
endgroup
```

Figure 7. Register block level covergroups definition

By default, the sampling of all covergroups in the register model should be disabled. To enable the sampling of a certain coverage model, *set_coverage* method is used, as displayed in Figure 8. If invoked for a register block, it will also be recursively called for all subcomponents.

```
void'(dvcon_rm.set_coverage(UVM_CVR_REG_BITS   +
                            UVM_CVR_FIELD_VALS +
                            UVM_CVR_ADDR_MAP));
```

Figure 8. Coverage sample enable

Two predefined virtual methods are provided to perform covergroup sampling at the register and the register block level. Method *sample* is automatically invoked by the predictor prior to the prediction. Method *sample_values* should be explicitly called from the environment. Within the methods, method *get_coverage*, which checks whether a certain model is included and supported and whether its sampling is enabled, is used to condition sampling of implemented covergroups. The implementation at the register and the register block level is shown in Figures 9 and 10, respectively.

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

  if (get_coverage(UVM_CVR_REG_BITS))
    cg_bits.sample(data, is_read);
endfunction

virtual function void sample_values();

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

Figure 9. Register level coverage sampling

```
protected virtual function void sample(uvm_reg_addr_t offset,
                                       bit            is_read,
                                       uvm_reg_map    map);

  if (get_coverage(UVM_CVR_ADDR_MAP))
    cg_addr.sample(offset, is_read);
endfunction

virtual function void sample_values();
  super.sample_values();    // invokes sample_values for all subcomponents

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

Figure 10. Register block level coverage sampling

Figure 11 summarizes all the steps in the process of functional coverage collection using UVM_REG API.

1. SPI transaction is collected by the SPI monitor
2. the collected transaction is published through a TLM analysis port
3. partial access is handled by the layer monitor
4. the method *bus2reg* of the adapter is invoked
5. the transaction provided to the predictor in a suitable generic format
6. the read value checking
7. register level and register block level covergroups sampling
8. prediction

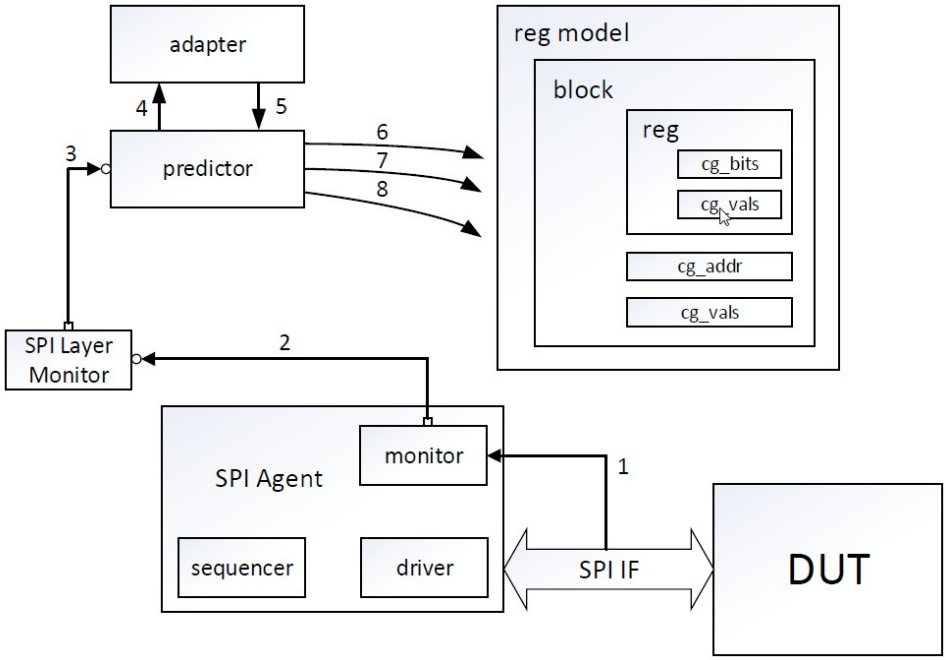

Figure 11. UVM_REG coverage collection diagram

By following the proposed guidelines, the coverage of some simple items, such as transaction direction, data, address or current register state can be successfully performed. However, the experience of working with engineers who use UVM_REG shows that the feature is very error-prone. Some typical mistakes include:

1. using *value* of *uvm_reg_field* class in place of written or read data
2. using *value* of *uvm_reg_field* class in place of mirrored value
3. failing to understand the order of predictor operation – sampling occurs before prediction

4. failing to understand the meaning of API methods – the role of *include_coverage, build_coverage, has_coverage, set_coverage, get_coverage* can be confusing
5. forgetting to enable coverage sampling
6. only partially following the guidelines – for example, sampling is done unconditionally
7. failing to understand the usage model of *sample* and *sample_values* methods – *sample_values* is not called automatically
8. providing references to the rest of the environment in a register, affecting reusability

Another drawback is that the covergroups defined within a register class reduce code readibility. Finally, any scenario involving consecutive accesses to various registers, the remaining transaction fields or the nonregister content creates an undesired dependency between the register model and the rest of the testbench.

Having all these limitations in mind, it turns out that the usage of the external coverage collection subscriber is a much more convenient solution.

IV. External Functional Coverage Subscriber

E. Basic coverage

In addition to the benefits mentioned in [3], a development of an external component, which structure is given in Figure 12, proves very advantageous in the case that a serial bus interface is used for register access. In that case, other than the very basic register access and field value coverage, the coverage model can include a number of crucial register access related items. All implemented covergroups are wrapped within uvm_object in order to support covergroup creation on demand, shown in Figure 13. The displayed covergroup performs field values coverage. The covergroup presented in Figure 14 performs register access coverage, by collecting accessed addresses and transaction data. It can also include the information about the current register content. That is essential for read-only status registers, for which write access should be unobtrusive, regardless of the value in the register.

```
class dvcon_reg_cov_subscriber extends uvm_subscriber #(dvcon_spi_item);
  `uvm_component_utils(dvcon_reg_cov_subscriber)

  dvcon_reg_model          dvcon_rm;
  dvcon_cfg                    cfg;
  dvcon_vals_wrapper   vals_wrapper;
  ...

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    if (cfg.build_vals)
    begin
      vals_wrapper = dvcon_vals_wrapper::type_id::create("vals_wrapper");
      vals_wrapper.dvcon_rm = dvcon_rm;
    end
    ...
  endfunction

  virtual function void write(dvcon_spi_item t);
    // sample
  endfunction
  ...
```

Figure 12. External coverage subscriber structure

```
class dvcon_vals_wrapper extends uvm_object;
  `uvm_object_utils(dvcon_vals_wrapper)

  dvcon_reg_model dvcon_rm;

  covergroup cg_vals;
    option.per_instance = 1;

    R1F1: coverpoint dvcon_rm.my_block.my_reg1.dvcon_field1.get_mirrored_value();
  endgroup

  function new (string name="dvcon_vals_wrapper");
    super.new(name);
    cg_vals = new();
  endfunction

  virtual function void sample();
    cg_vals.sample();
  endfunction
endclass
```

Figure 13. A covergroup wrapped within a covergroup wrapper

```
covergroup cg_access with function sample (uvm_reg_addr_t addr,
                                           uvm_reg_data_t data,
                                           bit            is_read);
  option.per_instance = 1;

  ADDR: coverpoint addr; // bins
  DATA: coverpoint data; // bins
  RW  : coverpoint is_read;
  // mirrored value
  // cross coverage
endgroup
```

Figure 14. Register access coverage

F. Partial access coverage

By using an SPI interface to access the registers, bit-level access can be supported. Therefore, partial and overflow scenarios become a key component of the coverage model. An external subscriber offers a simple way of incorporating transaction length into coverage metrics, as shown in Figure 15.

```
class dvcon_partial_access_wrapper extends uvm_object;
  `uvm_object_utils(dvcon_partial_access_wrapper)

  covergroup cg_partial_access with function sample (uvm_reg_addr_t addr,
                                                     int          length,
                                                     bit         is_read);
    option.per_instance = 1;

    ADDR:   coverpoint addr;   // bins
    LENGTH: coverpoint length; // bins
    RW:     coverpoint is_read;
    CROSS:  cross ADDR, LENGTH, RW;
  endgroup

  function new (string name="dvcon_partial_access_wrapper");
    super.new(name);
    cg_partial_access = new();
  endfunction

  virtual function void sample(uvm_reg_addr_t addr,
                               int          length,
                               bit         is_read);

    cg_partial_access.sample(addr, length, is_read);
  endfunction
endclass
```

Figure 15. Partial access coverage

G. Low-level communication coverage

As the low-level SPI communication timings should not impact the register access as long as the protocol is not violated, dedicated timing-related cover items are implemented. Figure 16 shows coverage of SPI clock frequency at which a register is accessed. The frequency measurement is done by the SPI monitor.

```
covergroup cg_frequency with function sample (uvm_reg_addr_t addr,
                                              real           freq,
                                              bit         is_read);
  option.per_instance = 1;

  ADDR:   coverpoint addr; // bins
  FREQ:   coverpoint freq; // bins
  RW:     coverpoint is_read;
  CROSS:  cross ADDR, FREQ, RW;
endgroup
```

Figure 16. SPI access clock frequency coverage

H. Power management

In order to reduce the power consumption, multiple power domains may be used. The global power monitor observes the input signals, such as voltage level and reset pin and determines whether a certain register is powered on. A locking field callback technique elaborated in [4] is utilized to prevent access to registers within power domains that are turned off. The implemented callback is shown in Figure 17. A critical scenario that the verification side needs to cover is that the registers whose power is shut off act as read-only. A read attempt results in read value isolation. Figure 18 shows the dedicated covergroup.

I. Register interaction

An important aspect of functional coverage is the coverage of the scenarios that include the interaction between several registers. For example, to handle the interrupt logic, a set of registers used to enable, clear or indicate the interrupt status flags is implemented. Of particular interest is the proper implementation of priority handling logic, which includes the triggering of an interrupt while the interrupt clear register is being accessed. The necessary metrics can be obtained by using transition bins, given in Figure 19. The bin in the first coverpoint will be covered if consecutive access to address `REG1_O and `REG2_O is sampled. The second coverpoint contains the bin expression (READ, WRITE => READ, WRITE), which expands to 4 transition bins dedicated to transitions (READ => READ), (READ => WRITE), (WRITE => READ) and (WRITE => WRITE) [5]. Since the bins array construct TRAN_RW[] is used, an individual bin will be associated with each of the 4 created transitions. Finally, cross coverage between the two conditions is implemented.

```
class dvcon_power_callback extends uvm_reg_cbs;
  `uvm_object_utils(dvcon_power_callback)

  function new(string name="dvcon_power_callback");
    super.new(name);
  endfunction

  virtual function void post_predict(input uvm_reg_field  fld,
                                     input uvm_reg_data_t previous,
                                     inout uvm_reg_data_t value,
                                     input uvm_predict_e  kind,
                                     input uvm_path_e     path,
                                     input uvm_reg_map    map);
    dvcon_power_reg temp_reg;
    temp_reg = dvcon_power_reg'(fld.get_parent());

    if (temp_reg.POWER == OFF)
      value = fld.get_reset();
  endfunction
endclass

//  The callback needs to be added to a register field
//  using following piece of code in uvm_reg::build method
    dvcon_power_callback dvcon_power_cb=new();
    dvcon_power_cb.set_name("dvcon_power_cb");
    uvm_reg_field_cb::add(field, dvcon_power_cb);
```

Figure 17. Power supply modeling callback

```
covergroup cg_power with function sample (uvm_reg_addr_t addr,
                                          power_e        power,
                                          bit            is_read);
  option.per_instance = 1;

  ADDR :  coverpoint addr;  // bins
  POWER:  coverpoint power; // bins
  RW   :  coverpoint is_read;
  CROSS:  cross ADDR, POWER, RW;
endgroup
```

Figure 18. Power supply coverage

```
covergroup transition with function sample (uvm_reg_addr_t addr,
                                            rw_e        is_read);
  option.per_instance = 1;

  ADDR: coverpoint addr
  {
    bins TRAN_12=(`REG1_O => `REG2_O);
  }
  RW  : coverpoint is_read
  {
    bins TRAN_RW[]=(READ, WRITE => READ, WRITE);
  }
  ADDR_x_RW: cross ADDR, RW;
endgroup
```

Figure 19. Transition coverage

Figure 20 summarizes all the steps in the process of functional coverage collection using an external component. The wrapped covergroups are instantiated within the component rather than within the register model and step (9) represents their sampling.

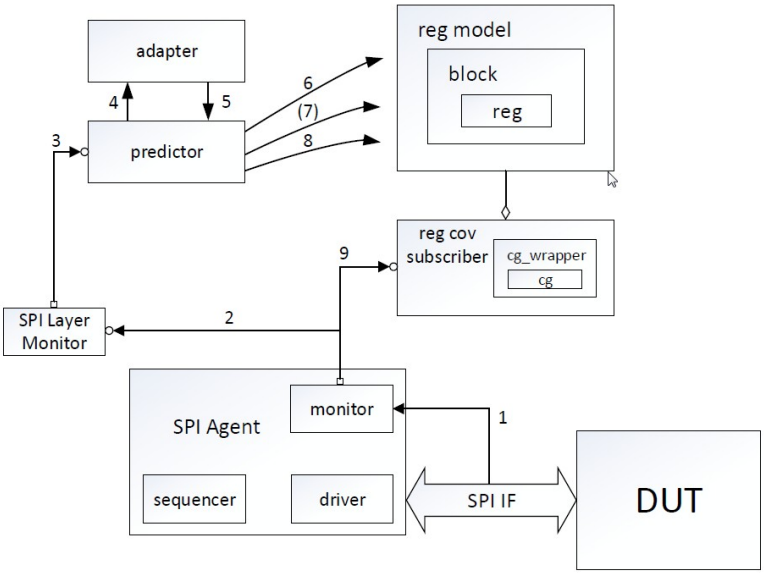

Figure 20. Coverage collection using an external component diagram

V SUMMARY

Using the aforementioned solution, the register modeling and the functional coverage of registers are decoupled. The main benefits of UVM Register Abstraction Layer, such as generation of abstract and reusable stimulus through register sequences and built-in checking mechanism are maintained. However, rather than relying upon the built-in UVM_REG elements, the external component that references the register model is used for coverage collection. This facilitates the coverage of some complex scenarios, such as register interaction and power management impact.

Also, the register model is not polluted by low-level communication details while the coverage of these elements is performed.

ACKNOWLEDGMENT

The author would like to thank the entire Veriest Vtool Serbia d.o.o. organization for their support during the implementation of aforementioned solutions. The code extracts have been generated using the in-house developed Vtool tool [6]. The tool, which is currently under development, supports automatic generation of registers and the register functional coverage, using a PDF file as the input. As the output, the user is provided with the register description in the IP-XACT format, as well as with a UVM/SystemVerilog file. The tool handles a set of register access policies that is a subset of definitions according to the IP-XACT standard, therefore making the solution described in the paper scalable. What is, however, outside of the scope is the coverage of policies which are currently not supported, leaving the implementation of these elements to the user.

REFERENCES

[1] D. Tomušilović, “Extending UVM Register Abstraction Layer for Verification of Register Access via Serial Bus Interface” DVCon Europe 2016.
[2] Accellera, UVM User Guide, v1.1, p87, http://www.accellera.org/
[3] Mentor Graphics, UVM Cookbook, https://verificationacademy.com/cookbook/uvm
[4] M. Litterick and M. Harnisch, “Advanced UVM register modeling – There’s more than one way to skin a reg”, Verilab & DVCon Europe 2014.
[5] IEEE Std 1800-2012, section 19.15.2, p527-528
[6] Vtool, thevtool.com