Verifying Layered Protocols – Leveraging Advanced UVM Capabilities

Parag Goel
Synopsys Inc
RMZ Infinity, Bangalore
0091.80.40188620,
paragg@synopsys.com

Amit Sharma
Synopsys Inc
RMZ Infinity, Bangalore
0091.80.40189192,
amits@synopsys.com

Abstract - Sophisticated generations of protocols are being standardized to address the complex communication within the SoC's as well with external peripherals. These enable the SoC's to deliver improved performance, power, quality of service and numerous advanced capabilities. A layered protocol provides improved data flow and hence is being adopted by the new emerging standards. A layered architecture compliant to the existing OSI model brings in the much desired standardization across different protocols. Most of the new emerging MIPI protocols have adapted to the OSI model for enabling advanced capabilities in mobile technology. PCIe, USB and other advanced protocols have a layered stack and revisions are being regularly rolled out for these standards as well. As the layered protocols bring in more advanced functionalities, we also see new verification challenges associated with them. Taking the MIPI Low Latency Interface (LLI) protocol as an example, we present the various challenges in the verification of a complex layered protocol. Then, we shall look at the how we can tackle these challenges leveraging the different capabilities the UVM base classes provide.

Categories and Subject Descriptors
Universal Verification Methodology, UVM, MIPI, Low Latency Interface, Transaction Level Modeling

General Terms-Verification, Methodology, Verification IP

Keywords-- UVM, Layered Protocol, System Verilog, MIPI, LLI

I. Introduction

The new emerging protocols enable SOC's to deliver much higher performance and bring in advanced capabilities. Most of these protocols comply with the existing layered Open Systems Interconnection **(**OSI) model (ISO/IEC 7498-1) and as such the adoption of such layered protocols has become ubiquitous. As the layered protocols bring in more advanced functionalities, we also see new verification challenges associated with them. The challenges associated with verifying layered protocols vary from the ability to create the appropriate transactions at each layer, verifying the transformations across the different layers, providing the desired visibility and granularity of control, to creating the appropriate scoreboarding techniques. Methodologies such as the Universal Verification Methodology (UVM) have been undergoing continuous evolution to keep up with the many complex requirements in functional verification of complex devices and would need to continue to do so to meet various complex requirements. This paper illustrates the various challenges which a stacked model brings in and presents different ways how these challenges are addressed using verification techniques built over and above what is provided by in UVM components and base classes. These techniques are illustrated with the help of the MIPI (**M**obile **I**ndustry **P**rocessor **I**nterface) LLI (**L**ow **L**atency **I**nterface) stack and can be easily extrapolated to other such layered designs and protocols.

II. Overview of the Layered Architecture (MIPI LLI)

Some of the reasons leading to the wide adoption of a layered architecture are as follows:

- Helps simplify networking designs by breaking them into functional layers. Each layer can then follow specific protocol to perform its task, like data delivery and connection management.
- Protocol layering enables each layer to be governed by simple protocols, each with a few well-defined tasks. These protocols can then be assembled into a multi-layered one with the desired capabilities.
- Helps to debug problems more easily. The user can easily identify the specific layer where network failures originate from and then address a smaller problem.
- The same user-level (application) program can be used over diverse communication networks. For example, the same WWW browser can be used when you are connected to the internet via a LAN or a dial-up line.

The MIPI LLI protocol has been defined by the MIPI-LLI-WG (The working group of the MIPI alliance: www.mipi.org) to provide for a point-to-point, bidirectional communication between the two dies (the application processor and modem/baseband processor).

MIPI LLI is loosely based on the ISO OSI Reference Model (OSI/RM). The architecture of this protocol comprises of the Transport, Data Link, Physical Adaptor and Physical Layer to map the protocol specification

appropriately. The information in the layer *n* header is used for the layer *n* protocol, creating independence among layers.

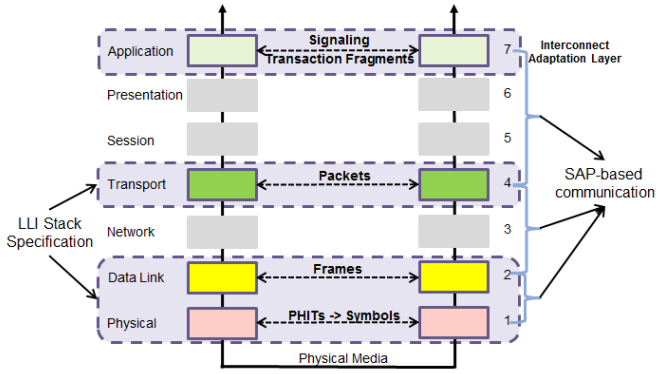

Figure 1 : OSI Reference Model adapted for MIPI LLI

Figure 1 depicts the layered model of MIPI LLI that is essentially derived from the OSI reference model. The two stacks communicate over the serial link of the physical layer (M-PHY). Once it reaches the stack, we have following transformations across the three layers, namely,

- The Transport Layer : Data packets are created in a format that the LLI stack comprehends
- The Data Link Layer: Channel information and credit information are added to form frames and to implement a credit based flow control protocol.
- Physical Adaptor Layer: Transports the frames from Data Link Layer onto the M-PHY

III. Verifying the Layered Structure

The previous section presented a high level overview of the stacked protocol and the transformations that a LLI fragment undergoes across different layers of the stack. The verification of such a model throws its own unique verification challenges. We break down these challenges into the following buckets

- The requirement of providing the desired control and granularity at each layer, i.e. the ability to exercise control on individual layer, to generate traffic corresponding to the intermediate layer, the mechanism to transform higher layer transactions to lower layer ones.

- The ability to have the verification environment map to the layered stack architecturally. This would bring in the additional requirement of having the ability to provide a similar communication mechanism across the layers modeled on the verification side. This has to map functionally to the specific protocol being verified and the infrastructure should be generic enough to map to a variant of the said protocol.

- The ability to independently control each lane in case of multi-lane physical scenarios

- The requirement to have appropriate scoreboarding techniques and the ability to verify transformations at each layer

- The need to create relevant mechanism to terminate the simulation gracefully while collecting all the relevant simulation metrics across all the layers of the stacked model.

- The need to provide debug hooks and visibility provided at each layer.

The following sections go through how different verification components can be created using UVM base classes to meet some of the requirements mentioned above.

A. Sequence Generation

Protocol layering happens when an upstream protocol (e.g. interconnect protocols like AXI/AHB/OCP) instead of being directly executed on a physical interface, is encapsulated in a downstream protocol (e.g. serial MPHY lines). LLI fragments being encapsulated inside LLI packets is an example of protocol layering. An upstream transaction may result in many downstream transactions. Segmenting a large interconnect transaction into multiple LLI fragments is an example of one-to-many layering. Reassembling multiple PHY symbols to form a frame is an example of many-to-one layering.

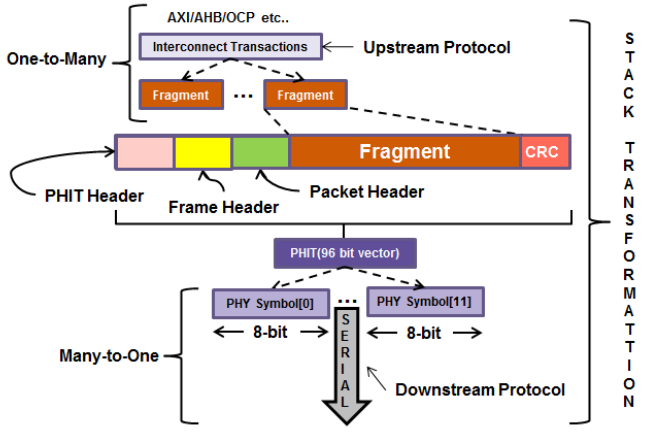

Figure 2: Transaction transformation across LLI stack – the OSI Way

Figure 2 shows the transformations across the LLI stack in more detail. From the layered architecture (as depicted from MIPI LLI) perspective and to bring in the necessary extensibility in terms of stimulus generation,

these are some additional considerations that need to be taken care of in order to create an effective verification infrastructure.

- Ability to inject stimulus of any required type at any layer through the specific sequencers at each layer. To effectively create stimulus at different layers, it should be possible to configure any of the layers as topmost layer generating the highest upstream sequence. By default, this layer would be the Application Layer (**I**nterconnect **A**daptation **L**ayer). This gives the verification engineer the ability to create and verify streams of transactions at the abstraction provided by each of these different layers

- Ability to arbitrate, i.e. mix and match stimulus from the upper layer as well as from the testbench directly at a specific layer. To enable a layer modeled on the verification side to drive and transform stimulus, there should be a collection of UVM sequences and a sequencer associated with that layer. Any layer should be able to drive stimulus from the sequence and sequencer associated with that layer with or without having a higher layer sequence being layered onto it. In a configuration when protocol layering from an upstream sequencer does not happen, the layer would be driving stimulus configured through the sequences and sequencers explicitly associated with it.

- Ability to retrieve and modify the transaction at any specific layer via callbacks, factory mechanism and UVM command-line override.

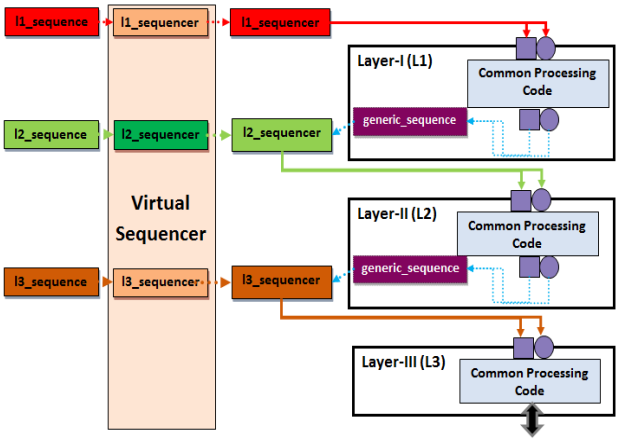

Figure 3: Generic UVM Architecture – stacked protocols

The figure 3 shows the relevant architecture to address the various requirements discussed. Each layer is associated with a sequencer through its TLM ports. A virtual sequencer is provided to synchronize different sequencers across the three layers. Layer-I (L1) is the topmost layer, which is feeding the lower layer via the *l1_sequencer*. After processing and converting the incoming transaction to one that is comprehensible by Layer-II (L2), it is passed down using the *generic_sequence* (which is a parameterized *uvm_sequence*) started on *l2_sequencer*. Now 'L2' can also be driven from the testbench by executing the *l2_sequence* again on the *l2_sequencer*. Traffic from either path gets arbitrated and finally driven to the 'L2' and 'L3' layers which finally process and drives the stimulus down to the physical interface.

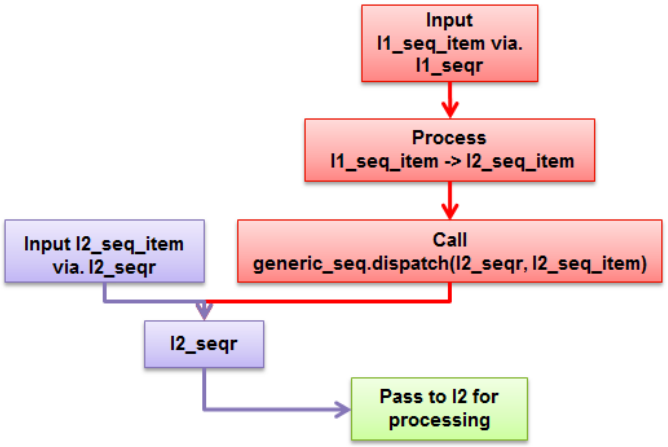

Figure 4 : Flow diagram for sequence item flow in Transmit direction

The code shown in Figure 4 & Figure 5 shows how the *generic_sequence* in each of the layers enables the layering of higher level sequences as and when needed.

```
class generic_sequence#(type REQ=uvm_sequence_item, type RSP=REQ)
                        extends uvm_sequence#(REQ,RSP);

`uvm_object_param_utils(generic_sequence#(REQ,RSP))

➔ Parent Sequencer Declaration ←
`uvm_declare_p_sequencer(uvm_sequencer#(REQ))

 local REQ req;

➔ Dispatch method - Drive REQ on downstream sequencer ←
task dispatch(uvm_seqeuncer#(REQ)  seqr, REQ req);
   this.req = req;
   this.start(seqr);
 endtask

➔ Body method - Initiate REQ ←
task body();
  if (this.req != null) begin
    this.wait_for_grant();
    this.send_request(req);
    this.wait_for_item_done();
  end
endtask

➔ response_handler method - To discard RSP ←
function void response_handler(uvm_sequence_item
response);
/* Just drop the response. */
endfunction
endclass
```

Figure 5: Code for *generic_sequence*

The communication infrastructure will wait for the processed transaction to be available to be driven to the lower layer. It will then populate the reference of the processed transaction in the request sequence queue and then start the sequence on the sequencer. Once the response is returned, the transaction will be flushed from the processed output queue which will then create space for the channel to accept more transactions. Through the UVM Resource Database and the configuration mechanism, the sequence layering can be controlled so that a specific layer can create stimulus based on the sequences associated with it or through sequence layering of upstream sequences. This allows the layering of an arbitrary number of sequences of arbitrary upstream transaction types to be executed on the same downstream sequencer. Furthermore, each layering sequence can be executed with different arbitration priorities, allowing the test or environment to shape the protocol traffic as necessary.

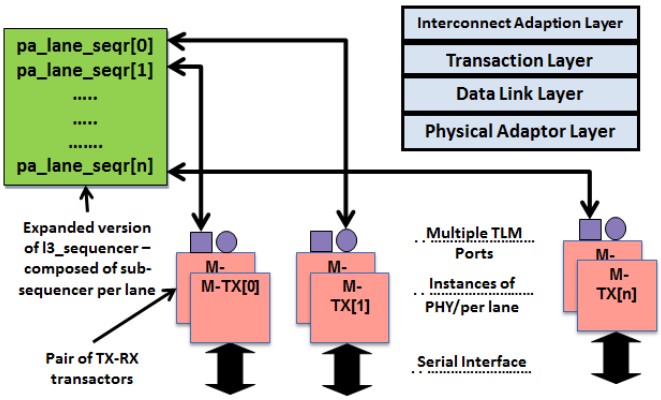

Figure 6: Handling multi-lane scenarios

At the physical layer, most of the protocols have expanded from the traditional use of a single pair of differential signals to multiple differential pairs to achieve increased speed, throughput and to meet the latency requirements. This requires that the user should have a fine grained control of the communication on the serial lanes so that desired speed, throughput etc. can be configured based on the end application. In LLI, the specification defines a support for a maximum of 12 lanes/channels (in multiples of 1, 2, 3, 4, 6 and 12). Hence, from the verification environment, it is required to have access to the configured number of lanes/channels. An additional complexity in the case of LLI is that the number of transmit and receive paths may be asymmetric. Thus, the stimulus generation mechanism should be able to modify the frames on a per-lane basis. This is enabled through an expanded version of the '*l3_sequencer'* which is composed of array of sequencers.

```
➔ Sequencer Definition←
typedef class uvm_sequencer#(mphy_transfer) mphy_sequencer;

class lli_agent extends uvm_agent;

  ➔ Sequencer Handle Declaration - Dynamic Array ←
 mphy_sequencer pa_lane_seqr[];

  function void build_phase(uvm_phase phase);
    if(!uvm_config_db#(lli_configuration)::get(this, "", "cfg", cfg))
                  `uvm_fatal(.....)
     ➔ Sequencer creation ←
    pa_lane_seqr = new[cfg.tx_lane_count];
    foreach(pa_lane_seqr[i])
      pa_lane_seqr[i] = mphy_sequencer::type_id::
                    create($sformatf("pa_lane_seqr[%0d]", i), this);
  endfunction
endclass
```

Figure 7: Multi-lane sequencer composition

This is shown in Figure 6 and the relevant code snippet for the same is shown in Figure 7. Each of these sub-sequencers would correspond to one lane.

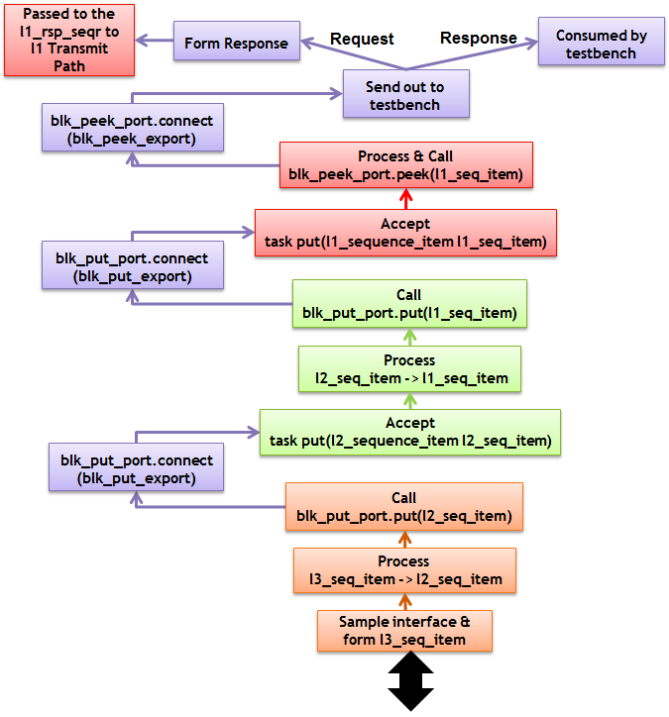

Figure 8 : Flow-diagram flow of sequence item in the Receive direction

The discussion so far concentrated on how the stimulus can be generated and layered on the 'transmit' side.
From a verification perspective, there would be a requirement to model and arbitrate the communication on the 'receive' path as well. For such communication across the different layers of the stack, the *uvm_blocking_put_port/uvm_blocking_put_export* can be leveraged. This is how this can be modeled. As the 'receive' side receives a transaction, the lower layer invokes the *put* method of the associated TLM port to transfer the transaction upstream and the upper layer provides the implementation of the *put* method where the transaction is accepted for further processing. This

processed transaction is then passed upwards subsequently.

The 'received' requests need to be made available to the response sequences. This is done so that the sequences can retrieve these and form the appropriate response outside the layers of the stack. Such processing and response creation should be passively managed. The *uvm_blocking_peek_port-export* TLM ports can thus be effectively leveraged for this. The topmost layer provides the implementation of the *peek* method to pass on the transaction and the response sequence would call the *peek* method and wait for a legal request to be made available. On a successful retrieval, the response gets driven on the response sequencer associated with each layer.

B. Verifying the Intermediate Layers

In a stacked architecture, verification of the intermediate layers is crucial. From the RTL development perspective, the maturity of the individual layers might be different. Verifying each of these layers in isolation is not going to be very efficient. Hence, the verification environment should be structured in such a way so that the same environment can be used for a robust verification of the intermediate layers as well as for the end to end verification of the entire stack. As mentioned earlier, the verification environment architecture should mimic the stacked model of the DUT. This allows stimulus, transformations, and responses to be mapped appropriately and can enable faster convergence when verifying either the 'transmit' or 'receive' paths. In addition to this basic requirement, here are a few items to be addressed when architecting such an environment.

- Individual verification component should have appropriate hooks/callbacks/ports which can be used to retrieve the transactions from any layer

- There should be a provision to hook up intermediate custom drivers which can then drive the interface between the layers. Such a 'virtual interface' would have to be created based on the interface between the layers. The additional drivers would be required as the communication between the layers of the complete stack on the verification side is otherwise done at the transaction level through TLM ports.

- The hooks provided in the structure and intermediate layer level monitors would be required to verify transformations across the layers.

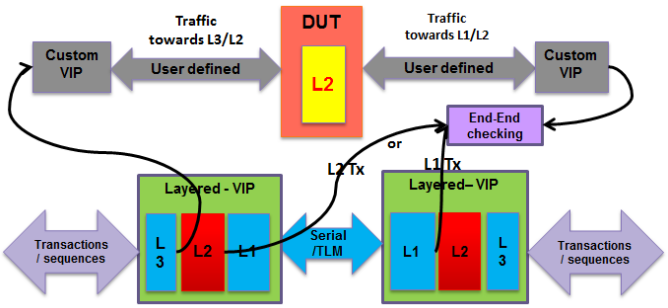

Figure 9: Example infrastructure for intermediate layer verification

The figure 9 above represents how an intermediate layer in the stack can be verified with the relevant infrastructure.
The layer being verified here is the 'L2" layer. Fragments from the L3 are transformed into Packets before entering Data Link Layer. Here there is a one-to-one correspondence for each Fragment-Packet pair. Packets in L2 are then appended with additional information which the other side of the stack has to decipher. This includes information related to the traffic class and credits. To verify such transformation on the LLI DUT, the user would need to create the appropriate stimulus coming in from L3 and then funnel it to the DUT through a custom virtual interface and Bus Function Model as shown above. Similarly on the 'receive' side, similar infrastructure needs to be enabled. Passive components on both side of the LLI stack of the verification infrastructure can extract the transactions for verifying the transformation on the L2 layer. With a symmetric architecture, the transformations across both the 'transmit' and 'receive' side can thus be verified.

C. Scoreboarding considerations: Verifying the intermediate and end-to-end transformations

In order to create a self-checking infrastructure, the scoreboarding techniques should be able to extract relevant transactions for analysis at each layer. This would enable the verification engineer to localize and converge on the problem area efficiently. Here are a few challenges that need to be addressed while creating a self-checking infrastructure for a layered architecture.

- How can one ensure that the transformations across the stack have occurred correctly? The complexity increases with multiple layers and with different configurations. The end transaction would be an aggregate of incremental information and aggregated transformations.

- One-to-many transformations in case of a disassembly on the 'transmit' side and a many-to-

one transformation of an assembly on the 'receive' side need to be additionally validated.

There might also be 'false pass' even if the end-to-end checking stamps the transformed transactions as correct. Hence, checking across individual layers is crucial. The use of UVM callbacks and TLM ports at each layer and the built-in UVM comparators help to address this need. The following diagram shows how a layered score-boarding approached can be created.

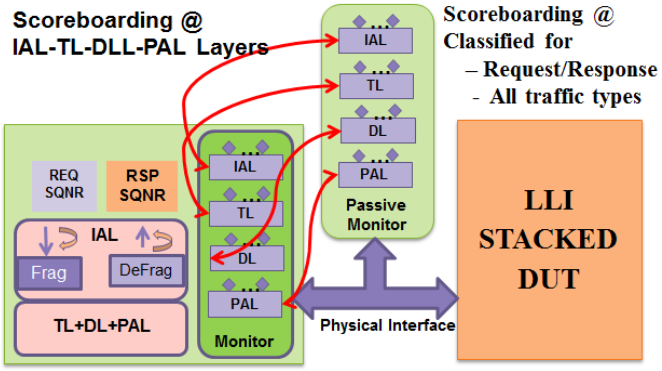

Figure 10: Verifying Transformations

The passive verification components which monitor the 'transmit' and 'receive' path extracts the signal level information and rebuilds the transactions at each layer and make them available for scoreboarding. This also ensures that the checks can be made at a significantly high granularity. Here are the different transformations that would need to be verified.

- End to end transformations in the 'transmit' and 'receive' paths

- Transformations across all the traffic types for MIPI LLI (Low Latency(LL), Best Effort(BE) and Service transactions (SVC))

For the transformations across the different traffic types, additional intermediate checks needs to be performed. These are:

- the request transmitted and received on the peer stack
- the response received and transmitted on the peer stack
- the request transmitted and the expected response received on the same stack
- the request received and the expected response transmitted.

```
class lli_scoreboard#(type T=uvm_sequence_item)
                              extends uvm_scoreboard;
  `uvm_component_utils(lli_scoreboard)
   ➔ The export to which Request Transmit Object is written ←
   uvm_analysis_export#(T) tx_export;
    ➔ The export to which Request Receive Object is written ←
   uvm_analysis_export#(T) rx_export;
    ➔ "in order comparator" to compare a written transfer objects ←
   uvm_in_order_comparator #(T, lli_comp#(T),
                  uvm_class_converter#(T)) comparator;

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    tx_export  = new("tx_export", this);
    rx_export  = new("rx_export", this);
    comparator = new("comparator", this);
endfunction

virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    ➔ Connecting Scoreboard export's to comparator export's ←
    tx_export.connect(comparator.before_export);
    rx_export.connect(comparator.after_export);
endfunction

virtual function void report_phase(uvm_phase phase);
  super.report_phase(phase);
  if (comparator.m_mismatches != 0 ) begin
      `uvm_error("report_phase",$sformatf("..."))
    end
    else begin
      `uvm_info("report_phase",$sformatf("..."))
    end
  endfunction
endclass
```

Figure 11 : Code snippet showcasing Generic Scoreboard

The parameterized scoreboard class is connected across the analysis ports available inside the master and slave monitor. A 'policy' class is used to compare the two transaction streams. The central element in policy-based design is a class template (called the host class), taking several type parameters as input, which are specialized with types selected by the user (called policy classes), each implementing a particular implicit method (called a policy), and encapsulating some orthogonal (or mostly orthogonal) aspect of the behavior of the instantiated host class. By supplying a host class combined with a set of different, canned implementations for each policy, a library can support an exponential number of different behavior combinations, resolved at compile time, and selected by mixing and matching the different supplied policy classes in the instantiation of the host class template. Additionally, by writing a custom implementation of a given policy, a policy-based library can be used in situations requiring behaviors unforeseen by the library implementer.

In the case of the LLI scoreboard, the policy class contains the static method, *function bit comp(T a, T b)* which returns TRUE if a and b are the same. Only the 'physical' fields of the transactions objects are compared using the relevant 'compare policy' which would be provided based on the different variations that were discussed earlier in terms of transformation types.

```
class lli_comp #(type T = int);
static function bit comp(input T a, input T b);
 `uvm_info("comp", $sformatf("Calling compare of
                  %0s with %0s", a.get_class_name(),
                  b.get_class_name(), UVM_FULL)
    return a.comp(b);
endfunction
endclass
```

Figure 12 : Generic policy class for Scoreboard

Thus, as the passive components extract out the relevant transactions and posts them through the TLM ports of the scoreboard, the built-in comparator of the scoreboard compares the request types received from the IAL, the LL and BE transaction data from TL, the SVC transaction data from the DLL and the PHIT types at the PAL and confirms the transformations by using different implementations of the 'compare' policy for each of these different types.

```
class lli_system_env extends uvm_env;

typedef lli_scoreboard#(svt_mipi_lli_transaction) trans_scbd;
typedef lli_scoreboard#(svt_mipi_lli_packet) pkt_scbd;
typedef lli_scoreboard#(svt_mipi_lli_frame) frame_scbd;

→ An instance of VIP AGENT to act as a LLI Master/Slave ←
svt_mipi_lli_agent mstr;
svt_mipi_lli_agent slv;

→ IAL Scoreboard Instances for request transaction ←
trans_scbd m_to_s_ll_req_xact_sb;
trans_scbd m_to_s_be_req_xact_sb;

function void lli_system_env::build_phase(uvm_phase phase);
 → Construct the IAL scoreboard instances ←
 m_to_s_ll_req_xact_sb = new("m_to_s_ll_req_xact_sb", this);
 m_to_s_be_req_xact_sb = new("m_to_s_be_req_xact_sb", this);
endfunction

function void lli_system_env::connect_phase(uvm_phase phase);
  → Connect the monitor to the scoreboard for Master LL request ←
  mstr.ial_mon.tx_ll_ta_req_xact_observed_port.connect(
                          m_to_s_ll_req_xact_sb.tx_export);
  slv.ial_mon.rx_ll_in_req_xact_observed_port.connect(
                          m_to_s_ll_req_xact_sb.rx_export);

  → Connect the monitor to the scoreboard for Master BE request ←
  mstr.ial_mon.tx_be_ta_req_xact_observed_port.connect(
                          m_to_s_be_req_xact_sb.tx_export);
  slv.ial_mon.rx_be_in_req_xact_observed_port.connect(
                          m_to_s_be_req_xact_sb.rx_export);
endfunction
endclass
```

Figure 13 : TLM connections for layer specific Scoreboarding

D. Effective "End of Test" Mechanism

Another critical aspect that needs to be modeled for stacked architectures is intelligent "end of test" mechanism. In the context of LLI, as the transactions flow across the stack, the verification components across each layer have to request for and relinquish control of transaction processing based on the protocol specifications. At any point in time, different components in the verification infrastructure might be actively processing transactions. Thus in such layered scenarios, the "end of test" mechanism needs to be managed appropriately so that the simulation ends gracefully when all the layers are through with their transformations. At the same time, it needs to be ensured that the entire "end to end" processing is complete as well.

For terminating simulations gracefully, the UVM base classes provide different mechanisms. Raising and dropping of 'objections' is the primary mechanism for determining the completion of different simulation phases. Objections must be explicitly raised and dropped by each phase implementation method as required. If no phase implementation method raises an objection to the end of the phase before the first non-blocking assignment, the phase will be immediately terminated and all of the implementation method threads will be killed. Environment components would typically raise the phase objection for every phase they implement. UVM also allows for setting of the drain time. It also has global timeout for each phase. All of these mechanisms serve a specific purpose which contributes towards a graceful exit from the simulation.

In the context of MIPI LLI, the 'objection' mechanism can be used to ensure that the entire traffic generated is adequately processed and reaches the design correctly. The drain time ensures that the latency the design pipeline brings is taken into consideration. This will cause the entire set of transactions to be flushed out on the IO before simulation ends. The 'global time out' mechanism causes the simulation to end only after a well-defined 'timeout' irrespective of unwanted simulation loops and design bugs. This ensures that an entire regression is not *bottlenecked* due to a few 'faulty' runs.

From the context of a layered architecture, here are some challenges tied to a graceful simulation exit. In UVM based testbenches, the termination of each phase and eventually the simulation is controlled through the raising and dropping of objections in the sequences. In a stacked architecture, where sequences are layered, the stimulus created by the upstream agent subsequently gets passed down to the lower layers. Here, the objections in the higher sequences are dropped once transactions are processed by the top layer. However, simulation cannot end at this point in time. The transfers have to go through the entire stack. Subsequently, once the transactions go through the physical interface, they have to be received by the highest layer of the peer stack from where again the response to a specific request has to be transferred back. Now, to ensure that the transactions have reached their respective destinations before simulation termination, the approach shown in the figure 14 can be followed.

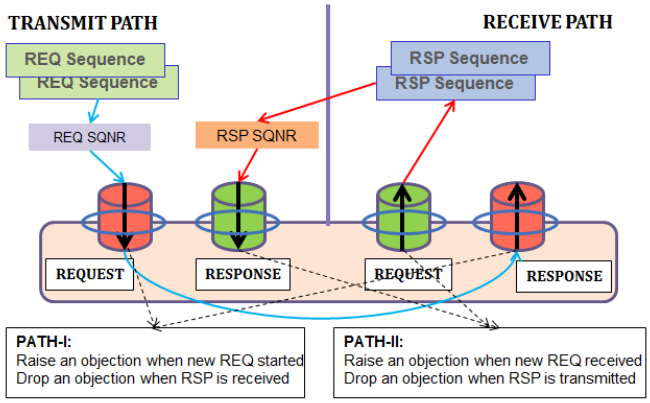

Figure 14: Effective objection mechanism

On the *Transmit Path*, the objection mechanism can be specified as below for a request which issued by the *req_sequence* (to ensure that its wait for the response at the receiving end)

- Raise/drop objection in the *req_sequence*'s *pre_body()* and *post_body()* methods as recommended by UVM. This controls the objection at the transaction generation-level at the topmost layer.

- Raise an objection in a callback or TLM port as soon as the request gets accepted by the highest layer.

- Once the generated request flows across the stack and the peer stack generates the response corresponding to the same, drop the objection (raised in the earlier step) once the response is received on the requesting end.

The Transmit side thus independently controls the dropping/raising of objections at its own end.
A generic method may be added to receive the *start* and the *end* event of a particular transaction at a specified layer. Now each layer is responsible for emitting the events appropriately which are tapped and passed down to the method controls objection mechanism and restricts premature end-of-test.

In the example code below, the *count* variable is used to keep track of requests for which responses are pending. So when the *count* transitions from 0-to-1 the objection is raised and then later dropped once the *count* variable transitions from 1-to-0. This can be generically used by any protocol irrespective of the fact it is stacked or non-stacked. For a stacked protocol, it has to be ensured that this infrastructure is present in each layer. As discussed earlier, any of the layers can be configured to be the top most layers. Hence, ensuring that the control of rising and dropping of objections can be configured to be assigned to any of the layers ensure that the stimulus generation requirements from each layer can be met. This shall make the testbench generic in case when any intermediate layer is promoted as the top-layer.

```
task system_env::objection_management(event
  start_ev, event end_ev, uvm_phase phase);
  int count = 0;
  fork
    begin
      while(1) begin
        @start_ev;
        if (count == 0) begin
          phase.raise_objection(this);
        end
        count++;
      end
    end
    begin
      while(1) begin
        @end_ev;
        if (count > 0) begin
          count--;
          if (count == 0) begin
            phase.drop_objection(this);
          end
        end
      end
    end
  join_none
endtask
```

Figure 15 : Code example to build a generic objection mechanism

On the *Receive Path*, the objection mechanism for the request received at the receive path of the stack should be as follows:

- Raise an objection as soon as request appears on the receive path.

- If there is a reactive sequence running on a *rsp_sequencer* to drive back the response, the objection can be a dropped as soon as the response is passed down on the transmit path.

In addition to the above, setting a drain time ensures that simulation is not terminated unless the required response is percolated across the DUT.

```
int stack_round_trip_time;

task main_phase(uvm_phase phase);
  phase.phase_done.set_drain_time
     (this, 2*stack_round_trip_time);
endtask
```

Figure 16 : Setting the drain time

The drain time is set up for the main phase, as the stimulus is set for the main_phase. The value should ensure such that all the transmitted frames are successfully received at the receiver as well as the monitor Interface.

The UVM callbacks registered on components on both sides of the stack keep track of the pending transactions as they flow through the two stacks. These would be used to appropriately raise and drop objections on either side of the stack.

As discussed earlier, a global timeout is necessitated to ensure simulation completes irrespective of any kind of undesired behavior during simulation. The *set_global_timeout* method can be used to appropriately configure the timeout value for each phase under different configurations. The usage is shown below:

```
`define HS_MODE_GLOBAL_TIMEOUT 5ms
`define LS_MODE_GLOBAL_TIMEOUT 25ms

function void test_base::build_phase(...);
  super.build_phase(phase);
   . . . . .
  ➔ Set the global timeout ←
  if(sys_cfg.mode == LS_MODE)
    set_global_timeout(`LS_MODE_GLOBAL_TIMEOUT);
  else
    set_global_timeout(`HS_MODE_GLOBAL_TIMEOUT);
endfunction
```

Figure 17 : Setting the Global Time-out

IV. Enabling debug abstraction

For layered protocols, we have to look beyond simple reports and trace file creation for debug. The stimulus supplied at the highest layer undergoes multiple transformations before finally being driven on the actual physical interface. As the complexity of debug increases, a few additional capabilities such as debug ports which especially help the RTL designers to have specific information at each layer in form of signals (not necessarily defined by specification) can help significantly. The examples of these signals can be ones which map to the change of the state of credit information as the simulation progress, the retry and the associated information in the physical layer etc. These signals can present a panoramic view of the entire stack in the waveform debugger thus enabling easier fault isolation.

As mentioned earlier, the initial input for the LLI stack is in the form of a *Transaction* and this undergoes various degrees of transformations before it is finally pushed out on to the physical interface via MPHY. Thus, it is crucial to enable some correlation for all the different object transformations to the debug interface to help converge on incorrect operation.

In addition to the debug ports mentioned earlier, the requirements are to provide a protocol-oriented analysis environment which should provide visualization and analysis at a higher abstraction. Leveraging the infrastructure enabled through UVM base classes, this can be enabled through the effective dumping of "protocol objects". A "protocol object" would be any description of data that is found in a protocol specification. To ensure that there are enough configurable hooks in the VIP, the layered component hierarchy is interspersed with UVM callbacks at all interesting execution points. These callbacks can be leveraged to dump an XML trace of the simulation enriched with the relevant details of the 'protocol objects' across the different layers. As these points are all 'protocol aware' and the UVM testbench is aware of the levels of abstraction at these different points and of the different transformations, the appropriate information can be dumped into the XML traced as mentioned earlier. Thus a tool like the Protocol Analyzer can efficiently analyze this information and provide an Interactive frontend to debug protocol behavior. The different displays that are available for visualizing protocol objects feed on the underlying methodology layer to provide a unique set of analysis capabilities. For example, as shown in the figure 18 below, the Object Timeline Display emphasizes the temporal relationship between the protocol objects in the view; the Object Tree/Table Display emphasizes the hierarchical relationship between objects and the Object Tree. Display focuses on the field attributes and values of the protocol objects. Here we observe how a single *Transaction*, leads to the formation of LLI stack compatible 3 *Fragments*, which further translates on one-to-one basis into *Packets* and *Frames* and *PHIT's*.. Thus, such graphical representations accelerate the investigation of protocol behavior by providing protocol-oriented analysis and debug capabilities in the associated verification environments. These capabilities not only reduce the time required to identify the source of bugs that can be directly attributed to protocol behavior, but also provide insight into aspects of the operation of a layered design that might otherwise be missed.

The first snapshot shows how the different transformations can be correlated. The next important aspect is to be able to closely monitor the content of each of these 'transformed' objects. This can be shown in the subsequent snapshot in Figure 19. Both these views leverage the capabilities that the UVM base

classes provide in terms of printing the content of individual class objects in the format desired.

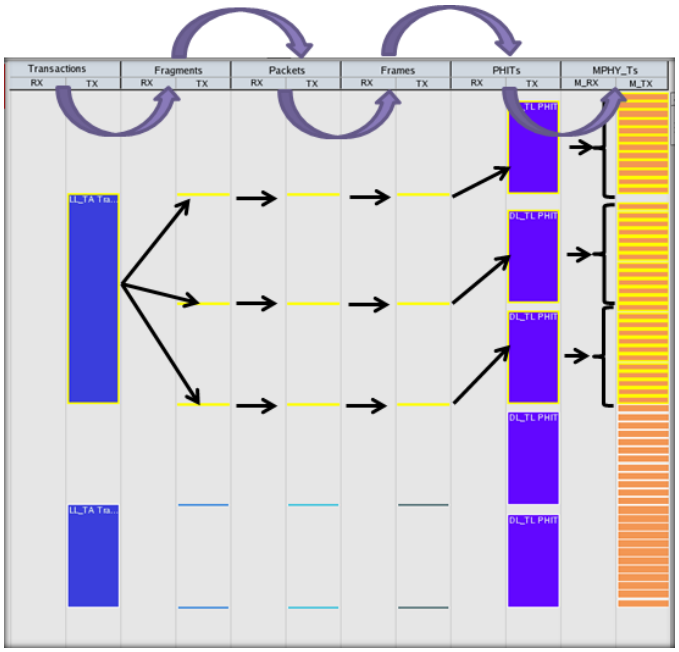

Figure 18: Effective transaction debugging - I

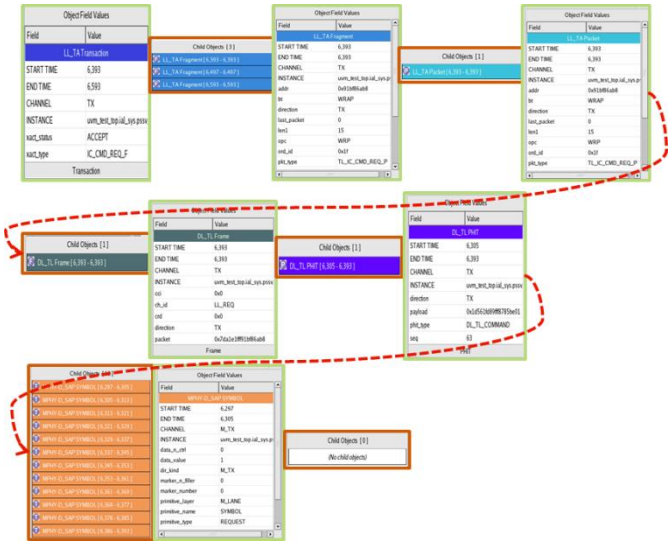

Figure 19: Effective transaction debugging – II

IV. Summary

As the complexity of protocols continues to increase and evolve, the infrastructure required for the verification of the same needs to scale up in sophistication as well. With the stacked protocol model becoming popular across these new protocols and standards, it is important to understand the challenges associated with the verification of such a stacked model. Robust verification architecture for one such model which can address the challenges that such a layered model brings up can then be leveraged across multiple protocols. For examples, the architecture described here can be leveraged for a stacked protocol like Unipro which bears a close relationship with LLI in terms of functionality. Similarly, there are other MIPI protocols such as UFS, CSI-3 etc. which would have similar requirements. Methodologies such as UVM have been undergoing continuous evolution to keep up with the many complex requirements in functional verification of complex devices. The capabilities provided for by the UVM library with respect to sequence layering, TLM communication, distributed phasing; configuration management provides the vehicle to build the required capabilities to address the verification requirements l to create robust verification architecture for stacked protocols.

V. References

[1] MIPI® Alliance Specification for Low Latency Interface (LLI): Version 1.0 – 26 January 2012
[2] UVM User Guide
[3] UVM Reference Manual
[4] Accellera Verification IP Technical Subcommittee Documents - http://www.accellera.org/apps/org/workgroup/vip
[5] UVM World Website http://www.uvmworld.org/
[6] Synopsys UVM CES Training
[7] Verification Martial Arts Blog
http://www.vmmcentral.org/vmartialarts
[8] Discovery MIPI LLI User Guide
[9] VIP Café : http://www.vip-central.org/blog
[10] Sharma A, Goel P, Acharya S, Luo R, Varghese R, Mohammed P, "ACE'ing the Verification of a Coherent System Using UVM", Proceedings of DVCon, 2012.