A Simplified Approach Using UVM Sequence Items for Layering Protocol Verification

Haiqian Yu
Microsoft Corporation
haiqiany@microsoft.com

Christine Thomson
Microsoft Corporation
chthomso@microsoft.com

***Abstract* – Layered protocol architecture can break down complicated protocols into simpler tasks and therefore is commonly used in different areas. Verifying layered protocols is essential to guarantee the protocol works correctly at different layers when implemented in hardware. There are multiple approaches to implement layered protocol verification using Universal Verification Methodology (UVM). This paper presents a solution for layered protocol verification using layered sequence items to simplify and expedite implementation. This approach encapsulates the protocol's details within sequence items to eliminate the overhead and complexity associated with layered agents or layered sequencers. It also provides a standard API for protocol conversion between layers to simplify use within the testbench.**

I. Introduction

Layered protocols are quite common for both block level and chip level design. Many standard packetized protocols use layering to translate a packet level transactions down to pin level signaling. Some examples of layered packetized protocols include PCIe, Serial Rapid IO and MIPI. These protocols all have an upper level logical layer of packetized data which is passed to the lower physical layer to be translated to protocol specific signaling for transmission on the link. For intellectual property (IP) design, standard interfaces are commonly used for block level communication which can simplify the overall architecture and improve reuse of design components. Layering is often used to translate the high level unique data into the standard format expected on the interface for transmission to other blocks.

This same concept can be applied in Universal Verification Methodology (UVM) for layered protocol verification. For common interface protocols, the testbench should be constructed such that only one agent is required to be developed and reused on the shared interfaces. To model the upper level layer, translation from the base level protocol is needed within the testbench. There are multiple approaches for modeling the upper layer within the testbench. Each should be looked at when defining the testbench architecture to choose the approach that best meets verification needs. This paper will discuss a simplified approach for layered protocol verification in UVM which uses sequence items. The UVM sequence items are used to model each layer to maximize component reuse and reduce development time when compared against other approaches.

This approach was successfully used to verify a real-world design containing multiple instances of a two-layer protocol. All instances shared a common interface protocol for the lower layer and unique behavior at the upper layer. A single parameterized agent was used for the shared interfaces and layering was accomplished using sequence items for translation from the base protocol to the high level protocol. Encompassing layer translation within sequence items simplifies code throughout the testbench which can be complex for layered agents and layered sequencer implementations. This design has been simplified and used as the example scenario in this paper to demonstrate this simplified approach for layered protocol verification.

II. Background

A simple design using a layered protocol will be used as an example verification scenario to describe this approach. A block diagram of the example design is shown in Figure 1 below. The lower level layer is common between two IPs and use a Valid/Ready handshaking protocol to pass data from the source block to the destination block. The lowest layer which drives the interface will be referred to throughout this paper as the Valid/Ready layer and is labeled Figure 1. The Valid/Ready layer does not care about the data contents and only operates at the protocol level. The high level layer operates on meaningful data fields and does not know the details of the communication protocol used on the interface. This will be called the Packet layer as labeled in Figure 1. The Packet layer is responsible for translating the raw interface data into meaningful fields to process at a high level. This packetized data could be forwarded to additional high level layers for further translation or processing, but for simplicity a two-layer example will be used.

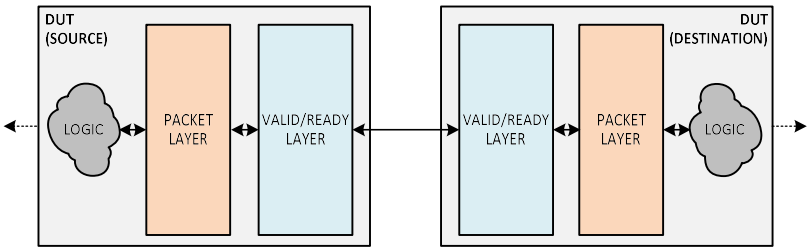

Figure 1: Layered Protocol Example

An overview of the Valid/Ready protocol is provided to better understand this example. With Valid/Ready protocol, the transmission originator (source) will present the raw data on the interface with a valid signal asserted to indicate to the transmission receiver (destination) that valid raw data is available for reception. The source is required to hold the data stable once valid has been asserted until a ready response is returned by the destination indicating the data has been accepted. This protocol allows for backpressure in the pipeline as the ready signal can be asserted at any time. When the receiver is ready to accept new data, ready is asserted to the source. When valid and ready are both asserted, the raw data is accepted and the handshake is complete. In this example, the data on the interface is referred to as raw data because the Packet layer's meaningful data fields are packed into the raw data as a flat array. A timing diagram of the Valid/Ready protocol is shown in Figure 2 displaying multiple raw data transfers.

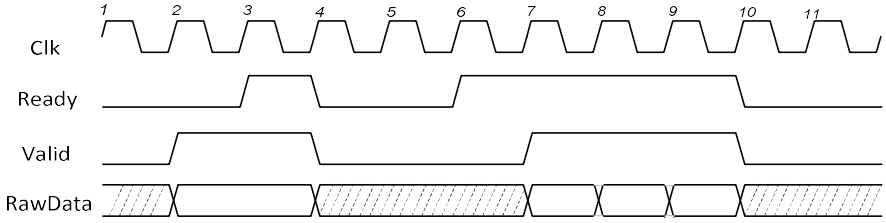

Figure 2: Valid/Ready Handshake Example Timing Diagram

Some examples of similar handshake protocols include AXI4-Streaming and Local Link.

The Packet layer may contain many fields. To demonstrate Packet sequence item construction, three arbitrary fields will be used: packet ID (an 8-bit value), ADDR (a 16-bit value) and DATA (a 32-bit value). Packet data is translated into Valid/Ready raw data by flattening each field into a larger 56-bit data array. The raw data used by the Valid/Ready layer is shown in Figure 3 below.

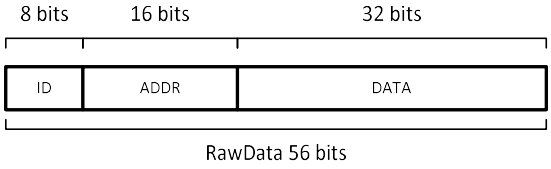

Figure 3: Packet Fields Packed to Form Raw Data

Translation in the reverse direction occurs by extracting each field from their offset location in the raw data bus and assigning them to their corresponding data field in the Packet sequence item.

III. Related Work

Multiple approaches have been discussed for layered protocol verification. Chauhan et al.[1] implemented a reusable and scalable architecture which layers and de-layers agents that correspond to each layer in the design. The drivers of the high level layers are used to convert the high level protocol data to the low level protocol data. This approach has the flexibility of verifying individual layers with peer-to-peer block level verification. In addition, it is easy to inject errors at different layers for maximum controllability. This solution can be time consuming and add complexity to the testbench if that level of flexibility is not required and the focus is solely on overall behavior. When analyzed for use with the example described in Section II, Background, this approach would not be a good fit.

More time would be spent on agent development and the extensive visibility and controllability is not required to verify the design at a high level.

Fitzpatric et al.[2] and Doulus[3] propose a similar approach however instead of developing a full agent corresponding to each layer, the drivers are removed from the high level layers. This trimmed down agent layers the sequencers which perform translation from the high level protocol to the low level protocol. The monitor performs the translation in the reverse direction reconstructing the low level protocol into the high level protocol before transmitting the transaction for use in the testbench. This approach would fit well for the described example scenario, however if there are multiple unique upper layers which share a common lower layer it can be time consuming to develop these custom trimmed down agents for each upper layer. A block diagram describing the testbench architecture for this methods is shown in Figure 4 below. Both implementations are proven to be flexible and used by different layered protocols. Figure 4 only shows the scoreboard connection to the Packet Layer, but connectivity to the Valid/Ready monitor would be added to match the first approach for verifying each layer.

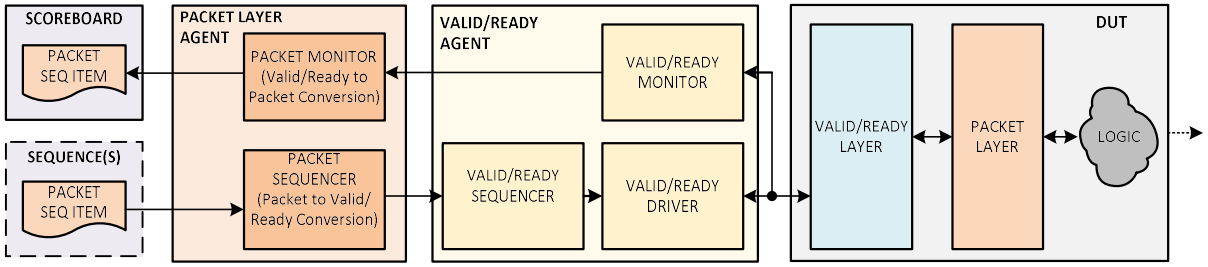

Figure 4: Proposed Method by Previous Work

By using layered components, transactions between different layers can be verified independently at each layer.

This paper will describe an alternative approach for layered protocol verification using UVM sequence items for layering and de-layering. Instead of creating new environment components for translation, the protocol specific conversion is refactored into the UVM sequence items. This approach simplifies the testbench architecture and can improve initial testbench bring up time by eliminating the need for developing new components.

IV. Proposed Method

Figure 5 shows an overview of our proposed method for layered protocol verification using the example from Section II. The Valid/Ready protocol is used for the low level layer and the Packet layer described in Figure 3 is used for the high level layer.

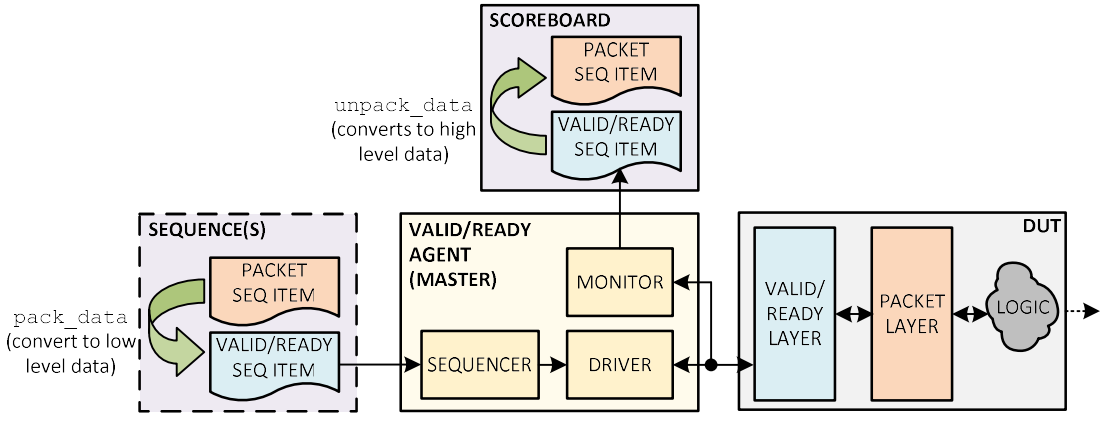

Figure 5: Proposed Method Using Layered Sequence Items

Translation from the Packet protocol to the Valid/Ready protocol occurs within the sequence by calling the translation function (pack_data) built into the Packet sequence item. In the reverse direction, translation occurs from the

Valid/Ready protocol to the Packet protocol by calling the unpack_data translation function on the Packet sequence item. In the figure, unpack_data is called in the scoreboard but would be needed for all classes connected to the Valid/Ready monitor. With translation occurring in the Packet sequence item, the Valid/Ready agent is instantiated directly without adding complexity to the environment for layers or translation classes. In this example, the generic Valid/Ready agent contains a parameterized bus width and will be instantiated with 56-bits to accommodate the size of the example data transfer.

One sequence item is needed to model each layer in the design. In this example, two sequence items are needed. The first corresponds to the base protocol level agent, in this case the Valid/Ready agent. The second corresponds to the Packet layer containing meaningful data fields and conversion functions. The high level sequence item must extend the base sequence item to inherit base level fields and functions. In this example, the Packet sequence item extends the Valid/Ready sequence item, adds the meaningful data fields, and implements the conversion functions: pack_data and unpack_data.

```
class vld_rdy_seq_item #(int DATA_WIDTH=`DATA_WIDTH_DEFAULT)
                                  extends uvm_sequence_item;

  rand bit                  valid;
  rand bit                  ready;

  rand bit [DATA_WIDTH-1:0] raw_data;

  ...

  // Packet -> Valid-Ready conversion
  // Virtual function to implement translation from
  // the Packet sequence item's meaningful fields
  // to vld/rdy raw data format
  virtual function void pack_data();
    // Placeholder function
  endfunction

  // Valid-Ready -> Packet conversion
  // Virtual function to implement translation from
  // the vld/rdy sequence item's raw data to the
  // extended sequence item's meaningful fields
  virtual function void unpack_data();
    // Placeholder function
  endfunction

endclass: vld_rdy_seq_item
```

Figure 6: Valid/Ready Sequence Item

```
class pkt_seq_item extends vld_rdy_seq_item #(56);

  rand bit [7:0]  id;
  rand bit [15:0] addr;
  rand bit [55:0] data;

  ...

  // Implementation for packing meaningful data
  // into raw data bus of base sequence item
  virtual function void pack_data();
    raw_data = {id, addr, data};
  endfunction

  // Implementation for unpacking raw data
  // into meaningful field for use
  virtual function void unpack_data();
    {id, addr, data} = raw_data;
  endfunction

  // Always convert data into a the base type
  // following randomization to ensure data
  // is always converted before it is sent to
  // the base type agent
  function void post_randomize();
    ...
    pack_data();
  endfunction
endclass: pkt_seq_item
```

Figure 7: Packet Sequence Item

Figure 6 and Figure 7 show the pseudo code for both the base sequence item (vld_rdy_seq_item) and the derived sequence item (pkt_seq_item). In the base sequence item, two virtual functions are defined: pack_data and unpack_data. These two functions must be implemented in the derived sequence item. Pack_data defines how the Packet data fields are translated into the raw data. Unpack_data defines how the raw data from the Valid/Ready protocol is converted into Packet data.These two functions are critical as they perform the conversion of data between the different layers of protocol.

Once the Valid/Ready and Packet sequence items are defined, sequences will generate data using the Packet sequence items as usual then convert this transaction into a Valid/Ready sequence item before it can be started on the agent's sequencer for transmission by the driver. The flow of the sequence body would be as follows:

1. Instantiate and create a local Packet sequence item in the sequence. This is used to generate the appropriate traffic at a high level when randomize is called on the Packet sequence item.
2. Within the sequence body pack_data is called on the Packet sequence item to convert the high level data into the low level data format, the raw data bus in this case. Alternatively, if randomize is used to generate the traffic scenario, pack_data can be placed within the sequence item at the end of the post_randomize function as is done in the pkt_seq_item pseudo code.
3. This will ensure that data is always converted after randomize and this extra step can be removed for random sequences. For directed scenarios where randomize is not called, this step is required.
4. Cast the Packet sequence item to a Valid/Ready sequence item so it is the correct type to be accepted by the sequencer.
5. Resume the standard UVM sequence body flow using the Valid/Ready sequence item to send traffic to the driver for transmission on the interface.

An example random Packet sequence is shown in Figure 8 demonstrating how to implement a sequence using these steps.

```
class pkt_rand_seq extends uvm_sequence #(vld_rdy_seq_item #(56));

  pkt_seq_item           pkt_item
  vld_rdy_seq_item #(56) req_item;
  vld_rdy_seq_item #(56) resp_item;
  ...

  task body();

    // 1. Instantiate and create a local Packet sequence
    // item generate the random data
    pkt_item = pkt_seq_item::type_id::create("pkt_item");

    if (!( pkt_item.randomize() with { (valid == 1); } )) begin
      `uvm_error(REPORT_TAG, "pkt_item.randomize failed!")
    end

    // 2. Data conversion from Packet to Valid/Ready
    // (optional if already done in pkt seq item's
    //  post_ranomize function)
    pkt_item.pack_data();

    // 3. Cast the pkt_seq_item to a vld_rdy_seq_item
    if (!$cast(req_item, pkt_item.clone())) begin
      `uvm_error(REPORT_TAG, "Cast on pkt_item failed!");
    end

    // 4. Resume standard sequence flow
    start_item(req_item);

    finish_item(req_item);

    get_response(resp_item);
    ...

  endtask: body

endclass: pkt_rand_seq
```

Figure 8: Packet to Valid/Ready Sequence Item Conversion

With this flow, sequences are written in terms of the high level Packet sequence item and hide all the underlying details of the low level protocol from the test or sequence writer. This allows for anyone who is familiar with the high level protocol to step in and easily assist with writing testcases without needing any knowledge of the low level protocol.

As previously mentioned, any class receiving items from the low level monitor is responsible for translation in the reverse direction. This must be done before Packet fields can be accessed for use in the testbench. Construction of the Packet sequence item follows a similar format as the sequences. Once the base sequence item is received on the input port, the raw data of the base sequence item is assigned to a Packet sequence item's raw data bus. The unpack_data function is then called to construct the Packet sequence item's high level data. After unpack_data is called, the Packet sequence item is ready for use. In Figure 5, since the scoreboard is connected to the Valid/Ready agent's monitor, it would need to follow this flow for each transaction received from the monitor. Pseudo code for the scoreboard demonstrating the Valid/Ready to Packet conversion is shown in Figure 9 below.

```
class scoreboard extends uvm_scoreboard;

  ...

  task pkt_scbd::run_phase(uvm_phase phase);

    vld_rdy_seq_item #(56) vld_rdy_item;
    pkt_seq_item           pkt_item;
    ...
    fork
      ...

      forever begin
        // Monitor Valid/Ready for valid transaction's
        vld_rdy_fifo.get(vld_rdy_item);

        // Create new Packet sequence item
        pkt_item = pkt_seq_item::type_id::create("pkt_item");

        // Convert to a Valid/Ready sequence item to a
        // Packet sequence item to access meaningful fields
        pkt_item.raw_data = vld_rdy_item.raw_data;
        pkt_item.unpack_data();

        // Fields are now accessible

        `uvm_info(REPORT_TAG,
          $sformatf("New Packet Item. ID: %0d Addr: 0x%16h",
                    pkt_item.id, pkt_item.addr),
          UVM_MEDIUM);

        // Sample coverage for meaningful fields
        pkt_item.sample();

        ...
      end //end vld/rdy_fifo.get

      ...
    join

  endtask: run_phase
  ...
endclass: scoreboard
```

Figure 9: Valid/Ready to Packet Sequence Item Conversion

In practice, there may be many interfaces with a shared interface protocols but different fields defined for the Packet layer. Alternatively, a common interface protocol could be used across projects but the Packet layer definition changes or evolves across projects. The proposed method proves increasingly beneficial for these applications and effectively manages verification time since it does not require new layered components to be developed for each unique instance.

V. Future Work

This paper used a two-layer protocol to demonstrate this simplified approach for layered protocol verification. This approach is easily scalable for multi-level layered protocols. As previous mentioned in Section IV, Proposed Method, a sequence item per layer would be created to model each layer of the protocol and corresponding translation functions.

The same example scenario will be expanded to demonstrate scaling to a three-layer protocol adding a new Large Packet layer on top of the Packet layer. Figure 10 below is an updated block diagram for this three-layer scenario. Each layer's sequence item must be derived from the direct lower layer. In this case, the Large Packet sequence item would extend Packet sequence item. The translation function for the Large Packet to Valid/Ready conversion (pack_data) would implement the translation from the Large Packet protocol to the Packet Protocol. super.pack_data is called at the end of the function to convert the Packet protocol to the Valid/Ready protocol and thus completing the full translation from Large Packet to Valid/Ready. Calling super.pack_data at the end of the function guarantees that the translation occurs in the correct order. The unpack_data implementation would be slightly different and require super.unpack_data to be called at the beginning of the function. This ensures the Valid/Ready protocol is first translated to the Packet level before Packet is translated to a Large Packet. Large Packet's unpack_data then only implements the translation from Packet data to Large Packet data. Each additional Layer added would follow this same structure for conversion between layers.

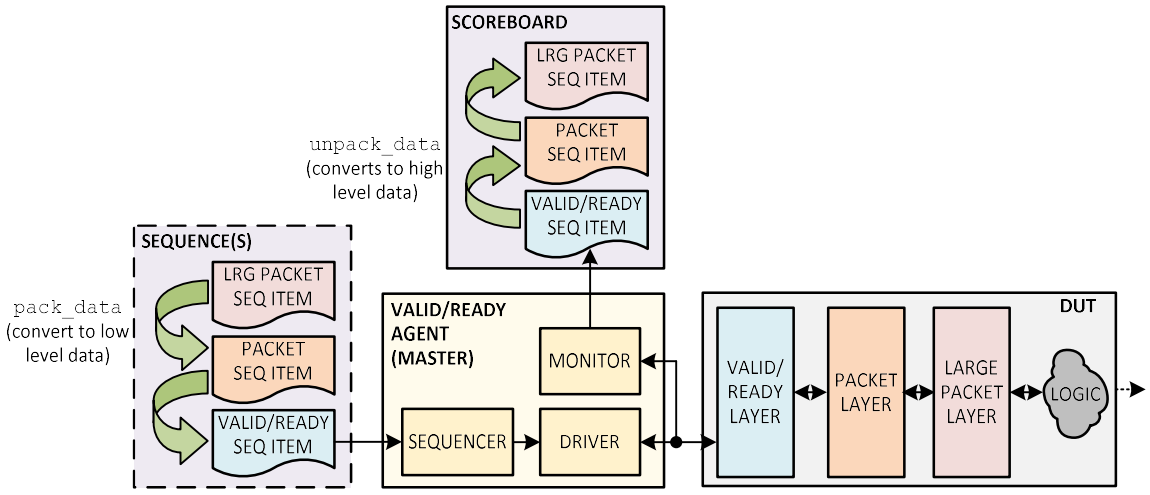

Figure 10: Extending for Multiple Layers

In the figure, each sequence item type is show to demonstrate that each sequence item type exists within the Large Packet sequence item through inheritance. The green arrows show that pack_data and unpack_data will be called recursively. Note that, it is not required to explicitly create each sequence item type and manually call translation functions for each layer. This will be done behind the scenes and still add only the minimal code for translation within testbench classes. When using this method for multi-layer verification, primary visibility and controllability would be at the highest and lowest level layers.

Another area for future work would be to experiment with this concept for per layer verification. This is conceptually feasible but has not been proven by use in a real-world design. Conceptually, each layer exists within the high level sequence item because of the sequence item's inheritance and recursive implementation in the translation functions. Therefore, any layer's fields can be accessible from the high level sequence item. For example, to check the middle Packet layer shown in Figure 10, instead of instantiating the high level Large Packet sequence item to unpack Valid/Ready data, a Packet sequence item could just as easily be instantiated if only the Packet level contents was needed. Another option would be always instantiating the Large Packet sequence item and unpack the Valid/Ready data into the Large Packet which will recursively unpack all layers behind the scenes. When defining each sequence item, a compare_<layer> function can be added which will compare the layer specific fields. Then to check only the Packet level fields, the compare_packet function can be called on the Large Packet sequence item which will compare only the Packet level fields that exist in the Large Packet. More work is needed in this area to determine the feasibility of this approach for per layer verification and any pitfalls or limitations.

VI. Conclusions

Using the translation flow described in Section IV, Proposed Method, only new sequence items need to be created for each layer and minimal code is added to existing classes. This eliminates the need for wrapper UVCs, translation classes, or complex layered sequencers. Reducing the number of environment components reduces initial development time and can expedite testbench bring up. Development time savings scales for each unique Packet layer in the design. Comparing this approach with that shown in Figure 4, a design with two unique packet layers would require two sequence items, two conversion monitors, two conversion sequencers, and two wrapper class to be developed and added to the testbench environment. This simplified approach only requires two sequence items.

This approach is intended to simplify layered protocol verification by abstracting away the underlying protocol details between layers from the testbench. It was proven beneficial when implemented in a real-world design which focused on high level functionality and not individual layer verification. It is important to ensure the right approach is selected for layered protocol verification to meet verification needs.

Section V Future Work, describes how this concept can be expanded upon for layered protocols with more than two layers. Based on previous experience and conceptual analysis, this simplified approach appears promising for verifying individual layers, but could require some code modification from what has been described in the example. More work is needed to determine what modifications would be needed since it has not been conceptually proven to verify a real-world design.

VII. References

[1] Rahul Chauhan, Grupreet Kaire, Ravindra Ganti, Subhranil Deb, “Layering Protocol verification: A Pragmatic Approach Using UVM”, SNUG 2014

[2] Doulous, “Requests, Responses, Layered Protocols and Layered Agents”, Online resources at http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/layering

[3] Tom Fitzpatric, “Layering in UVM”, Verification Horizons