Calling All Checkers: Collaboratively Utilizing SVA In UVM Based Simulation

Hui K. Zhang
Broadcom
1320 Ridder Park Dr,
San Jose, CA 95131

Abstract- **Scoreboard and SVA are two commonly used self-checking mechanisms in verification. However in common practice, they are employed independently without any elaboration except in the interface. This paper presents a new method and practical solution to utilize SVA collaboratively in UVM based simulations. By combining SVA and scoreboard in conjunction with uvm_config_db, control event and informational parameters can be shared and communicated bidirectional between SVA and scoreboards to quicken and ease the debug. The paper also demonstrates the use cases, the execution flow and some sample codes.**

Keyword- Scoreboard; SVA; UVM; uvm_config_db;

I. INTRODUCTION

The goal of verification is to determine if design implementation meet the specification requirements. Verification employs two major methods, static formal verification and dynamic simulation. Self-checking is rapidly becoming a requirement, as the number of potential errors increases with the complexity of designs. Self-checking in UVM class based simulation is mainly achieved by various checkers residing in monitors and scoreboards, along with SVA. There are two kinds of SVA: immediate and concurrent assertion. Immediate assertion can be used directly inside class based UVM components like uvm_test, scoreboard and monitors. Since concurrent assertion is not allowed in a class, its usage is largely restrained in UVM based simulation except in the interface and modules. The common usage of SVA in interface is typically only checking the signal level protocols that are localized to interfaces. SVA assertion seldom utilizes the abundant information available in the complicated scoreboard. On the other hand, scoreboard [1] is transaction based and it is hard to pinpoint the exact protocol violation location to get close enough to the root cause and the source of the violation. Another drawback of the usage of SVA is that the simulation has a performance penalty after turning on all concurrent assertions, since these concurrent assertions are temporal and checked every clock tick during the simulation. This paper proposes a new method to utilize SVA collaboratively in UVM based simulations and demonstrates how to combine them in symphony to reach their full potential.

II. SCOREBOARD: USAGE & LIMITATION

Scoreboard plays an important role in the UVM based simulation due to its class based nature. The usage of scoreboard is to collect DUT actual inputs and outputs through analysis port. It calculates expected outputs and performs comparisons with the observed outputs. Pass or fail reports are based on the results of the comparisons.

A scoreboard consists of two parts, the predictor which calculates expected outputs, and the comparator which compares the actual and predicted outputs during the run_phase. Figure 1 below shows a typical scoreboard implementation.

Here we use two uvm_analysis_exports which connected to both the input and output monitors' analysis_port to retrieve the observed input and output transactions. The input transaction is passed through to the predicator afterward and transformed into the expected output transaction. Both the expected output transaction and observed output transaction are stored into two uvm_tlm_analysis_fifos.

```
  class   tran    extends uvm_sequence_item;
        rand  bit [31:0]    data_in;
        bit [31:0]        data_out; …
    endclass

    class   my_sb   extends  uvm_scoreboard;
       `uvm_component_utils(my_sb)
        uvm_analysis_export  #(tran)   sb_analexp_in;
        uvm_analysis_export  #(tran)   sb_analexp_out;
        sb_predictor            sb_prd;  //extends from uvm_subscriber
        sb_compartor           sb_cmp; …
    endclass

   class sb_comparator extends uvm_component;
       `uvm_component_utils(sb_comparator)
        uvm_analysis_export  #(tran)   cmp_analexp_in;
        uvm_analysis_export  #(tran)   cmp_analexp_out;
        uvm_tlm_analysis_fifo  #(tran)   cmp_exp_fifo;
        uvm_tlm_analysis_fifo  #(tran)   cmp_act_fifo;
 ….
        function void connect_phase (uvm_phase phase);
            super.connect_phase(phase);
           cmp_analexp_in.connect(cmp_exp_fifo.analysis_export);
           cmp_analexp_out.connect(cmp_act_fifo.analysis_export);
        endfunction

   task run_phase(uvm_phase phase);
     tran exp_tran, act_tran;
    forever  begin
       fork
         cmp_exp_fifo.get(exp_tran);
         cmp_act_fifo.get(act_tran);
     join
    compare(exp_tran,act_tran);
   `uvm_info("sb_cmp",$sformatf("ActTran is %s vs ExpTran %s \n",
          act_tran.print(),exp_tran.print());
  end
 endtask
endclass
```

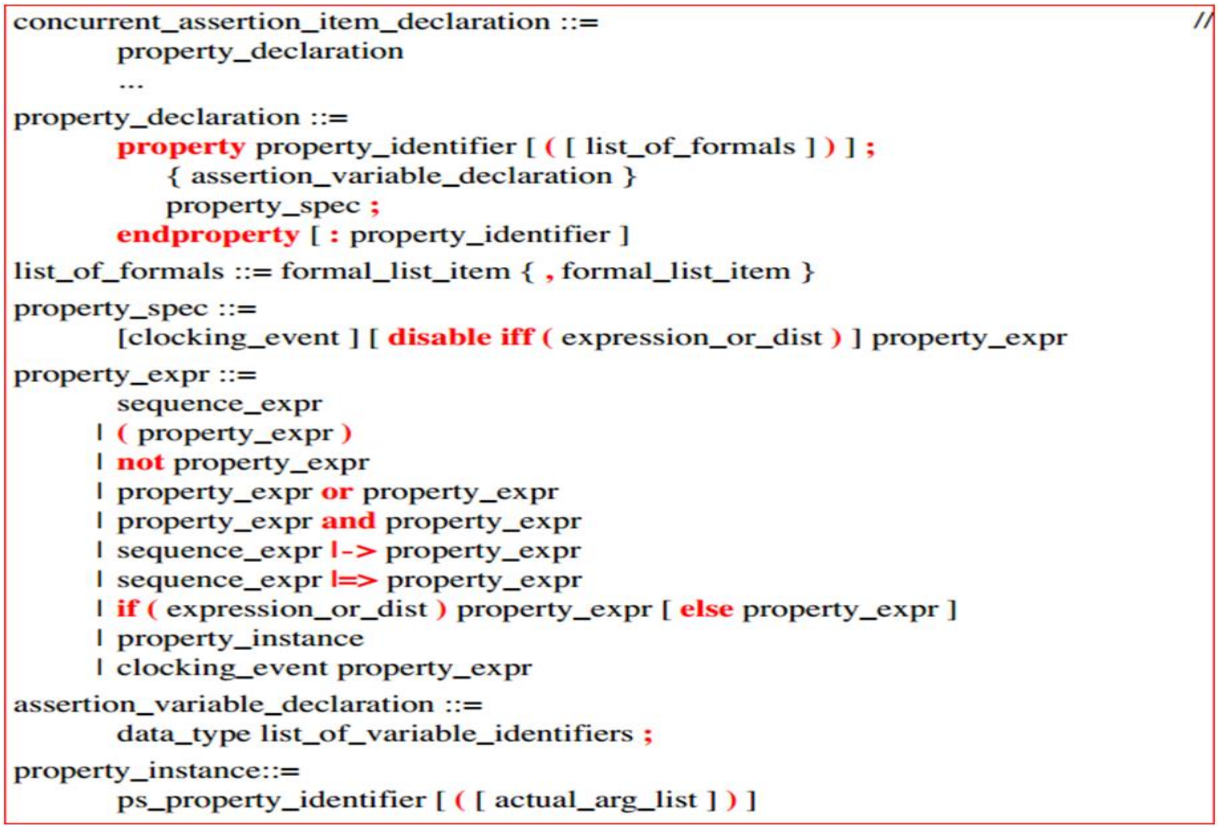
Figure 1: Sample code of scoreboard

Even though the scoreboard can compare and report the pass/fail result in the run_phase, it can only give out a high level outline of the failure which includes information such as the payload length, type, data, and status mismatch at the very end of the transaction. Usually this could be many clock cycles after the origin of the error. Since the scoreboard solely depends on monitors to receive the observed transactions, the scoreboard's content is confined to the IO interface level. All other information like FSM and internal protocols, which are buried deeply down inside the design, is hidden from the scoreboard. Thus the scoreboard is incapable of checking and verifying more detailed protocol violations, down to the clock cycle level. If an error happens in the simulation, this leads the debugging process to be lengthy and painful, as the debugger must then eyeball the waveform and trace back all related signals to their sources.

III. SVA: USAGE & LIMITATION

On the other hand, SVA provides a complementary solution for protocol checking at the cycle-accurate signal level. Specifically, concurrent assertion has the ability to define more accurate temporal design properties which can help to localize the error source and alleviate the debugging process. However, concurrent assertion is illegal within classes and this process cannot access random and dynamic variables either.

Two common uses of concurrent assertions in UVM based simulation are inside of interface and in the modules bound to the DUT. Assertions inside the interface often only focus on the primary IO interface signals. For assertions binding with module or submodule signals, usually the property library has so many assertions that if they are all turned on in the simulation, the simulation performance will be greatly impacted since all the concurrent assertions will be checked in every clock tick in simulation. If a simulation run passes, usually the SVA check will be redundant. However, if a simulation run fails, the supplemental SVA check can be very helpful to facilitate the debug process. This being the case, an automatic mechanism to turn on and off the SVA checkers on demand during simulation will add a lot value in debug. SVA has a few system functions to turn on/off assertions dynamically like $asserton, $assertoff, $assertkill. However, they can only turn on/off all of the assertions if not given specific assertion names.
A concurrent assertion has the following syntax per SystemVerilog LRM. [2]



```
concurrent_assertion_item_declaration ::=                                        //
        property_declaration
        ...
property_declaration ::=
        property property_identifier [ ( [ list_of_formals ] ) ] ;
            { assertion_variable_declaration }
            property_spec ;
        endproperty [ : property_identifier ]
list_of_formals ::= formal_list_item { , formal_list_item }
property_spec ::=
        [clocking_event ] [ disable iff ( expression_or_dist ) ] property_expr
property_expr ::=
        sequence_expr
      | ( property_expr )
      | not property_expr
      | property_expr or property_expr
      | property_expr and property_expr
      | sequence_expr |-> property_expr
      | sequence_expr |=> property_expr
      | if ( expression_or_dist ) property_expr [ else property_expr ]
      | property_instance
      | clocking_event property_expr
assertion_variable_declaration ::=
        data_type list_of_variable_identifiers ;
property_instance::=
        ps_property_identifier [ ( [ actual_arg_list ] ) ]
```

[name:] assert property (property_declaration) pass_statment [else fail_statement]

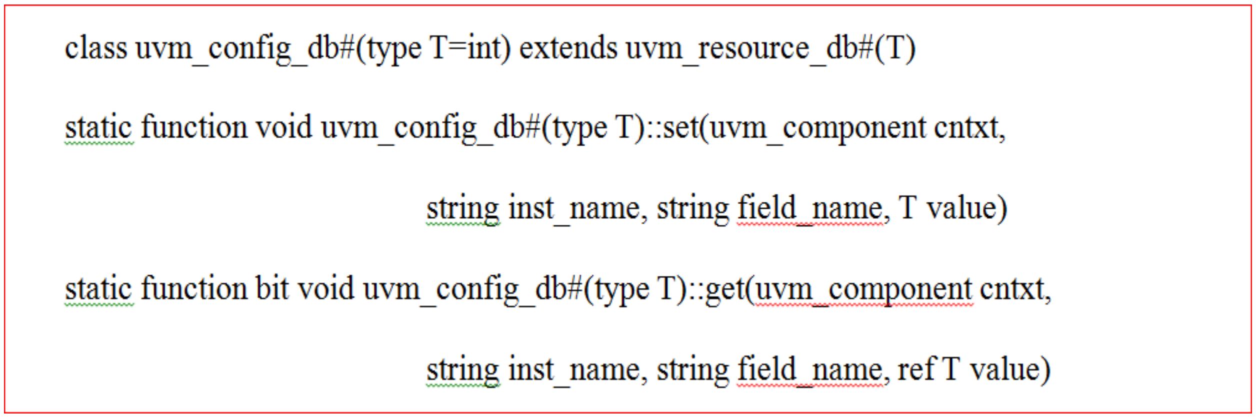
Figure 2: SVA and property syntax

By manipulating expression_or_dist in disable iff clause and property_expr, we can control the trigger of the reset expression or antecedent to minimize the simulation performance penalty. Since concurrent assertion can access the static variables inside a class, some scoreboard information can be saved and passed into SVA for control purposes. Meanwhile, the Pass or Fail information from the SVA can also be fed back into UVM class environment like scoreboard. A detailed example will be provided later in this paper.

IV. CONFIGURATION DB: THE CONNECTING BRIDGE

As we discussed above, both SVA and UVM simulation based checkers like scoreboards, each have their own advantages and limitation as well. In order to combine them to perform verification tasks more efficiently in harmony, we need to bridge them to complement each other to maximize
their benefits while minimizing their limitations.

UVM has a powerful configuration mechanism which together with the UVM Phase concept, can serve the above purpose very well. UVM configuration database is a versatile feature which allows objects and variables been stored and retrieved using lookup strings across various verification components within different hierarchical setup in testbench. There is a pair of set/get functions in uvm_config_db with the below syntax:



```
class uvm_config_db#(type T=int) extends uvm_resource_db#(T)

static function void uvm_config_db#(type T)::set(uvm_component cntxt,

                                    string inst_name, string field_name, T value)

static function bit void uvm_config_db#(type T)::get(uvm_component cntxt,

                                    string inst_name, string field_name, ref T value)
```

Figure 3: uvm_config_db syntax

“cntxt “and “inst_name” are used to specify the hierarchical path of the location of the object handle. Uvm_config_db can provide not only a repository for parameters and objects but also uvm_event for synchronization.

UVM simulation is phase based which includes 9 common phases and 12 run_time phases. When using the set/get functions of uvm_config_db, attention must be paid to the UVM phase. Normally the build_phase works in a top_down manner, which means that higher level components build_phase construct the lower components. But if the class or SVA is defined inside a module, it becomes the most top level. It can be extracted by uvm_root::get() or “null”.

V. PUTTING IT ALL TOGETHER: A PRACTICAL SOLUTION

Collaboratively utilizing SVA in conjunction with Scoreboard and uvm_config_db in UVM based simulation can solve some verification challenges and facilitate simulation debug.

The verification of corner and error case in simulation is a typical task yet sometimes hard to fulfill. By putting these three powerful tools together and employing the concept of feedback and adaptive adjusting, we can bring up a practical solution. The basic idea is similar to search a wrecked ship in the open sea. Likewise, first we rely on scoreboard to give out a broad range for anchoring. Then SVA is utilized to narrow down and zoom into the specific location.

Given below is the diagram for the tb_top module and verification environment to show the relationship among these three utilities.

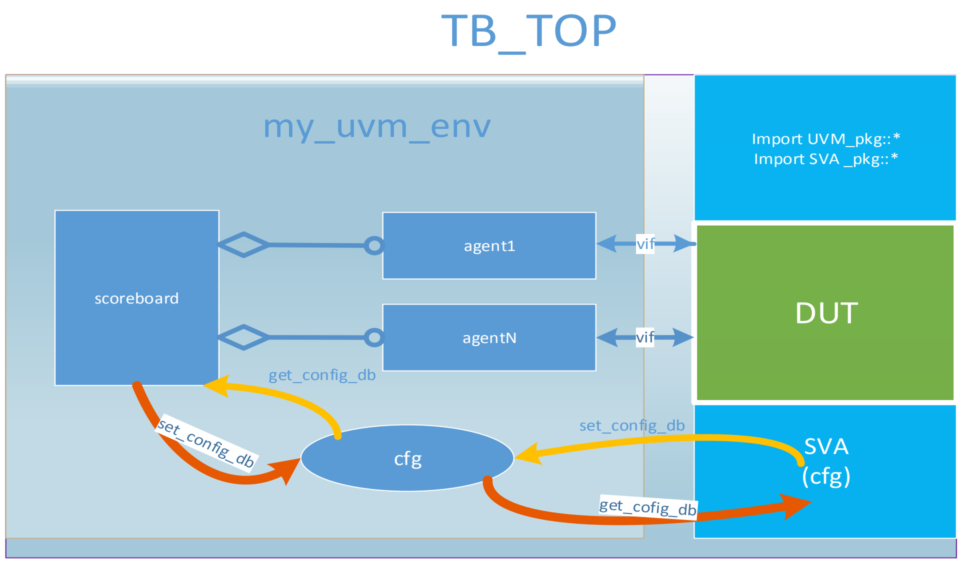


Figure 4: tb_top and env diagram

The basic execution flow and common steps involved are:

1. Run simulation. Record the interest point and parameters reported by scoreboard.
2. Create property and Pass the control parameters to SVA through uvm_config_db.
3. Run simulation with the same seed with SVA.
4. Debug the failure reported by SVA.
5. Adjust control parameters adaptively and repeat the step 3-5 if required. (optional for backward case).
6. Feedback from SVA can be used to control simulation (Optional).

VI. SAMPLE CODE & USAGE

This method can be used to look forward in time for error propagation and recovery, and backward in time to trace the sources of errors and identify error sources. It can also be applied to other scenarios including not limited to corner cases, stress tests, and low power state verifications. It is worth noting that this method can also be extended to any other verification environment components like monitors.

When scoreboard encounters an interest point, an uvm event which has been registered in uvm_config_db can be triggered and informational parameters can be stored in uvm_config_db. All SystemVerilog properties can be encapsulated inside a SVA property package. Tb_top module import the SVA property package along with other uvm packages. Tb_top module consistently monitors and waits for the named event trigger in the procedural code. Once triggered, the information is retrieved from uvm_config_db. It then can be employed in the conditional disable iff() clause to control the turning on of the SVA assertion checking in the simulation. For the forward way, SVA assertion can further check the following protocols and specific sequences until the simulation ends or the named event resets.

Sample code of the forward flow is given below.

```
  class sb_comparator extends uvm_component;
        `uvm_component_utils(sb_comparator)
         …
        uvm_event   sb_sva_event;
        bit  sb_sva_chk_en;
   ….
          function void build_phase (uvm_phase phase);
              …
             sb_sva_event = new ("sb_sva_event");
             uvm_config_db#(uvm_event)::set(  null,
"“*”,“sb_sva_event”,sb_sva_event);
…
         endfunction

      task run_phase(uvm_phase phase);
        tran exp_tran, act_tran;
        forever  begin
          fork
              cmp_exp_fifo.get(exp_tran);
              cmp_act_fifo.get(act_tran);
          join
         compare(exp_tran,act_tran);
       if (comp_error)   begin
        sb_sva_event.trigger();
        uvm_config_db#(bit)::set(  null, “*”,“sb_sva_chk_en”,1);
        …     end
       endtask

   endclass
```

Figure 5: Modified scoreboard

```
package sva_pkg;
     property p_sb_forward(clk,rst_n,sb_sva_chk_en,state);
          @ (posedge   clk) disable iff (!rst_n ||!sb_sva_chk_en)
              state==STATE_ERR|=>##[1:5] (STATE==RECOVERY|| STATE==NORMAL);
     endproperty  : p_sb_forward

     property p_sb_forward_lp(clk,rst_n,sb_sva_chk_en,state);
         @ (posedge   clk) disable iff (!rst_n ||!sb_sva_chk_en)
              state==STATE_PML1|=>##[1:10] (STATE==PML2);
     endproperty  : p_sb_forward
   …
     class sva_cfg extends uvm_object;
                 static int  sb_sva_chk_en;
      endclass

   endpackage
```
Figure 6: SVA property package

```
module top
      import sva_pkg::*;
      uvm_event  tb_sva_event;
       bit  sb_sva_chk_en=0;
       sva_cfg    sva_cfg1;
       dut  dut_1(.*);
     …
      initial begin
      sva_cfg1 = new();
      end_of_elaboration_ph.wait_for_state(UVM_PHASE_DONE,UVM_EQ);
     void'(uvm_config_db #(uvm_event)::get(null,"","sb_sva_event",tb_sva_event);
     tb_sva_event.wait_trigger();
     void'(uvm_config_db #(bit)::get(null,"","sb_sva_chk_en",sb_sva_chk_en);
     sva_cfg::sb_sva_chk_en = sb_sva_chk_en;
     …
     end
      a_sb_forward: assert  property
      (p_sb_forward(sys_clk,rst_n,sva_cfg::sb_sva_chk_en,state))
          `uvm_info("tb_sva",$sformatf("SVA PASSED \n"));
           else  `uvm_error("tb_sva",$sformatf("SVA  FAILED\n")
     …
     endmodule
```

Figure 7: TB_top module

The backward checking of the preceding cause of the named event is also feasible by recording the time stamp of the occurrence of the named event. The other alternative approach is to use uvm_config_db::wait_modified.

static task uvm_config_db::wait_modified(uvm_component cntxt, string inst_name, string field_name)
The task blocks until a new configuration setting is applied that effects the specified field.

The recorded time stamp is fed into the conditional antecedent of another property and asserted as a local time variable to decide how further back the SVA will check the leading conditions. By applying the second round of the simulation with the same seed, the SVA checking can be executed to quickly localize and pinpoint the source to facilitate the debug process greatly. The time threshold can be adaptively adjusted to be fine-tuned to be much closer to the source location by looping through a few iterations of simulation. The result of SVA checking can also be fed backed into the UVM verification components through event based configuration to notify relevant UVM verification components to take some further action or even stop the simulation.

Sample code of the backward flow is given below:

```
class sb_comparator extends uvm_component;
        `uvm_component_utils(sb_comparator..    …
        uvm_event   sb_sva_event;
        int   sb_time0;
        time  sb_time;
   ….
         function void build_phase (uvm_phase phase);
             …
             sb_sva_event = new ("sb_sva_event");
              uvm_config_db#(uvm_event)::set(  null,
"*","sb_sva_event",sb_sva_event);
…
        endfunction
        task run_phase(uvm_phase phase);
           tran exp_tran, act_tran;
           forever  begin
           fork
              cmp_exp_fifo.get(exp_tran);
              cmp_act_fifo.get(act_tran);
           join
          compare(exp_tran,act_tran);
         if (comp_error)   begin
           sb_sva_event.trigger();
          sb_time0=$time/period;
          uvm_config_db#(int)::set(  null, "*","sb_time0",sb_time0);
         end
    …
      endtask
    ….
    endclass
```

Figure 8: Scoreboard for backward flow

```
package sva_pkg;
   property p_sb_backward(clk,rst_n,sb_time0,state);
     @ (posedge   clk) disable iff (!rst_n)
             ($time> (sb_time0-
therhold_backforward)*peroid)&&(state==STATE_ERR)|=>
                 $past(state,2)==STATE_PRE2||$past(state,1)==STATE_PRE1;
    endproperty  : p_sb_backward
    …
    class sva_cfg extends uvm_object;
               static int  sb_chk_en;
               static int sb_time0;
    endclass
endpackage
```

Figure 9: SVA property package for backward

```
module top
      import sva_pkg::*;
      uvm_event  tb_sva_event;
      int tb_time0;
      sva_cfg    sva_cfg1;
      dut  dut_1(.*);
    …
     initial begin
        sva_cfg1 = new();
        end_of_elaboration_ph.wait_for_state(UVM_PHASE_DONE,UVM_EQ);
       void'(uvm_config_db #(uvm_event)::get(null,"","sb_sva_event",tb_sva_event);
        tb_sva_event.wait_trigger();
        void'(uvm_config_db #(int)::get(null,"","sb_time0",tb_time0);
      sva_cfg::sb_time0 = tb_time0;
    …
    end

     a_sb_backward: assert  property
     (p_sb_backward(sys_clk,rst_n,sva_cfg::sb_time0,state))
         `uvm_info("tb_sva",$sformatf("SVA PASSED \n"));
           else   `uvm_error("tb_sva",$sformatf("SVA  FAILED\n")
    …
    endmodule
```

Figure 10: tb_top module for backward flow

VII. CONCLUSION

This paper presented a new method and practical solution to utilize SVA collaboratively in UVM based simulations, as by combining SVA with scoreboard in conjunction with uvm_config_db. Control event and informational parameters can be shared and communicated bidirectional between SVA and scoreboards freely. It can facilitate the debugging process and localize the root cause of the interest point/event as well as analyze the preceding and following protocol/sequences efficiently. It can also minimize the SVA simulation performance penalty. The use cases, execution flow and sample code examples were included in this article to further illustrate the usage of this method.

REFERNCES

[1] Universal Verification Methodology (UVM), www.uvmworld.org
[2] Accellera, SystemVerilog 3.1a Language Reference Manual, http://www.accellera.org