# Layering Protocol Verification: A Pragmatic Approach Using UVM

Rahul Chauhan (rchauhan@broadcom.com)
Gurpreet Kaire (gpsingh@broadcom.com)
Broadcom, Inc. San Diego, CA - USA
www.broadcom.com

Ravindra Ganti (rkganti@synopsys.com)
Subhranil Deb (sdeb@synopsys.com)
Synopsys, Inc. Mountain View, CA - USA
www.synopsys.com

**ABSTRACT**

*Layering protocols are modeled using layering structures that mirror the protocol layers. There are significant challenges in modelling verification components for layering protocols such as (1) reuse, (2) scalability, (3) controllability, and (4)observability. Furthermore, there may be requirements for complex test scenarios where a great deal of interaction is required between test sequence execution and response. It is important that the test sequences be provided with fine grain control of the desired verification components to execute the required complex test patterns for protocol verification at various layers. In this work, we present a pragmatic approach using Universal Verification Methodology that we developed for layering protocol verification to address the challenges mentioned above. This framework provides (1) a rich set of controls for layering drivers and sequencers to allow interactive complex test pattern generation and verification, (2) the ability to inject errors at any given layer without having to modify the underlying sequences, (3) the ability to run any given layer test sequence from a top-level virtual sequencer, and (4) the ability to perform peer-to-peer and complete protocol stack verification.*

***Keywords – UVM; protocol; layering; message; packet; frame; sequence; passthru-sequence; peer-to-peer, LLC; MAC; delayering; ACK.***

# Table of Contents



# List of Figures

# I. Introduction

Communication protocols are modeled as layers, and these layers are often labelled using the popular Open systems interconnection model. For transmission, the information flows from upper layers downstream to lower layers and for reception, the information flows upstream from lower layers to upper layers. Each layer services the upper layer and performs certain tasks based on the protocol defined for the respective layer. The information that flows through the various layers is subjected to (1) Segmentation, (2) Encapsulation, (3) Transformation, and so forth. For example, as illustrated in Figure 1, user-defined data in the form of messages can first be segmented into packets. The packets can then be transformed into LLC frames. LLC frames can, inturn, be transformed into MAC frames, then transmitted over a physical interface.

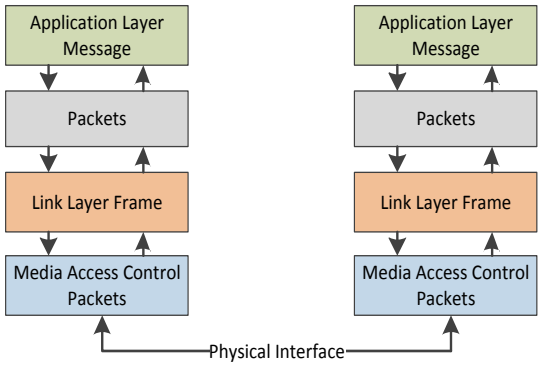


*Figure 1.* ***Layered Protocol Data Flow***

While modelling the protocol Verification IP, the following items have to be considered: (1) maintain independent structure for layers so that each layer can be controlled and observed independently [1], (2) make drivers adaptive for each layer to be able to enable/disable selective portions of the transmission and reception process, (3) provide flexibility from the top-level virtual sequencer to execute any underlying sequences selectively, and (4) provide the flexibility to inject errors at any given layer without having to modify underlying sequences.

# II. Layering Structure

A reusable and scalable implementation for verifying layering protocols is achieved through (1) layering agents, (2) layering drivers [1], and (3) delayering monitors. In Figure 2, the drivers of the higher layers are connected to lower layer pass-through sequences for transforming the higher-layer data stream to the lower-layer data stream. Monitors are delayered to carry out the inverse transformation of the lower-layer data stream to the higher-layer data stream.

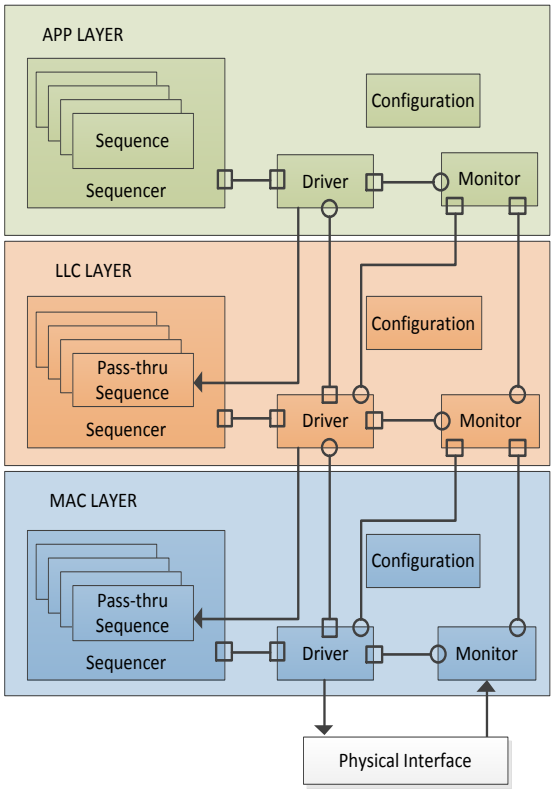


*Figure 2.* ***Upstream and Downstream Connection in Layered Architecture***

Such a setup can verify individual layers with a peer-to-peer testbench (block level) and also use the complete protocol stack to run top-level scenarios.

## III. Adaptive Drivers

The challenges lie in the drivers for handling error injection scenarios for layered protocols that take care of certain key aspects of the protocol such as (1) flow control and (2) internal automatic response generation to the frames received where higher layers are insulated from handling such activities. Providing control for the driver to enable and disable certain portions of the transmission and reception process can effectively deal with situations such as complex error injection scenarios without breaking the driver's core functionality. For example, the LLC Layer in Figure 2. is responsible for sending frames with sequence numbers in ascending order and also expects to receive an Acknowledgement from the other endpoint for the frames transmitted. When frames are sent, they are buffered and a timer for each frame is activated and deactivated upon reception of an ACK. If no ACK is received, the LLC Layer will retransmit the frames from the retry buffer after timeout. Similarly, there might be other test conditions where a corrupted frame from an endpoint is sent and no ACK from the other endpoint is received, in which case, the corrupted frame is not to be retransmitted. This requires fine-grain control of the driver to disable the timer when sending corrupted frames. In this way, corrupted frames would not be retransmitted if an ACK is not received. The code snippet in Figure 3 shows the control knobs in the transaction descriptor class that are used by the adaptive driver.

```
class llc_frame extends
uvm_sequence_item;
...
rand bit bypass_model; //To Bypass
Model
rand bit wait_before;  //Wait Before
driving
rand bit wait_after;   //Wait After
driving
...
constraint llc_frame_default_c {
  bypass_model == 0;
  wait_before == 0;
  wait_after == 0;
}
...
endclass : llc_frame
```

*Figure 3.* ***Transaction Class Code Block***

Also, there might be some other test conditions where, (1) wait before receiving a response from an endpoint driving a frame, (2) wait after receiving a response from an endpoint driving a frame, and (3) enable/disable the driver's response from the test sequences to model complex test scenarios. For instance, to model no-response behaviour for the driver such that no ACKs are sent out for the frames received, generate a special sequence item (configuration frame) from a test sequence with the constrained property "bypass_model == 1" which disables the driver's response mechanism. In order to put the driver back into auto-response mode, generate a special sequence item (configuration frame) from the test
sequence with the constrained property "bypass_model==0" which enables the driver's response mechanism. The code snippet in Figure 4 captures key hooks in the driver describing how to model the desired behaviour.

```
class llc_driver extends uvm_driver;
...
local uvm_event llc_frame_rcvd_ev;
```

```
virtual task run_phase(uvm_phase);
  fork
    this.tx_driver();
    this.rx_driver();
  join_none
endtask : run_phase

virtual task tx_driver();
  forever begin

seq_item_port.get_next_item(llc_frame)
    //-- Wait Before
    if (llc_frame.wait_before)

this.llc_frame_rcvd_ev.wait_ptrigger();
    //-- Process frame for transmission
    if (!llc_frame.bypass_model)
      this.send_frame();
    else
      this.send_corrupt_frame();
    //-- Wait After
    if (llc_frame.wait_after)
      //-- Clear Existing Event
      if
(this.llc_frame_rcvd_ev.is_on())
        this.llc_frame_rcvd_ev.reset();

this.llc_frame_rcvd_ev.wait_ptrigger();
    ...
    seq_item_port.done();
  end
endtask : tx_driver

task send_frame(llc_frame);
  //-- Convert LLC to MAC
  this.convert_llc2mac(llc_frame);
  //-- Send to MAC passthru-sequence
  this.send_llc2mac(llc_frame);
  //-- Selective Enable Mechanism
  //-- Activate Timer for Flow Control
  if (!llc_frame.bypass_model)
    this.set_timer(llc_frame);
endtask : send_frame

virtual task rx_driver();
  forever begin
    this.frame_rcvd_ev.wait_ptrigger();
    //-- Selective Enable/Disable
Response
    if (!this.bypass_model)
      this.process_rx_frame();
  end
endtask: rx_driver
endclass: llc_driver
```

*Figure 4. **Driver Class Code Block***

We also used an alternate way to achieve a similar result for a different higherlayer, whereby the *uvm_sequence* class method *get_response()* is used to handle transactions in the sequence. Exceptions that are handled in the driver are defined in the transaction class. The driver is implemented in such a way that after sending the sequence or request (downstream) transaction, it either waits for the response (upstream) transaction or continues based on the user setting of the expect_response in the transaction class. In a scenario where the design is expected to send a response but does not, the loop doesn't wait infinitely. A timeout response transaction is created and sent back to the sequence indicating the missing response and helps the testcase to proceed further. For a case where a response is not set and the sequence does not wait for the *get_response()* method, a user has more control over the responses and can set the subsequent request transactions accordingly. The code snippet in Figure 5 captures important steps within the transaction class, sequence, and the driver.

```
class app_tr extends uvm_object;
   bit  expect_response;
   bit  expect_error;
endclass : app_tr

class app_seq extends uvm_sequence
#(app_tr);
  int msg_cnt;

  task body;
    repeat(3) begin
      `uvm_create(tr);
      if(msg_cnt == 0)
         // Good Tr, Wait for response
        tr.expect_response = 1;
      if(msg_cnt == 1)
```

```
      // Known Bad Tr, No Wait for
    response
        tr.expect_error = 1;
      if(msg_cnt == 2)
        ...
      `uvm_send(tr);
      get_response(rsp);
    end
  endtask : body

endclass : app_seq

class app_drv extends uvm_component;
  task main_phase();
    forever begin
      seq_item_port.get(req);

      send(req);
      ...
     // Checking if response is ex-
    pected
      if(req.expect_response) begin
        wait(rsp_recd_ev);
        $cast(rsp, rsp_recd_tr);
        rsp.set_id_info(rsp_recd_tr);
      end
      else
       // Send the req back, kind of
       dummy
        rsp.set_id_info(req);
      ...
      item_done(rsp);
    end
  endtask : main_phase
endclass : app_drv
```

*Figure 5.* ***Alt. Driver Class Code Block***

Apart from this, the driver supported automatic response generation based on the current configuration of the agent. This allowed the user to trigger a valid request and leave the rest to the driver's intelligence. Potentially, this could be used to compare the incoming response from design as well.

## IV. Flexible Virtual Sequencer

The virtual sequencer used in this approach allowed us to manipulate the flow control as per our requirements. The philosophy behind this approach was to allow the user to test scenarios which were either directed or random. This flow control was achieved by monitoring the implementation ports of each layer in the virtual sequencer. This gave us visibility into all the transactions at each layer, from either side. The trick is to create wrap-around tasks for these monitor port transactions which could be manipulated to allow the flow control to be stalled until a certain protocol state is reached in the simulation.

The virtual sequencer also had the capability of injecting errors at any level directly from the sequences. Since we were following the layering driver [1] approach, we only had access to higher-layer sequences from the virtual sequence. In order to overcome this and allow the user to inject errors at any level without modifying the underlying sequence, a virtual sequence was provided a handle the pass through sequence of each lower layer. We could then fine-tune the lower layer packets to inject errors as per our needs without modifying the flow control. The error injection technique along with capturing monitor state information, provides great flexibility in verifying complex scenarios. The code snippet in Figure 6 describes how the virtual sequencer is modelled, and the code snippet in Figure 7 describes how the virtual sequence is modeled to achieve the flexible behaviour.

```
class virtual_sqr extends
uvm_sequencer;
  //Handles for all sequencers
  mac_sequencer mac_sqr;
  llc_sequencer   llc_sqr;
  app_sequencer app_sqr;

 //Declare monitor imp ports
  uvm_analysis_imp_mac  #(mac_frame,
virtual_sqr) mac_export;
……  llc_export;
…… app_export;
```

```
//Use the monitor port to create wait
conditions for the sequences

//Wait task for waiting on one particu-
lar frame_kind from the monitor ports
task wait_for_frame(frame_kind_e
frame_kind);
  wait_for_frame_event(frame_kind);
  ….
  process_frame_for_sequence();
endtask : wait_for_frame

//Implentation port write function im-
plantation.

function write_app();
   if (frame.frame_kind == frame_kind)
     …..
    emit_frame_event();
endfunction : write_app

 function write_llc();
 endfunction : write_llc

function write_mac();
endfunction : write_mac

endclass : virtual sqr
```

*Figure 6.* ***Virtual Sequencer Class Code Block***

```
class virtual_seq extends uvm_sequence;

 function new();
    get_handle_for_llc_passthru_seq();
    get_handle_for_mac_passthru_seq();
 endfunction : new

 task body();
   //Start application  sequence on ap-
plication sequencer

app_seq_1.start(p_sequencer.app_sqr);

app_seq_2.start(p_sequencer.app_sqr);

   //Wait for frame response for second
    application seq by calling parent
    sequencer task
p_sequencer.
wait_for_frame(app_frame_kind);

   //Optionally inject llc error using
    pass  through sequence in third
   application sequence
   fork
     llc_passthru_seq.inject_error = 1;
   join_none
  app_seq_3.start(p_sequencer.app_sqr)

endtask : body

endclass : virtual_sequence
```

*Figure 7.* ***Virtual Sequence Class Code Block***

## V. Results

Use of these techniques improved the efficiency of testbench verification and test case creation, ultimately delivering a modular, reusable, and robust testbench. The highlights of this approach were to show how seamlessly everything fell in place with good planning and architecture. Figure 8 shows how different development tasks were shared and executed.

Overall:

- Five test environments were created – Three peer-to-peer; one each for stack-to-stack and design.
- ~220 man-days of development time with three engineers working at three different locations and in two time zones.
- Two standard test suites implemented with more than 100 directed and random sequences.

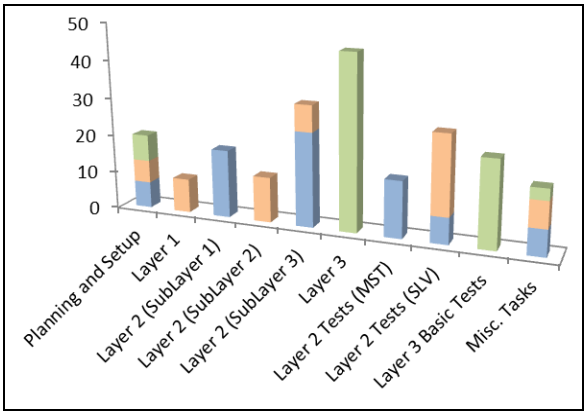


*Figure 8.* ***Effort Bars During Execution Phase***

## VI. Conclusion

The motivation for this paper is to share the concepts and simple techniques that were implemented and also share the benefits we achieved with the methodology. The techniques described in this paper can be extended to create even more robust and complex test pattern scenarios. The focus of this methodology was to have maximum controllability at every layer of abstraction while still having an automated test flow.

## VII. References


[1] Synopsys, Inc., Beyond UVM: Creating Truly Reusable Protocol Layering.

[2] Accellera, Universal Verification Methodology (UVM) 1.1 User's Guide.