Beyond UVM: Creating Truly Reusable Protocol Layering

Janick Bergeron, Fabian Delguste, Steve Knoeck, Steve McMaster, Aron Pratt, Amit Sharma
Synopsys, Inc
Mountain View, CA

***Abstract*—Protocols that are transported by other lower-level protocols are modeled using a layering structure that mirrors the layering of the protocol. In UVM, it is recommended that the layering be performed using a *layering sequence*. However, the many examples of protocol layering sequences in the UVM literature show that it requires implementation techniques that are not scalable and often not reusable. This paper details a UVM protocol layering approach that uses a *layering driver* as an implementation that is both scalable and reusable.**

Keywords—UVM; protocol; layering; delayering; reusable; scalable; sequence

I. Introduction

Communication protocols are specified and implemented according to layers. These layers are often labeled using the popular OSI model**Error! Reference source not found.**. A higher-layer protocol is transparently transported on a lower-layer protocol. That lower-layer protocol may in-turn be transparently transported on an even lower-layer protocol. For example, as illustrated in Figure 1. , TCP protocol packets can first be segmented into IPv4 frames; the IPv4 frames can then be encapsulated into Ethernet frames then transmitted over a XAUI interface onto a fiber optic medium. Another example illustrated in Figure 2. , USB transfers are composed of USB transactions which are composed of USB packets.

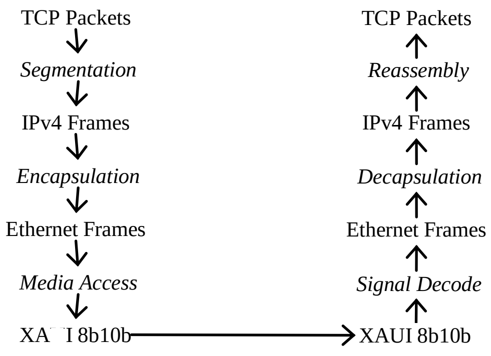

Figure 1. TCP/IP over Ethernet

Because the transport layer is transparent to the higher-layers it carries, a higher-layer protocol may be transported on a variety of lower-level protocols. For example, a TCP packet may be transported over an IPv4, IPv6 or PPP protocol. Conversely, a lower-layer protocol may transport different higher-layer protocols, often at the same time. For example, an Ethernet link may transport a mix of IPv4 frames, IPv6 frames or UDP packets. To complicate matters even further, lower-level protocols can be transparently tunneled through a higher-layer protocol. For example, an IP stream (itself carrying a variety of higher-layer protocols) can be encrypted then wrapped into TCP packets and tunneled through a TCP transport layer to be decrypted and processed at the other end as if they had been natively transported.

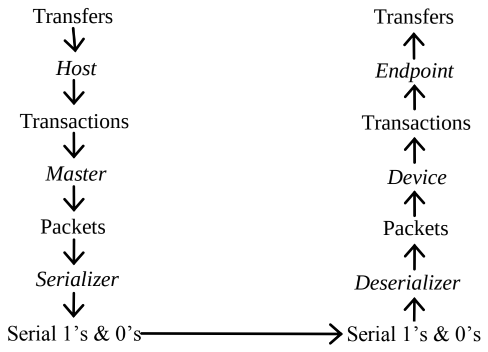

Figure 2. USB Protocol Layers

When implementing protocol verification IP, it is important that the independence of the protocol layering structure be maintained, The output of a higher-layer protocol VIP can thus be transported by different lower-layer protocol VIPs. Similarly, a single instance of a lower-layer protocol VIP must be able to transport multiple higher-layer protocols coming from multiple instances of different higher-layer protocol VIPs. Conversely, lower-layer protocol monitors must be able to feed a variety of higher-layer protocol monitors.

II. Protocol Layering in UVM

A. *UVM Stimulus Generation*

As illustrated in Figure 3. UVM dictates that stimulus be generated using sequences executing on a sequencer. While executing, sequences create transactions using a random, algorithmic or directed specification that are then executed by a driver, usually by transmitting them over physical signals.

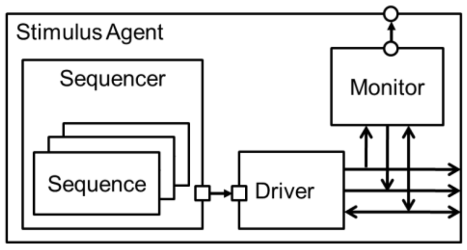

Figure 3. Architecture of UVM Stimulus Agent

Conversely, UVM dictates that a monitor observes the execution of transactions and reports them on an analysis port, usually using the same transaction objects that are exchanged between the sequencer and driver.

This abstraction level works well if the protocol functionality under verification is at the same layer as the transactions that are executed by the driver. But if it is at a higher layer, it becomes cumbersome to create stimulus that is relevant to the verification objectives at hand, and to interpret the low-level transaction stream into higher-level response. It thus requires more effort and increases the time required to verify the design to the desired degree of confidence.

B. Layering Sequences

Transaction-based verification is about abstracting the low, signal-level operations into higher-level transactions. The same abstraction process can be recursively applied to lower-level transaction: verification can be performed at a higher level of abstraction by abstracting the low-level transactions into higher-level ones.

One possible approach would be to create a set of monolithic protocol VIPs, each using a different higher-level protocol on the testbench side that can be connected to the same physical signals. But this would require one VIP for every combination of higher-level and low-level protocols. Furthermore, it would be next-to-impossible to combine different higher-level protocols onto the same lower-level protocol.

That is why UVM recommends the use of *protocol layering sequences*. Just like drivers execute transactions by wiggling physical signals, a layering sequence will execute higher-layer transactions by executing a set of lower-level transactions. Similarly, a delayering monitor will observe lower-level transactions on a lower-level analysis port and report higher-layer transactions onto its analysis port.

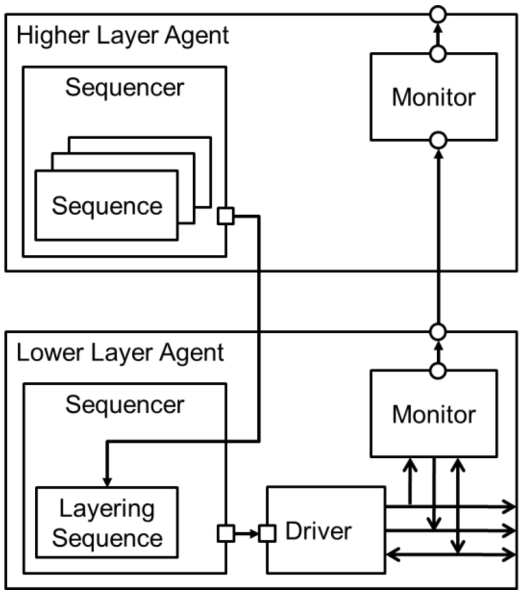

Figure 4. Layered Sequencers and Monitors

Because layering sequences have a function that is very similar to drivers, they are significantly different from usual sequences in a several aspects.

Firstly, layering sequences execute continuously throughout the *run* phase. Regular sequences are started on demand and terminate once they have completed their stimulus requirements by returning from their *body()* task. But just like drivers never return from their *run_phase()* task, layering sequences must always be available to execute higher-layer transactions and thus cannot ever return.

Secondly, layering sequences must be started at the beginning of the *run* phase so they can be ready to execute higher-layer transactions as soon as possible. This is just like drivers implementing their functionality in their *run_phase()* task which is automatically started at the beginning of the *run* phase.

Thirdly, UVM recommends that layering sequences accept higher-layer transactions to be executed via a *sequencer port*. Whereas regular sequence take their input from local variables defined before the execution of their *body()* task, layering sequences take their input using the same mechanism as a driver. As shown in Figure 4. , this allows a higher-level sequencer to be connected to a layering sequencer in the same way it could be connected to a driver.

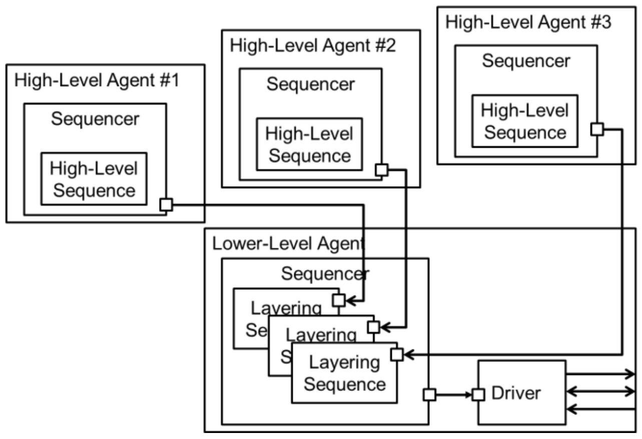

Figure 5. Complex Protocol Stack

A layering sequence, even though it behaves like a driver, remains a sequence. And because a sequencer can concurrently execute multiple sequences, it is possible to execute multiple layering sequences on the same sequencer, each layering different protocol or additional streams of the same protocols onto the same lower-layer protocol. And because a layering sequence is fed by a higher-layer sequencer, that higher-layer sequencer may also be executing a layering sequence, executing even higher-layer transactions. As shown in Figure 5. , this allows an arbitrary number of protocol layers to be stacked and multiplexed in arbitrary (but compatible) combinations.

Layering sequences can concurrently execute with regular sequences to inject lower-level stimulus (such as errors or spurious traffic or protocol exception transactions) in a protocol stack. Furthermore, the relative priority of the layering and normal sequences can be adjusted to shape the resulting traffic. Although traffic shaping is a fascinating topic, it is outside the scope of this paper.

III. Prior Art

Several examples of layering sequences have been published[1][3]. However, each exhibit limitations that prevent them from being truly reusable and scalable.

A. *Layering inside one Sequencer*

Section 6.5.2.3.1 of the UVM User's Guide[2] shows how a higher-layer transaction is created, randomized then translated into lower-level sequence items that are then executed on the driver:

```
task body();
   `uvm_create(hli)
   hli.randomize();
   send_high_layer(hli);
endtask: body

task send_high_layer(hli_typ hli);
   while (...) begin
      `uvm_create(lli);
      // Slice and dice hli to form lli's
      lli.data[i] = ...;
      ...
      `uvm_send(lli)
   end
endtask : send_high_layer
```

This layering sequence is reusable only by extending it to redefine its *body()* task and create different higher-layer transaction items to reuse the layering *send_high_layer()* task. Because SystemVerilog lacks multiple inheritance, it is impossible to leverage existing higher-layer sequences and combine them with the layering *send_high_layer()* task without re-implementing them for each lower-layer sequence. In an environment where N higher-layer protocols, each with S higher-layer sequences, need to be layered on M lower-layer protocols, it requires N x S x M additional sequences. Adding a new higher-layer protocol requires N x M additional sequences. Adding a new lower-layer protocol requires N x S additional sequences. This approach is clearly not scalable.

B. *Layered Sequencer*

Section 6.5.2.6 of the UVM User's Guide[1] shows how a layering sequence can pull higher-layer items from a sequence port on the parent sequencer accessed via *p_sequencer*, translate them into lower-level sequence items and execute those.

```
class hli_to_lli_sqr extends
      uvm_sequencer#(lli_typ);
   uvm_seq_item_pull_port#(hli_typ) hli_port;
endclass

class hli_to_lli_seq extends
      uvm_sequence#(lli_typ);
task body();
   forever begin
      p_sequencer.hli_port.
         get_next_item(hli);
      while (...) begin
         `uvm_create(lli)
         // Slice and dice hli to form lli's
         lli.data[i] = ...;
         ...
         `uvm_send(lli)
      end
      p_sequencer.hli_port.item_done();
   end
endtask
```

This second example is more reusable than the first one: it only needs to be started on the appropriate sequencer without requiring any modifications. It is also more scalable as the higher-layer sequences are executed independently on the higher-layer sequencer and the layering sequence takes care of the layering. All existing higher-layer sequence can be reused, without modifications on any layering sequence. In an environment where N higher-layer protocols, each with S higher-layer sequences, need to be layered on M lower-layer protocols, it requires N x M additional sequences.

The problem though lies in the location of the higher-layer sequencer port: in a specialization of the lower-level sequencer. This is approach hinders reusability and scalability in three ways:

First, should the lower-layer sequencer not be of the appropriate type, it will not be able to execute the layering sequence. In an environment where N higher-layer protocols need to be layered on M lower-layer protocols, this approach requires N x M additional sequencer types (to go with the N x M additional layering sequences). Adding a new higher-layer protocol does not require any additional sequences. Adding a new lower-layer protocol requires N additional layering sequences. However, should the environment already have specialized a lower-level sequencer to provide configuration information to the normal sequences it runs, the single-inheritance nature of SystemVerilog will make this approach unusable.

Second, should it be necessary to layer more than one protocol stream on the same lower-layer sequencer, there is only one higher-layer sequencer port to connect to. It is not possible to connect more than one sequencer to the same sequencer port, thus it will be impossible to connect more than one sequencer to the layering sequence.

Third, should it be necessary to layer different protocols on the same lower-layer sequencer, it becomes necessary to declare multiple sequencer ports of different higher-layer protocol types in the same lower-layer sequencer to enable their respective layering sequence to refer to their respective higher-layer port. In an environment where N higher-layer protocols need to be layered on M lower-layer protocols, this approach requires N different sequencer ports in N x M additional sequencer types. Adding a new higher-layer protocol requires modifying M sequencer types. Adding a new lower-layer protocol requires one new sequencer with N sequencer ports and N new layering sequences.

Reusable? Yes. Scalable? Definitely not.

C. *Reference to Higher-Layer Sequencer*

The layering example shown in [3] recommends putting a reference to the high-layer sequencer in the layering sequence. The layering sequence then directly accesses the implementation of the *get_next_item()* and *item_done()* methods.

```
class hli_to_lli_seq extends
   uvm_sequence#(lli);
task body();
   forever begin
      hl_sequencer.get_next_item(hli);
      while (...) begin
         `uvm_create(lli)

         // Slice and dice hli to form lli's
         lli.data[i] = ...;
         ...
         `uvm_send(lli)
      end
      hl_sequencer.item_done();
   end
endtask
```

This approach can be a lot more reusable and scalable than the previous ones. Layering different protocols or multiple streams of the same protocol onto a lower-layer sequencer simply requires that the appropriate number of layering sequences be started. In an environment where N higher-layer protocols need to be layered on M lower-layer protocols, this approach requires N x M layering sequences. Adding a new higher-layer protocol requires creating only one new layering sequence. Adding a new lower-layer protocol requires N new layering sequences.

The major fault with this approach is that it violates the TLM connection methodology that is integral to UVM by calling the implementation methods directly. Further, should multiple layering sequence instances be erroneously connected to the same higher-layer sequencer, they will interfere with each other, resulting in difficult-to-explain behavior.

IV. DELAYERING

If protocols are layered on the way down, they must similarly be "delayered" on the way up. Reference [3] does show how a delayering monitor can observe lower-level transactions through an analysis export and report observed higher-layer transactions through its own analysis port.

```
function write(lli_typ lli);
   // Only consider lower-level items
   // that belong to us
   if (lli.dest != addr) return;

   // Collect lower-level items and
   // re-constitute higher-layer items
   hli.data[i] = lli.data;
   if (hli.data.size() == hli.len) begin
      ap.write(hli);
      hli = hli_typ::type_id::create("hli");
   end
endfunction
```

As shown in Figure 4. , delayering monitors are connected in a structure that mirrors the layering sequence structure. Because an analysis port can be connected to multiple analysis exports, as many delayering monitors can be connected as there are layering sequences running on the sequencer.

All of the sequence layering examples in the existing literature use a straight forward unidirectional protocol layering. The timing and content of the lower-level layer transactions are solely determined by the timing and content of the higher-level transactions. Unfortunately, the behavior of real-world protocols are often affected by the responses received from the lower layer. For example, executing a USB transaction involves the correct back-and-forth exchange of several USB packets. Should a reply packet contain a negative response or go missing, the transfer must be aborted. Or should forward packets be send without waiting for the preceding response, the rules of the protocol will have been broken. In other protocols, such as PCIe, a lower-layer transaction may not be sent unless the layering sequence possesses the necessary data flow credits.

Therefore, the sequence layering mechanism must be able to handle duplex protocol, where the result and timing of the layering is a function, not only of the upper-layer sequencer, but of transactions observed by the lower-level monitor. Thus, the layering sequence must be connected to the analysis port of the lower-level agent as well, as shown in Figure 6. .

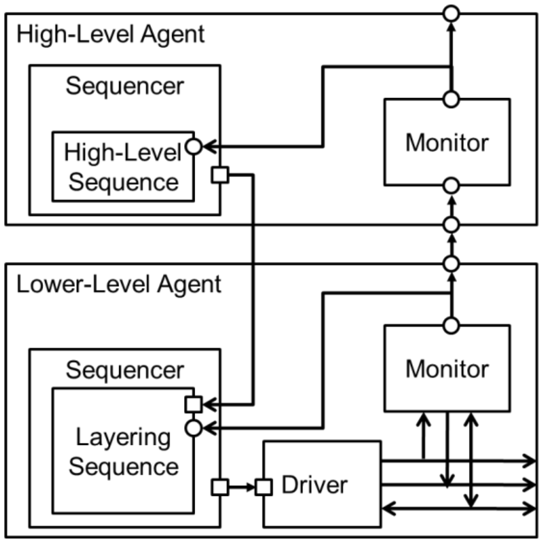

Figure 6. Layered Duplex Protocol

The only problem with connecting the analysis port of a monitor to the layering sequence (or any sequence for that matter) is that analysis exports must be implemented by a *write()* method located in a *uvm_component* type. Unfortunately, because sequences are not components, they cannot have analysis exports. That is why [3] encapsulates the layering sequence and delayering monitor in a layering *uvm_subscriber* component that includes an analysis export for the lower-layer protocol.

This puts the onus on the users to encapsulate each protocol layering operation into individual components so the analysis ports can be properly connected. In an environment where N higher-layer protocols need to be layered on M lower-layer protocols, this approach requires N x M layering sequences encapsulated in N x M components. Adding a new higher-layer protocol requires creating M new layering components. Adding a new lower-layer protocol requires N new layering components.

V. Layering Agents

All of the existing layering examples seem to assume that higher-level protocols can only be layered onto a lower-level protocol and that no agent already exists for that protocol. They assume that the user is free to create and instantiate a sequencer for those higher-level protocols. But what if a sequencer for that particular protocol already exists? What if it needs to be connected to the monitor or driver in specific ways because of the reactive nature of the protocol? What if the protocol transactions require configuration information from the agent context to be properly randomized? That is why UVM defines the *agent* as the smallest unit of protocol-level reuse, not the sequencer or the monitor.

Furthermore, some higher-level protocols may very well have a physical transport implementation and thus need not necessarily be layered. For example, Ethernet frames can be transmitted over a variety of physical interfaces for which a driver would be available. But they may also be transported by another protocol (for example Ethernet-over-PPP) and thus would require layering.

Layering agents is a more efficient reuse strategy, as it enables reusing the sequencer and monitor they contain instead of instantiating and connecting new one. The only component that needs replacing is the driver: should it remain in place, it will compete with the layering sequence for the agent's sequence items and break the execution of the higher-layer protocol.

VI. A Reusable and Scalable Implementation

Because the driver in a protocol agent needs to be replaced or shut down when that protocol needs to be layered, and since layering sequences connect sequencers and are therefore essentially components, the layering should be implemented in a *layering driver*, more specifically in its *run_phase()* method. A layering driver is implemented using the same familiar techniques used to implement a "regular" driver, except that a sequence item is executed in terms of lower-level transactions instead of pin wiggling through a virtual interface.

```
class h2l_layering_driver extends ll_driver;
...
virtual task run_phase(uvm_phase phase);
   forever begin
      seq_item_port.get_next_item(hli);
      while (...) begin
         `uvm_create(lli)

         // Slice and dice hli to form lli's
         lli.data[i] = ...;
         ...
         execute(lli)
      end
         seq_item_port.item_done();
      end
   endtask

   endclass
```

The layering driver should be implemented as an extension of the default agent driver. This will make it possible to replace the original driver in the higher-level agent using the factory.

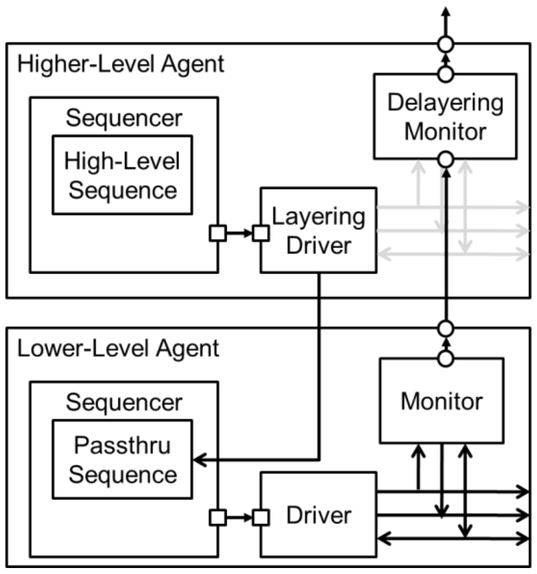

Figure 7. Layering Drivers

The layering driver should be connected to the lower-layer sequencers using a simple *passthru sequence*. That passthru sequence is functionally equivalent to the virtual interface of a "regular" driver and thus should similarly be passed via the configuration database. A passthru sequence is used in preference to using the *uvm_sequencer_base::send_request()* or *uvm_sequencer_base::execute_item()* methods so it can be configured with state information, such as priority, to shape the lower-level protocol traffic.

```
class ll_passthru_seq extends
   uvm_sequence#(lli_typ);

   lli_typ req;
   int priority = -1;

   task body();
      forever begin
         wait (req != null);
         start_item(req, priority);
         finish_item(req, priority);
         req = null;
      end
   endtask
endclass

class h2l_layering_driver extends ll_driver;

virtual task run_phase(uvm_phase phase);
  ll_passthru_seq ll_seq;
  uvm_config_db#(ll_passthru_seq)::get(this,
     “seq”, ll_seq);
    forever begin
       seq_item_port.get_next_item(hli);
       while (...) begin
          `uvm_create(lli)

          // Slice and dice hli to form lli’s
          lli.data[i] = ...;
          ...
          ll_seq.req = lli;
          wait (ll_seq.req == null);
       end
       seq_item_port.item_done();
    end
 endtask

 class layered_env extends uvm_env;
 ...
 ll_passthru_seq ll_seq;

 function build_phase(uvm_phase phase);
   hl = hl_agent::type_id::create(“hl”, this);
   set_inst_override_by_type(“hl.drv”,
      ll_driver::get_type(),
      h2l_layering_driver::get_type());
 endfunction

 function connect_phase(uvm_phase phase);
   ll_seq = new();
   uvm_config_db#(ll_passthru_seq)::set(this,
     “”, “seq”, ll_seq);
 endfunction

 task run_phase(uvm_phase phase);
    ll_seq.start(hl.sqr);
 endtask
 ...
 endclass
```

Because the layering driver is a component, it can have and implement an analysis export for duplex protocols.

```
 class h2l_layering_driver extends ll_driver;
    uvm_analysis_imp#(lli_typ) ap;

    function write(lli_typ lli);
       // Only consider lower-level items
       // that belong to us
       if (lli.dest != addr) return;

       // Process the response/credits
       ...
    endfunction
 endclass
```

Different higher-level protocols or multiple streams of the same higher-level protocol may be layered onto the same lower-level agent by simply starting multiple concurrent passthru sequences on the lower-level sequencer.

In an environment where N higher-layer protocols need to be layered on M lower-layer protocols, this approach requires N x M layering drivers and M passthru sequences. Adding a new higher-layer protocol requires creating M new layering drivers. Adding a new lower-layer protocol requires only one passthru sequence.

The concept of the layering driver is also in keeping with the current concept of a driver in UVM. What is a driver if not a layering device between a sequence item and physical signals?

VII. Summary

TABLE I. shows a comparative summary of using layering sequences vs. layering drivers for N higher-level protocols and M lower-level protocols, and the incremental effort for adding a high-level and low-level protocol. It clearly shows that using layering drivers is the more scalable approach.

TABLE I. Comparison of Layering Approaches

Number Of	**Layering Sequence (as in [3])**			**Layering Driver**		
Protocols	NxM	N+1	M+1	NxM	N+1	M+1
Sequences	*NxM*	*M*	*N*	*M*	*0*	*1*
Sequencers	*N*	*1*	*0*	*0*	*0*	*0*
Components	*NxM*	*M*	*N*	*NxM*	*M*	*N*

VIII. Looking Forward

In most networking application, it is necessary to verify the dynamic reprovisioning of components and the ability of components to adapt to changes in the protocol topology. For example, adding and removing devices and hubs from a USB network is a normal operation of that protocol. Similarly, it is normal for TCP sessions to appear and disappear, or for the bandwidth of an OC-192 channel to be reconfigured between a variable number of tributaries of different lower bandwidth.

Therefore, it should be possible for the layered protocol VIP structure to adapt to dynamic changes in the modeled network, and be able to add and remove additional protocol layers and sibling protocol streams.

These normal protocol network operations must be verified. In keeping with the constrained-random philosophy of UVM, it would be desirable to be able to randomly modify the protocol structures at random times. Using layering, these varied protocol structures can be implemented in a reusable and scalable way. Unfortunately, due to the static nature of UVM components, the entire protocol hierarchy cannot be dynamically modified at run-time. All alternative protocol hierarchies must be created entirely at build time.

To support these verification scenarios, a future UVM should allow the dynamic modification of the component hierarchy.

References

[1] Open Systems Interconnection (OSI) model, ISO/IEC 7498-1.

[2] Accellera, “Universal Verification Methodology (UVM) 1.1 User’s Guide, May 2011

[3] Mentor Graphics, “Layering in UVM”, Verification Horizon, Vol 7, no 3, pp 25-27