EasyChair Preprint

№ 15264

UVM Portable Stimulus: Synchronized Multi-Stream Parallel-State Scenario in UVM

Ahmed Allam

October 18, 2024

2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

UVM Portable Stimulus: Synchronized Multi-Stream Parallel-State Scenario in UVM

Ahmed M. Allam, ICpedia, Cairo, Egypt (*ahmed.allam@ic-pedia.com*)

***Abstract*—UVM becomes the state of art methodology in the area of digital verification. At some instant, there were different competing verification methodologies and languages such as VMM, AVM and Vera. VMM mainly relies on the power of scenarios to generate TLM stimuli. Then OVM was developed to be open source which permits cooperation across different EDA providers to improve. OVM was supporting both scenarios and sequences but then it was decided to continue with sequences as the TLM randomization strategy. UVM has been evolved from OVM and inherits the concept of sequence in turn. While the sequence is playing the main role in UVM verification, it lacks power of scenarios in running multiple sequences in complex random manner. There was an attempt to include scenarios in UVM but it applies one to one mapping, one sequence runs one scenario. Also, Portable Stimulus Standard (PSS) is a recent C++ based methodology that aims to generate random UVM tests based on defining set of possible scenarios. However, it still requires another language to define scenarios and tool to generate the UVM testcases. In this paper, a novel approach is developed in UVM to use multi-stream parallel state scenario adding more complexity in SoC verification to increase test coverage without additional overhead. Moreover, it can be exploited by PSS to transfer the verification intent easily using the embedded synchronized schemes.**

Keywords—UVM, Sequence, Scenario, Verification, EDA, SoC, Randomization, Coverage, Bugs

I. Introduction

UVM is a standardized methodology used for verifying digital designs and systems-on-chip (SoCs) in the semiconductor industry [1]. It provides a comprehensive framework for creating robust and reusable testbenches to verify the functional correctness of a Design Under Test (DUT).

UVM becomes the state of art verification methodology after number of methodologies released by different EDA corporates and communities such as OpenVera, AVM, VMM and OVM. The power of any verification methodology can be demostrated in its scalability, reusability, maintainability and DUT state space coverage through constrained randomization.

As part of constrained reandomization, UVM adopted the uvm_sequence as a core part of generating stimulus and driving it to the DUT interface. Developing the randomization strategy is the pivot of a reliable verification process. Hence it is aimed at this paper to introduced a novel UVM portable stimulus via dynamic parallel fine synchronizaed scenarios .

The remainder of the paper is organized as follows:

1. Section I introduces current methodologies for scenarios and sequences.
2. Section II introduces the new proposed multi scenario scheme and finally
3. Section III concludes the paper

I. RELATED WORK

A. VMM Scenario

VMM provides scenario and scenario generator classes as the core unit for random exploration of DUT state space. VMM scenario consists of set of items which can be constrained and randomized. According to this scenario random stimulus can be applied to DUT by mapping each state to a task to drive interfaces according to each state. The VMM scenario architecture is demonstrated in Fig. 1.

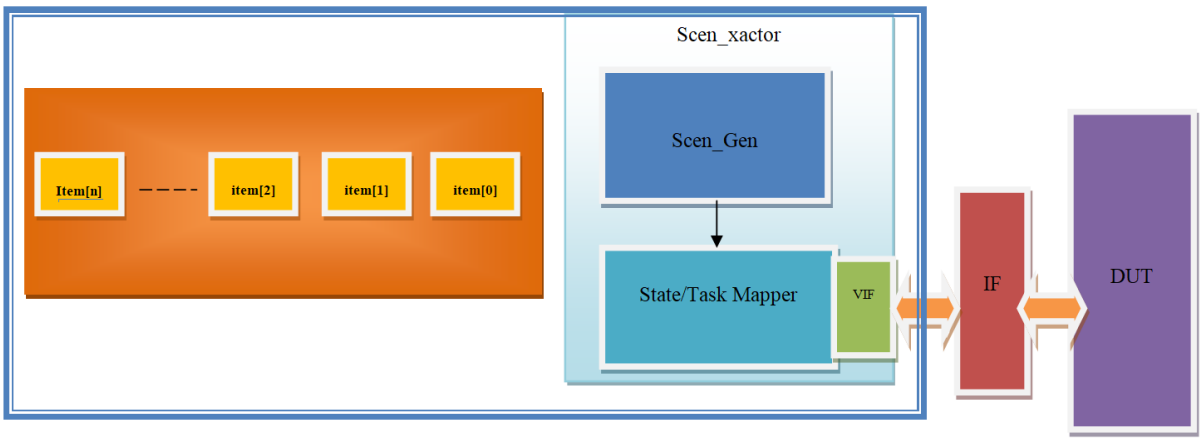

Fig.1 VMM Scenario Architecture

B. UVM Sequence

UVM sequence is a container that instantiations UVM sequence items which includes random transactions. A UVM sequence is executed by specific UVM sequencer. Then the transaction is sent to DUT interface through driver which is connected to the sequencer in the UVM connect phase.

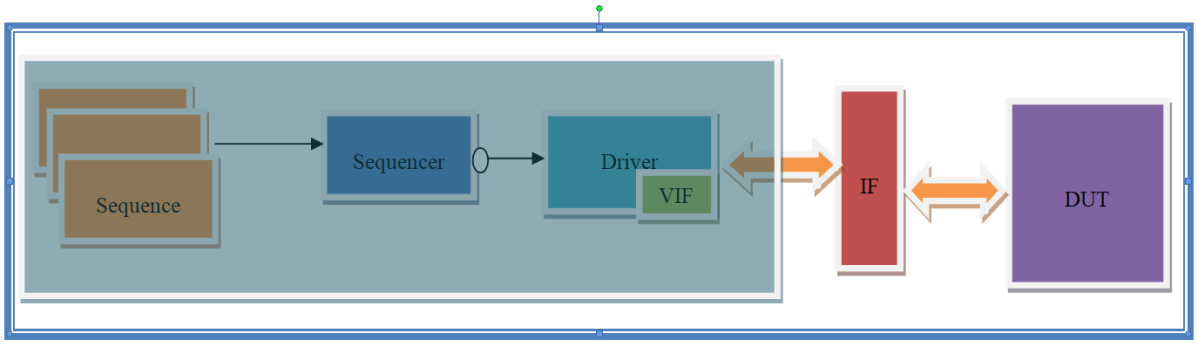

Fig.2 UVM Sequence Architecture

A complex verification environment requires orchestration of different sequences running on different interfaces. This cane be achieved by UVM virtual sequence in conjuction with virtual sequencers [2][3].

C. UVM sequcne as scenario

In [4], it was proposed to develop multi stream scenario library to embed ***vmm_scenario*** strategy into ***uvm_sequence***. Instead of generating raw random stimulus, the sequence is decomposed into multiple scenario states that are to be randomized as shown in the code snippet below

```
class my_alu_scenarios extends uvm_ss_scenario_sequence
#(alu_txn) ;
int simple; // Declare scenario integer
constraint simple_scenario_c{
if(scenario_kind == simple) { // Add scenario constraint
length inside {4};
foreach (items[n]) {
items[n].opcode inside {ADD, SUB, MUL, DIV};
items[n].opcode != items[n+1].opcode;
}
function new (string name = "my_scenario_sequence");
// Define scenario API
simple = define_scenario("First scenario", 10);
endfunction
endclass
```

```
class my_ms_scenario extends
    uvm_ms_scenario_virtual_sequence;
...
// Selecting the scenarios.
   if(!(Seq0.randomize with { scenario_kind == simple ;}))
   if(!(Seq1.randomize with { scenario_kind == simple1;}))
   if(!(Seq2.randomize with { scenario_kind == simple2;}))
// Define Phase A
  PhaseA.add_barrier(Seq0.get_name, 0, 0); // catcher
  PhaseA.add_barrier(Seq1.get_name, 0, 0); // catcher
  PhaseA.add_barrier(Seq2.get_name, 0, 1); // releaser
  Seq0.add_seq_barrier(PhaseA);
  Seq1.add_seq_barrier(PhaseA);
 Seq2.add_seq_barrier(PhaseA); ...
 // Run in parallel
  fork
    Seq0.apply(null);
    Seq1.apply(null);
    Seq2.apply(null);
  Join
endclass
```

However, as shown above the way it defines the sequences and barriers are still created and added in non scalable way. In the following section, the new proposal would be introduced to extend that work and add another portable layer to UVM constrained randomization.

D. Portable Stimulus Standard

One of the challenges in verification process is how to cover all valid paths in DUT state space. In 2018, Accelera published Port Stimulus Standard known by PSS which is a C++ based DSL that relies on mathematical graphs to cover all possible verification scenarios [5]. Then an EDA tools can be used to generate verification environment and test benches such as Cadence PerSpec, Siemens Questa InFact and Breker Trek. A useful study is conducted in [6], to show how PSS can ease and speed up verification cycle. It studies PSS from different perspectives such as, Compile-time parameters, Run-time configuration, Inheritance, Partial description and Semantics equivalence.

II. MULTI-STREAM PARALLEL-STATE SCENARIO

While the solution adopted by [4] is good in terms of importing the power of scenarios into UVM, it only transformed the uvm_sequence to be scenario based sequence. The running of ***uvm_sequence*(s)** is still run in a deterministic way and the randomization is still limited. Also, PSS is written in another language which raises some challenges to spread it among verification teams such as:

- Learning curve and migration
- Requires change in strategic management to migrate to this new methodology
- Tools required to compile PSS and generate UVM test cases
- Over abstraction of scenarios may miss some critical corners
- Changing spec requires regenerating new testcases or modifying old ones. Keep reviewing these changes and keep consistency across different tool versions is quite challenging
- Tools limitation. Till now, there are limited tools supporting PSS.

On the other hand, SystemVerilog [7] [8] and UVM [1] are already embedded with different synchronization mechanisms such as event, semaphore, uvm_event, uvm_barrier. The usage of uvm_event in sequence synchronization is discussed in [9].

In this section, a novel approach is introduced to extend that solution to add scenario of sequences in addition to import scenario architecture into ***uvm_sequence***.

This section is being divided into three parts:

- Solution architecture in demonstrated in subsection A
- Solution implementation is sketched in subsection B
- Finally, use case study is presented in subsection C.

A. Solution Architecture

The verification environment is proposed to be architected as follows:

1. Sequences are declared as usual but embedded with possible DUT states corresponding to sequence functionality
2. Sequences are mapped to tasks act as scenarios states
3. Each test car run scenarios which run sequences in a coordinated way and sequences in turn run internal randomized states

By using this verification strategy, we can keep related functions bounded in sequence while using coordination scheme to synchronize different sequences' states within verification scenario.

As depicted by Fig.3, multiple scenarios are applied to the DUT. There are some dependencies across these parallel scenarios. Single scenario parallel states can wait either for some of the same or different scenario states. The challenge is to develop a flexible architecture that can manage these dependencies and make scenario synchronization easy to achieve.

The solution is decomposed to set of associative arrays that act as lookup for the scenario dependency execution flow. In order to make the scenario state unique, each state is tagged with concatenation of

1. SCEN_ID
2. SCEN_STATE_ID
3. SEQ_STATE_ID

Thus, each state has a unique identifier that distinguishes it in different scenarios. This scenario unique state can also be mapped to DUT by capturing DUT state space and compare it against applied scenario states. This will make sure DUT is responsive to the running verification scenario. DUT current state can be mapped to set of enumeration times placed in an interface. Example for this would be shown in a case study shown later.

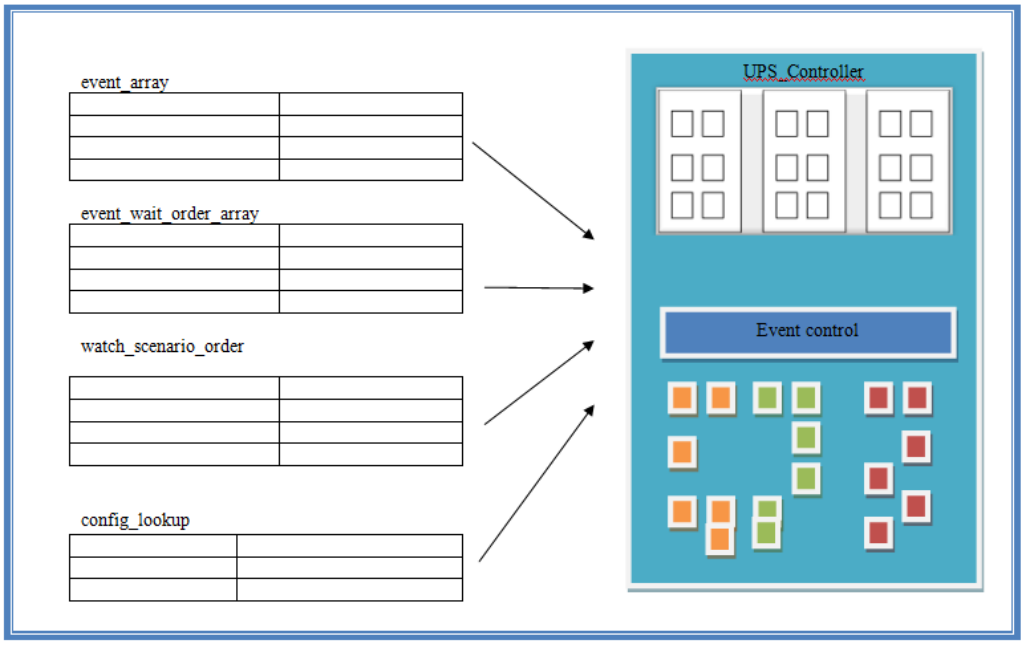

Fig.3 UPS Class Architecture diagram

B. Solution Implementation

The proposed architecture is implemented through set of tasks to ease reusability and scalability of the multi scenario application. The following are set of tasks and their corresponding descriptions. The class is easy to scale to add more tasks to bring more functionality to the multi stream package.

Task	Description
trigger_pre_state_event(string trig_state_id)	This task triggers an prior start of a given state
trigger_post_state_event(string trig_state_id)	This task triggers an event upon completion of a given state
wait_pre_state_event(string wait_state_id)	This task holds an execution of a specific state until an event is triggered that permits execution. Waits for nothing if no event is registered for this state.
wait_post_state_event(string wait_state_id)	This task holds in a specific state until an event is triggered that permits execution. Waits for nothing if no event is registered for this state.
register_lock_state(string wait_state_id, string wait_state_order_id[])	This task registers specific state to wait on hold until some specific states trigger completion
register_lock_scenario(e_ups_scen_kind scen_kind, string wait_state_order_id[], string start_state)	This task registers specific scenario to start when some specific states trigger completion.
register_lock_scenario_w_event(e_ups_scen_kind scen_kind, uvm_event wait_event_order_id, real p)	This task registers specific scenario to start with percentage p when some specific event occurs
add_watch_scenario(string scen_tag, string watch_scen_states[])	This task add watch for some specific scenario to occur with predefines set of states
watch_scenario_to_start(e_ups_scen_kind scen_kind, string watch_scen)	This task watches specific scenario and when hit, it calls another scenario
watch_scenario_to_stop(e_ups_scen_kind scen_kind, string watch_scen)	This task watches specific scenario and when hit, stops execution of another scenario
watch_scenario_to_resume(e_ups_scen_kind scen_kind, string watch_scen)	This task watches specific scenario and when hit, resume execution of another scenario
watch_scenario_override_state(string watch_scen_states[], string state1, state_override)	This task overrides a state in a specific scenario under some event
watch_scenario_override_seq(string watch_scen_states[], string seq, seq_override)	This task replaces a sequence with another extended sequence under some event
super_scenario_stimuli()	This task can be used to call some or all other tasks to create a super scenario

Table.1 UPS class methods

The scenario states are implemented in SystemVerilog tasks with pre and post execute, wait and trigger tasks. Prior state execution, the state holds waiting for a triggering event if exists. Also, post state execution, the state triggers an event indicating state is done as shown in the code below.

```
task ups_foo_state(string state_id);

if(!state_skipped(state_id))begin
  wait_pre_state_event(state_id);
  trigger_pre_state_event(state_id);

  /// start exec

  //// end exec
  trigger_state_done_event(state_id)
  wait_post_state_event(state_id);
  trigger_post_state_event(state_id);
end

endtask
```

By this mechanism, we can add more control on scenarios synchronization and dependency flow. For more control on scenario states, the constraints are done in a dynamic way as shown in code below. This would add more dynamics to running scenarios.

```
constraint c_basic_scen {

  scen_length == 4;
  foreach (ups_scen_new_state[i,j]){
    ups_scen_new_state[i][j] ==
get_state(UPS_BASIC,i,j)
    }
}
```

In this way, a test can just describe the scenario in a higher level of abstraction without being tied to a fixed implementation. Having a mechanism to control dependency across different sequences in a higher level will not only make test easy to understand but also easy to modify and maintain.

Moreover, developing test cases become very interesting using scenario based verification with the following additional benefits:

1. It is easy to understand the test scenario at a glance.
2. Extending a test is very scalable.
    a. One can override scenario order in parent test.
    b. Call super test run scenario and add more states to the scenario.

For sure this is not the end of this work and more and more tasks and orchestration mechanisms can be added to add more scenarios complexity and flexibility in the verification environment. Also, there is an interesting work on dynamic constraint done at [10] [11] [12] and it is interesting to reuse some of these contributions and add them in future work.

C. Case-Study

From traditional computer systems to Data Centers, High Speed Serial Links (SerDes) represents a vital component in nowadays communication technologies. Struggling to meet computational speed, more challenges are now arise in increasing SerDes bandwidth through multi lane support, increasing frequency and PAM4-6 support. SerDes PHYs should act the physical layer for standard communication protocols such as PCIe1-6, USB2/3, SATA1/2/3, CCIX, and DDR2/3/4/5. Increasing bandwidth helps satisfy processing speed demands which is driven by challenges appear at emergent applications such as AI, Big data, Autonomous vehicles, Data centers, etc.

SerDes Test Environment should apply verification strategies across different levels of hierarchies by running sequence based on test purpose. However, running random sequence ordering is still limited as demonstrated before which would limit applying complex stimulus and approaching chip real life scenarios.

Running complex scenarios requires running multiple parallel sequences across different SerDes PHY interfaces. Sequences applied across TX, RX, PLL digital interfaces. Handling dependencies between PLL, TX sequences in a dynamic way would be hard to manage.

For sure any complex dependency can be implemented in within sequence arbitration. However, adding more and more conditions will not only complicate the test environment but also will make it hard to explore.

The UPS scenario creates a super hierarchy to easily manage complex scenarios without neither changing the test environment Skelton nor modifying the environment components deeply. This will make it easy to understand and modify test purpose. Having a higher layer will enhance test purpose observability and controllability simultaneously.

As an example, in some specific test, we may apply the following scenario. Here we have different sequences.

1. RESET Sequence
2. PWR_STATE Sequence
3. CALIBRATION Sequences
4. DATA Sequence

In Fig.4 a code snippet is shown how to orchestrate parallel scenarios in an easy way. The verification engineer can use wrapper tasks to define interdependencies between scenario states in a nutshell. Here you can see three scenarios applied, **PWR_STATE, CALIBRATION**, **and DATA_TRANSMISSION**. Beacon is applied in P2 while packet transmission is applied in P0. Here the orchestration mechanism easily decides when to run beacon and when to run packet transmission according to the parallel running power state scenario. Same applied for PLL, TX and RX calibration.

```
task super_scenario_stimuli();

    // wait for p2 done to start pll_clk_cal and hold in p1 till pll_clk_cal ends

    register_lock_state("UPS_PLL_CLK_CAL",{"UPS_PSTATE_UPS_PLL_P1"},"hold")

    // wait for p2 done to start tx_beacon and rx_beacon and hold in p2 till pll_clk_cal ends

    register_lock_state("UPS_DATA_TRANSMIT___UPS_TX_BEACON",{"UPS_PSTATE___UPS_P2"},"hold")
    register_lock_state("UPS_DATA_TRANSMIT___UPS_RX_BEACON",{"UPS_PSTATE___UPS_P2"},"hold")
    // wait for PSTATE Scen state P0 then continue to execute RX Data transmission
    register_lock_state("UPS_DATA_TRANSMIT___UPS_RX_TRANSMIT",{"UPS_PSTATE___UPS_P0"},"hold")
    // wait for PSTATE Scen state P0 then continue to execute TX Data transmission
    register_lock_state("UPS_DATA_TRANSMIT___UPS_TX_TRANSMIT",{"UPS_PSTATE___UPS_P0"},"hold")
    fork

    // wait for scen1 rx and tx data transmission to be done
    // then start low power scenario
    register_lock_scenario("UPS_LOWPOWER",{"UPS_DATA_TRANSMIT___UPS_RX_TRANSMIT",
    "UPS_DATA_TRANSMIT___UPS_RX_TRANSMIT"},"join_all"/*"join_any","serial"*/);
    join

endtask
```

Fig.4 SerDes scenario

This scenario can be visually inspected using the wave form timing diagram depicted in Fig.5. Capturing DUT state space can be done by mapping some PLL, TX, RX , analog supplies, … etc, to enumeration type as shown in Fig. 6. After execution of certain verification scenario state, one can apply some checker to verify the DUT enters the corresponding physical state.

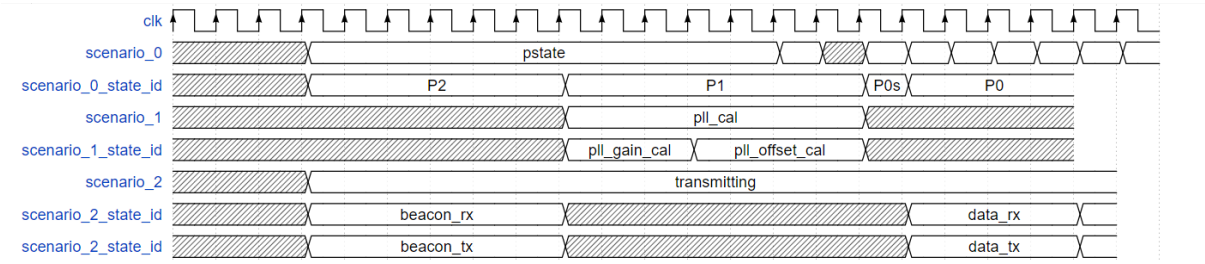

Fig.5 State waveform timing diagram

```
`ifndef DUT_STATE_SV
`define DUT_STATE_SV
 typedef enum {P2,P1,P0s,P0} pwr_state_e;
 typedef enum {VREG_ON,VREG_OFF} vreg_state_e;
 typedef enum {RX_PWRDN,RX_PWRUP,RX_VCO_CAL,RX_VCO_FINE_CAL,RX_DATA_TRANSMIT} rx_state_e;
 typedef enum {RX_PWRDN,RX_PWRUP,RX_VCO_CAL,RX_VCO_FINE_CAL,RX_DATA_TRANSMIT} tx_state_e;
 typedef enum {PLL_PWRDN,PLL_PWRUP,PLL_COARSE_CAL,PLL_FINE_CAL} pll_state_e;

 `define PLL_ANA tb.dut.....ana.pll
 `define TX_ANA tb.dut.....ana.laneX.tx
 `define RX_ANA tb.dut.....ana.laneX.rx
 `define PLL_DIG tb.dut.....dig.pll
 `define TX_DIG tb.dut.....dig.laneX.tx
 `define RX_DIG tb.dut.....dig.laneX.rx

 interface dut_state;

   rx_pwr_state_e rx_pwr_state;
   tx_pwr_state_e tx_pwr_state;
   ////////////////////////////
   rx_state_e rx_state;
   tx_state_e tx_state;
   pll_state_e pll_state;
   ////////////////////////
   vreg_state_e rx_vreg_state;
   vreg_state_e tx_vreg_state;
   vreg_state_e pll_vreg_state;

   /// Ex for DUT state capture
   assign pll_vreg_pwr_state (`PLL_ANA.v1 && `PLL_ANA.v2 ..) ?  VREG_ON : VREG_OFF;
   assign rx_vreg_pwr_state (`RX_ANA.v1 && `RX_ANA.v2 ..) ?  VREG_ON : VREG_OFF;
   assign tx_vreg_pwr_state (`TX_ANA.v1 && `TX_ANA.v2 ..) ?  VREG_ON : VREG_OFF;
   assign pll_state = `PLL_DIG.pll_fsm;

 endinterface

 `endif // DUT_STATE_SV
```

Fig.6 DUT State capture in interface enumeration data types

III. CONCLUSION

This paper proposed a novel synchronized multi stream scenario which is added to the UVM verification power. Each uvm_sequence will act as a state in a parent scenario. Using this scheme, it is easy to run multiple scenarios simultaneously. Moreover, it adds the scenario/state dependency lookup to run interoperable scenarios. This approach makes verification environment simpler while adding the dynamics of the test stimulus in the scenarios and dependency layer specified in super scenario in a more readable and dynamic way. Moreover, this strategy allows developing more complex test stimulus by running scenarios on demand based on complex DUT state traversal.

REFERENCES

[1] "Universal Verifiation Methodology (UVM) 1.2 Class Reference", Accelera, June 2014

[2] C. Cummings, J. Bergeron, "Understanding the UVM m_sequencer, p_sequencer Handles, and the `uvm_declare_p_sequencer [2] Macro" SNUG 2024

[3] C. Cummings, Using UVM Virtual Sequencers & Virtual Sequences, DVCON 2016

[4] K. Kabil, K. Salah, "Efficient Scenario Based Testing Methodology Using UVM," in 17th International Workshop on Microprocessor and SOC Test and Verification, Dec 2016.

[5] PSS 2.1 Language Reference Manual, Accelera 2023

[6] Xia Wu, Jacob Sander, Ole Kristoffersen, Does It Pay Off To Add Portable Stimulus Layer On Top Of UVM IP Block Test Bench? DVCON Europe 2020

[7] "SystemVerilog Language Reference Manual", IEEE P1800-2017

[8] C. Spear, G. Tumbush, "SystemVerilog for Verification, A Guide to Learning the Testbench Language Features", 3d Edition, Springer

[9] R. Edelman, "Fun with UVM Sequences, Coding and Debugging," in proceeding DVCON 2019

[10] J. Ridgeway, "Interchangeable SystemVerilog constraints", in proceeding SNUG 2014.

[11] J. Ridgeway, "Engineered SystemVerilog constraints" , in proceedings DVCON 2015

[12] J. Ridgeway , "Randomizing UVM Config DB Parameters" , in proceedings DVCON 2015