A UVM Multi-Agent Verification IP architecture to enable Next-Gen protocols with enhanced reusability, controllability and observability

Prathik R[1] **Ramesh Madatha[2]** **Girish Kumar Gupta[3]** **Tony Gladvin George[4]**

[1]Samsung Semiconductor India, Bengaluru, India, prathik.r@samsung.com; [2]Samsung Semiconductor India, Bengaluru, India, ramesh.mr@samsung.com; [3]Samsung Semiconductor India, Bengaluru, India, g.girish@samsung.com; [4]Samsung Electronics, Korea, tony.gg@samsung.com

***Abstract* – This paper introduces a multi-agent Verification IP (VIP) architecture tailored for next-generation, high-speed data transfer protocols. Employing UVM, the architecture addresses the complexity inherent in layered protocol subsystems by decentralizing the verification process across multiple UVM agents. This design enhances granularity in verification, yielding improvements in reusability, controllability, and observability. The use of protocol-configurable agents facilitates dynamic stimulus generation and timely observation, ensuring backward compatibility and reducing development cycles. Key to this architecture is the synchronization mechanism across agents, which allows the handling of multi-protocol traffic, which is crucial for the verification of complex multiplexed protocols such as CXL and PCIe6.0. The introduction of debug interfaces provides real-time, protocol-aware state pattern visibility, drastically improving debug efficiency. Results demonstrate a 35% reduction in VIP development time and a 50% improvement in issue resolution turnaround. This architecture not only streamlines the validation of existing protocols but is also agile enough to accommodate the evolution of future protocol standards.**

I. Introduction

As high-speed computation and data transfers rise, layered protocol subsystems become increasingly prominent. Verification IP (VIP) plays a crucial role in verifying these protocols, where the complexity of the test environment grows proportionally with the complexity of the design under verification. Traditional VIP development, centered around controllability and observability, might fall short with next-gen, layered, and multiplexed protocols. This makes us think of an approach to scale the VIP horizontally along with the reusable standards to minimize the development cycle and look for ways to enhance reusability, controllability and observability of a VIP. In this paper, we would like to propose a solution to develop a reusable VIP that accounts to scale for next-gen multiplexed protocols with enhanced controllability and observability.

II. Problem Statement

Next-gen protocols introduce a hybrid architecture with complex structures involving multiple layers, arbiter and multiplexer. The UVM-based VIPs, targeting single protocol [3], face challenges to scale to the next-gen multiplexed protocols and requires high re-usability, backward compatibility and an ability to accommodate the intricate nature of layered information transfers. As shown in Fig.1, we can see that the next-gen multiplexed protocols have multiple components, where each combined vertically defines a single protocol semantics.

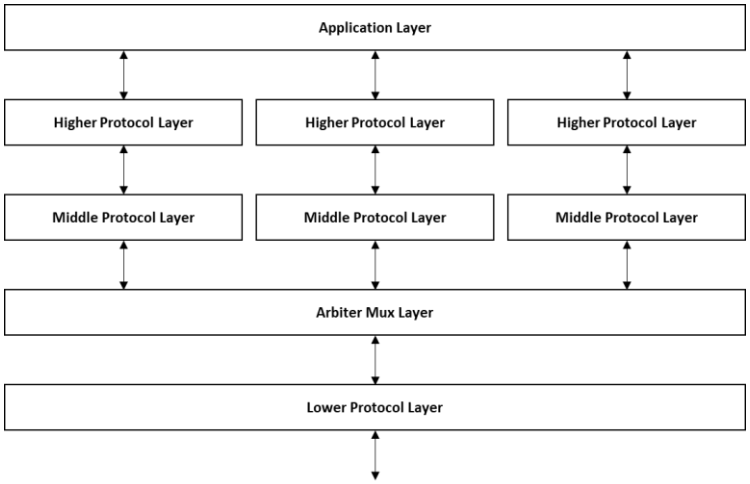

Figure.1 Example of next-gen multiplexed protocols

Configuration of (1) Next-gen multiplexed protocols often involve multiple protocol packets being transmitted simultaneously. For example, CXL supports CXL.io and CXL.mem/cache protocols in parallel, requires synchronization across protocols/applications with arbitration and multiplexing. (2) Few protocols require a change in mode of operation for the demand in its application. For example, PCIe6.0 compared to previous generation, supports FLIT mode (FM) of operation and need backward compatibility to Non-FLIT mode (NFM) operation and does not require arbitration and multiplexing as seen for CXL. The complex architecture and requirements of these protocols can limit the expansion of functionality within a single UVM agent and pose a challenge in scaling bus functional models. The layer based modular implementation alone doesn't help in scaling the functionality as agents becomes bulky and difficult to maintain. Also with limited observability and configurability for such complex protocol models, visualizing and controlling the data transformations at each layer become tedious.

III. Proposed Architecture

In Fig.2, the proposed architecture for next-gen multiplexed protocols is to model the VIP with parallel multi-agent stacks to target different protocols, along with arbiter and multiplexer layer to control the stimulus flow across agents. This synchronized multi-agent approach ensures better dynamic controllability along with observability by having debug interfaces at each layer to visualize the stimulus transformation

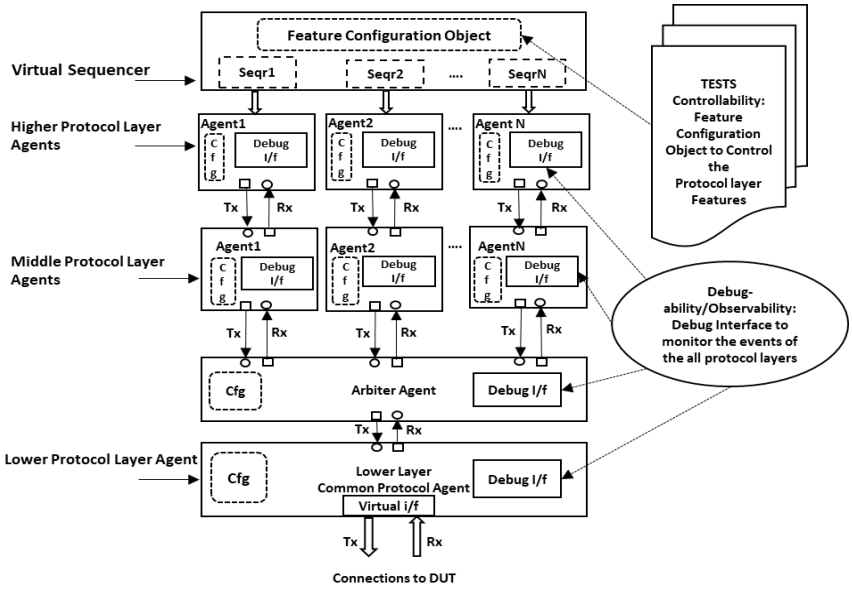

Figure.2 Multi-agent modeling for next-gen multiplexed protocols

Configuration controlled multi-agents: Common configurations with the feedback on negotiated modes/features would provide a better dynamic controllability for virtual sequencing [2] in application layer. These configuration objects shared across agents, help to maintain the backward compatibility and synchronization between agents.

Protocol and state pattern mapped debug interfaces: Each debug interface of an agent containing multiple System Verilog structures [1] would help to visualize the data/control packets, events and state pattern information [3] of every protocol layer. This will aid an engineer to debug the scenario by observing the VIP's state and stimulus flow across multiple agents and layers in waveforms.

Salient features in proposed VIPs architecture targeted for multiplexed Next-Gen protocols:

1. *Configurability* – To address the aforesaid problem of configurations to support multiple protocol packet formats integrated in a common VIP, this architecture enables multi-protocol support over single serial interface. Layered agents are re-used with protocol multiplexing to provide a provision for enabling next-gen or to maintain the legacy usage based upon the intent of verification.
2. *Controllability* – Common configuration object that is passed across layer will control protocol features during run time, ensures better functional reusability and helps in vertical expansion of VIP.
3. *Synchronization* – Across Multi-agents;
    a. similar and interdependent set of configurations are effectively utilized to enable/disable a transaction path, ensures backward compatibility.
    b. arbitrating and multiplexing transactions generated across parallel protocols, helps to maintain synchronization across the multi-agent architecture.
4. *Observability/Debug-ability* – System Verilog structure [1] based event monitoring on debug interface to track the state pattern along with the protocol information for better visibility.

IV. Details of configuration controlled multi-agents

In every revision of layered protocols, each layer will have a set of features which may be newly added or enhanced from the previous version. The extension of features from one generation to another demands the need for backward compatibility. Similarly, in the development of VIP, all features supported has to be included in a configuration class. In order to enhance the VIP for subsequent versions, configuration classes have to be extended to indicate the support of a feature. In Fig.3, it is shown that the layer specific features are grouped into a configuration class and passed down to each UVM agent. The operation of a layer is controlled by the class based configuration, which helps in the modular approach of coding and also helps to ease the backward compatibility.

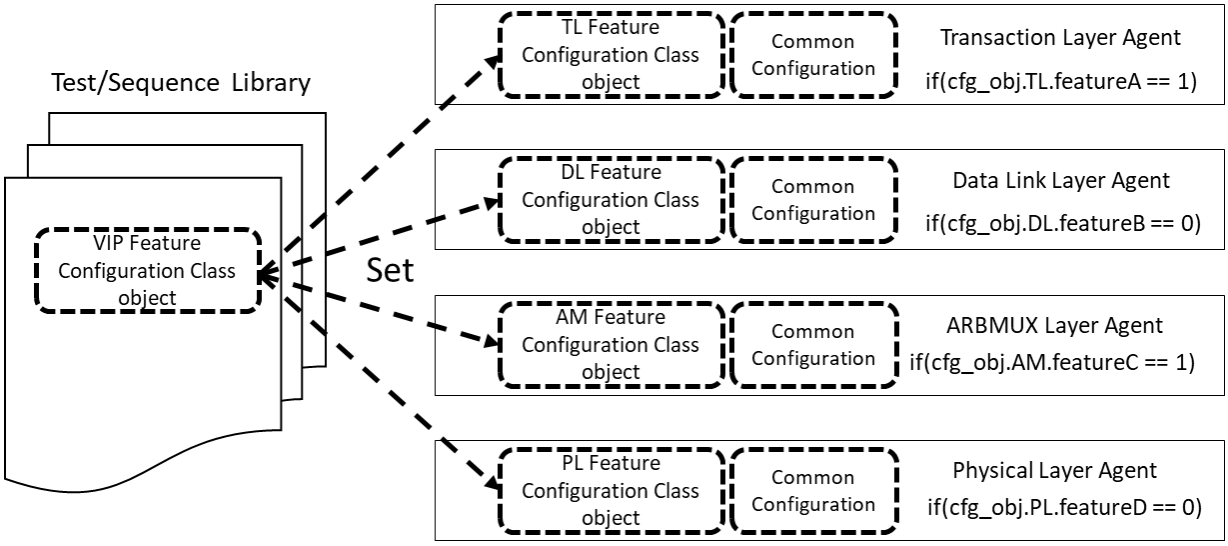

Figure.3 Feature configurations to control the multi-agents

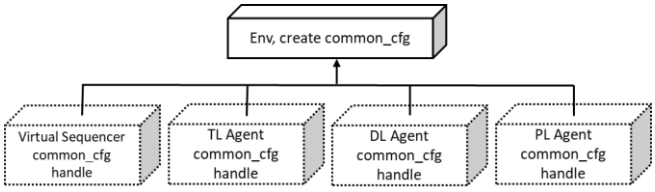

Figure.4 Creation and assigning common configuration

From Fig.4, common configuration class created in the UVM environment, hold variables to control the flow of transaction which are enabled/disabled dynamically. Configuration object handles are passed down to all the agents and also to the virtual sequencer. Layers responsible to enable a feature that effect overall transaction path is based upon the negotiation with link partners. A feedback mechanism established to the top layers and virtual sequencer, provides control on initiating sequences on appropriate sequencer. Similarly, helps other layers to function as per the negotiated mode of operation.

From Fig.5, we can consider the following two cases on mode negotiation between the link partners.

Case1: PCIe Flit & Non-Flit Mode

The negotiated mode in the Physical Layer (PL) decides the FM or NFM operation for PCIe/CXL IO and helps sequence developer to decide the type of transactions to be driven either on to FM agent or NFM agent. Thus the control between agents is established by the use of common configuration object, a simple approach of handle assignment helps to model a feedback mechanism to decide on the dynamic adaptability of the VIP.

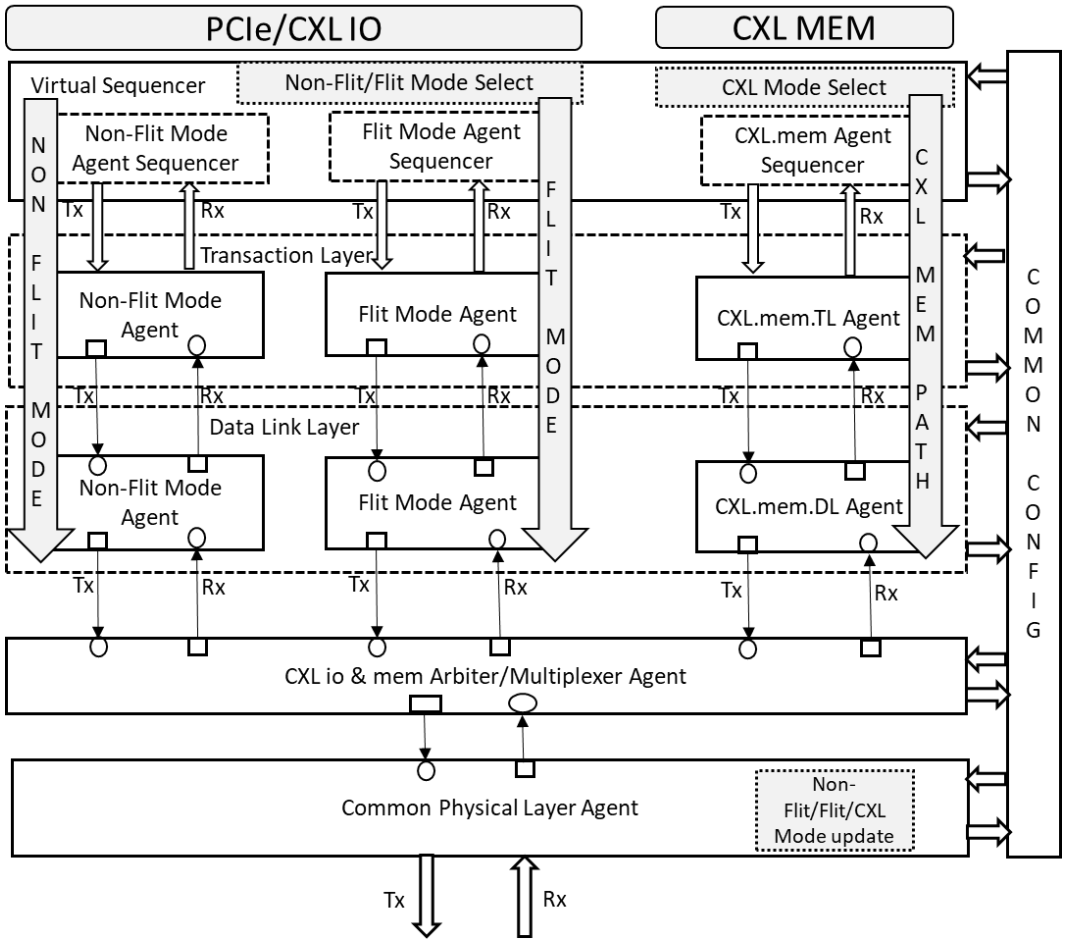

Figure.5 Example to demonstrate the feedback in VIP for PCIe and CXL

Case2: CXL Mode

If the negotiated mode is CXL, we need to activate the arbiter and multiplexer agent and the associated layer specific feature configuration object will decide the arbitration scheme. Here two agents mapped to different protocols would be active in parallel and different traffic to pass to lower layer has to be arbitrated and multiplexed. The arbiter agent ensures synchronization between parallel agents, leading to a vertical expansion of VIP. In non CXL mode, the arbiter and multiplexer agent would pass through the incoming transaction without any modification.

Fig.6, provides an example of the common configuration class that holds the variables to indicate the mode of operation which gets programmed dynamically during the runtime.

```
//Common configuration class
class vip_common_config extends uvm_object;
    //Variables to hold the configurations to be shared across layers
    bit flit_mode_en;
    bit non_flit_mode_en;
    bit cxl_mode;
    `uvm_object_utils_begin(vip_common_config)
        `uvm_field_int(flit_mode_en,UVM_ALL_ON)
        `uvm_field_int(non_flit_mode_en,UVM_ALL_ON)
        `uvm_field_int(cxl_mode,UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

Figure.6 Common configuration class code snippet

In Fig.7, the creation of the object for common configurations is done in the UVM environment and the objects handle is passed to all the required agents. Since the object is created once and an agent is responsible to update the mode of operation by setting the values, and these configurations would be seen by other agents. The handle of common configuration object passed down the hierarchy, helps to maintain the synchronization between the agents and to control the sequencing.

```
//VIP environment to create and pass the handle of config class
class vip_env extends uvm_env;
    `uvm_object_utils(vip_env)
    //Create all layer agents
    //Create the vip_configuration and pass to all layers and virtual sequencer
    i_common_cfg = vip_common_config::type_id::create("i_common_cfg");
    vip_virtual_sqr.i_common_cfg = i_common_cfg;
    uvm_config_db#(vip_common_config)::set(this,"i_nfm_tl_agent.*","vip_common_config",i_common_cfg);
    uvm_config_db#(vip_common_config)::set(this,"i_fm_tl_agent.*","vip_common_config",i_common_cfg);
    uvm_config_db#(vip_common_config)::set(this,"i_cxl_mem_agent.*","vip_common_config",i_common_cfg);
    //So on for all agents (layers)
  .
  .
  .
  .
endclass
```

Figure.7 VIP environment code snippet

In Fig.8, the lower layer (PL) provides the feedback on the mode of operation decided between the link partners by updating the features in the common configuration class object.

```
//PL Agent to negotiate and updates the mode
class vip_pl_ltssm extends vip_pl_state;
    `uvm_object_utils(vip_pl_ltssm)
    //State to negotiate PCIe Flit Mode or CXL
    //Update the vip_common_config
    i_common_cfg.flit_mode_en = 1;
    i_common_cfg.non_flit_mode_en = 0;
    i_common_cfg.cxl_mode = 0;
endclass
```

Figure.8 PL to decide and update the configuration code snippet

Fig.9, demonstrates that VIP developer can make use of the common configuration object in virtual sequence to have a smart way of sequencing, which assures reuse and aids to create a multi-level abstraction in tests and VIP in itself would act as a self-configurable verification environment.

```
//Virtual sequence using the common configuration updates to decide the mode of operation
class vip_virtual_sequence extends uvm_sequence;
    `uvm_object_utils(vip_virtual_sequence)
    //Common object is declared and the handle is assigned in virtual sequencer
    `uvm_declare_p_sequencer(vip_virtual_sqr)
    //Access the common configuration variable from virtual sequencer to decide on the agent (FM or NFM)
    if(p_sequencer.i_common_cfg.flit_mode_en ==1) begin
        //start Flit Mode Sequences
    end
    else if(p_sequencer.i_common_cfg.non_flit_mode_en ==1) begin
        //Non-Flit Mode is enabled, starts Non-Flit Mode Sequences.
    end
    else if(p_sequencer.i_common_cfg.cxl_mode == 1) begin
        //start CXL related sequences
    end
    else begin
        //Default condition: Generally, an erroneous configuration
    end

endclass
```

Figure.9 Virtual sequence code snippet for multi-agent controllability

In Fig.10 and Fig.11, code snippets for arbitration and multiplexing is captured. This would help to explain how multiple modes of operation controls the behavior of an agent and maintains the synchronization between the agents carrying different types of transaction.

```
class vip_arb_mux_agent extends uvm_agent;
    //TLMs to get transactions from DL and to send to PL (Tx)
    uvm_blocking_put_port#(vip_dl_DL_to_PL_packet) AM_to_PL_send_req_put;
    uvm_blocking_get_port#(vip_io_flit_packet) IO_DL_to_AM_req_get;
    uvm_blocking_get_port#(vip_io_flit_packet) MEM_DL_to_AM_req_get;
    vip_common_config common_cfg;
    vip_arb_mux_driver i_am_driver;
    //Get the common config handle
    `uvm_object_utils(vip_arb_mux_agent)
    virtual function void connect_phase(uvm phase);
       if(i_common_cfg.cxl_en) begin
           //Connect AM with CXL.cache/mem agents
       end
       else begin
           //pass the DL transaction to PL
       end
    endfunction
endclass
```

Figure.10 Arbitration and Multiplexer agent code snippet

```
class vip_arb_mux_driver extends uvm_component;
    `uvm_object_utils(vip_arb_mux_driver)
    //Get the common config from the config db
    virtual task run_phase(uvm_phase phase);
       //States to fetch and store IO (PCIe)
       if(i_common_cfg.cxl_mode)
           //States to fetch and store cache/mem
          //States to arbitrate and drive depending upon the mode
    endtask
endclass
```

Figure.11 Arbiter and Multiplexer driver code snippet

If the negotiated mode is not CXL, then the arbiter agent will act as a pass through, similar to a wire to pass the incoming transactions. If the negotiated mode is CXL, then the weighted round robin arbitration logics decides which transaction to pass. The weight for the arbitration is decided by the value configured in agent specific feature configuration object.

V. Details of protocol and state pattern mapped debug interfaces

In the multiplexed protocol, there are several events happening in the VIP related to protocols and to the architected flow of the VIP. Along with the logs, the visual realization of the transaction flow helps for faster debug and also helps to relate to the test scenarios. As shown in Fig.12, a separate interface for debugging is used to map the protocol and VIP state pattern events.

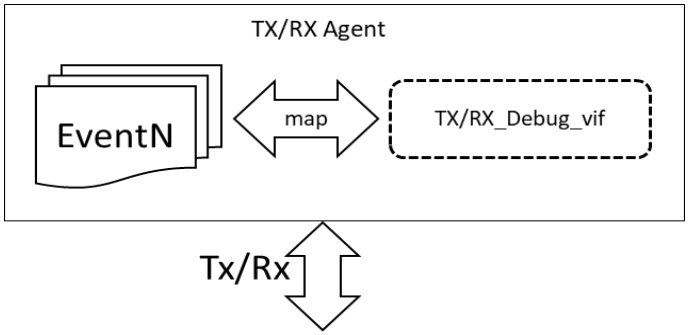

Figure.12 Protocol/state pattern mapping on to interfaces

For example, PL has LTSSM and every state transitions are mapped to the debug interface similarly we see state machines in arbitration and multiplexing called the vLSM. DL has Data Link Control Management State Machine (DLCMSM), every state and relevant data mapping in the state helps the end-user to visualize the protocol transaction flow.

```
//Types of ENUM to represent states in ASCII
typedef ltssm_state_e = {POL, CFG, RECV, L0};
typedef dlcmsm_e = {IDLE, INIT, ACTIVE};
typedef tl_state_e = {form_tlp, drive_tlp, cred_updt};

//Interface definition
interface vip_debug_intf;
    ltssm_state_e i_ltssm_state;
    dlcmsm_e i_dlcmsm_state;
    tl_state_e i_tl_state_pattern;
endinterface

//Example of event generation for state map
class dlcmsm_active_state extends dlcmsm_state;
    //Trigger the event
    →dl_active_state_change;
endclass

//Logic to map the state on to interface
//Interface is passed to the agents
class dl_driver extends uvm_component;
    //run_phase
    virtual task run_phase (uvm_phase phase);
        //forever loop to block on to the event and update
        forever begin
            @(dl_active_state_change)
            vip_dbg_intf.i_dlcmsm_state = ACTIVE;
        end
    endtask
endclass
```

Figure.13 Debug interface code snippet

VI. Results & Conclusion

Feature configuration based agents help in backward compatibility and reduces the VIP development cycle by approximately 35%. Also, validation of older generation products can be completed without any efforts involved to change the legacy tests which ensures timely verification closure. Dynamic feedback approach with common configuration, makes the newly developed sequences very modular and supports complex stimulus generation capabilities. This helped in covering large state space, increasing quality and quantity of functional coverage approximately by 40%. Debug interfaces reduces the issues debug turnaround time approximately by 50% as the root cause analysis would be faster with the enhanced debug capability. The proposed multi-agent architecture promises enhanced configurability, controllability and observability, offering a significant evolution in VIP development for complex multiplexed protocols.

Table.1 Statistics of traditional vs proposed approach

Features	*Traditional*	*Proposed*	*Gain/Loss*
VIP Development time	6 months	4 months	33%
Issue Resolution (TAT)	2 days	1 day	50%
Test Development	1 month	3 weeks	25%
Validation	5 months	4 months	20%
Coverage	2 months	5 weeks	40%

References

[1] System Verilog 3.1a Language Reference Manual, Accellera Organization, Inc.pp. 321-335
[2] UVM Cookbook, Verification academy, Mentor Graphics
[3] Tony Gladvin George, Girish Kumar Gupta, Hojun Shim, ByungChul Yoo, “UVM Layering for Protocol Modelling Using State Pattern”, Samsung Semiconductor.
[4] Ahmed Kamal, “Reusable Extension Layer for UVM to Simplify Functional Modeling”, Mentor.
[5] Rahul Chauhan, Grupreet Kaire, Ravindra Ganti, Subhranil Deb, “Layering Protocol verification: A Pragmatic Approach Using UVM”, SNUG 2014. Verilog 1800-2012”, DVCon 2016.
[6] Tom Fitzpatrick, “Layering in UVM”, Verification Horizons