Effects of Abstraction in Stimulus Generation of Layered Protocols within OVM

Josh Rensch
Lockheed Martin
3333 Pilot Knob Rd
Eagan, MN 55121 USA
(00)651.456.7121
joshua.w.rensch@lmco.com

Jesse Prusi
Lockheed Martin
3333 Pilot Knob Rd
Eagan, MN 55121 USA
(00)651.456.4289
jesse.r.prusi@lmco.com

ABSTRACT

A multi-layer protocol is a lower-layer protocol wrapped in a higher-layer protocol, for example IP over Ethernet. Multi-layer protocols are challenging because of the linkage between the layers required for stimulus generation for a design under test (DUT) that is aware of and processes both layers simultaneously. This paper will discuss the challenges of verifying a design that supports multi-layer protocols and the use of Open Verification Methodology (OVM) transaction objects to overcome them, particularly in the creation of stimuli.

Stimulus generation may occur at different layers of the protocol. For example, error injection can be handled at each layer of the protocol. An OVM transaction containing the configuration information is created at the highest layer of the protocol and the highest level of abstraction. At a lower layer of the protocol, there is a sequence that converts higher-layer protocol objects to lower-layer protocol objects for the DUT-level driver. At this layer, the configuration information from the higher-layer object is used to drive both valid data and errors into the DUT.

By detailing our approach, we demonstrate how to use OVM sequences at the proper abstraction levels to generate stimulus at multiple protocol layers to meet coverage requirements.

General Terms

Languages, Verification.

Keywords

SystemVerilog, Constrained Random, OVM, Layered Protocol.

1. INTRODUCTION

It has become a standard practice in system on a chip (SoC) development to layer protocols in order to balance use of industry standards with the customization that is required for complicated designs. SoCs frequently have protocol-aware hardware at multiple protocol layers. This can lead to some challenging verification problems, especially in the area of stimulus generation.

Before becoming aware of Open Verification Methodology (OVM), we created simulation environments at a very low level of abstraction. Because of the similarities to traditional design work, this approach made it easier for team members who were transitioning from hardware design to verification. This approach was time intensive, requiring the creation of high-concept scenarios and extensive maintenance of the simulation environment.

Among the questions we needed to ask when we made the switch: What is the highest level of abstraction for the environment that still allows us to create the entire stimulus spectrum needed? How do we maintain visibility at every level of abstraction? At what level does the environment compare the outputs of the DUT with those of the reference model?

2. DEFINITION OF ABSTRACTION

A decision on abstraction level necessarily comes before time spent designing any portion of a verification environment. A component can range in abstraction level from Register-Transfer-Level (RTL) to a highly abstract model. At more detailed levels, timing accuracy is important and all DUT signals must be modeled. For example, take a simple multiplier shown from two different perspectives in Figure 1. On the left you have the simple RTL view of the multiplier and on the right is a highly abstract view of the same component. (Meyer, 2004)

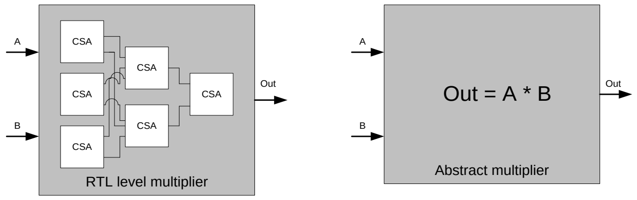


Figure 1 - Abstraction of a Simple Multiplier

As the model's abstraction level increases, timing and/or signal accuracy is lost, however the verification component will still be able to thoroughly model a DUT's functionality. Signals and timing may be driven using a range of valid values specified by the definition of the protocol.

Abstract models have several advantages. They are usually easier to write and maintain. And given that it's often easier to generate stimulus for them, abstract models also tend to encourage less white-box testing.

Stimulus generation depends on the abstraction level of a given model's components. A less abstract model may require more information to drive signals and timing than one at a higher level of abstraction.

An abstract model generally can cover unspecified behaviors by using constrained random techniques to generate stimulus that spans the full range of the legal protocol.

On the downside, abstract models may generate traffic that is within the legal parameters of the protocol, but outside the range of possible operations of the DUT. Furthermore, to ensure complete stimulus generation, functional coverage techniques may be required to ensure sufficient coverage.

Due to these tradeoffs, it is critical to choose the proper level of abstraction when designing a verification environment. If the environment is too abstract, it might be difficult to have enough control of stimulus. If it's not abstract enough, a lot of time will be spent in creating stimuli that test functionality outside the space of possible operations. Therefore abstraction level should be chosen as high as possible to facilitate the creation of stimulus that meets the verification requirements.

3. DEFINITION OF LAYERED PROTOCOLS

Protocol layering is one way of modeling encapsulated protocols, where one protocol is embedded within another. Common examples of this include IP over Ethernet and MPLS. We have chosen to use as an example a simple message protocol made up of many packets, shown in Figure 2, which illustrates some of the issues we faced in our project while removing the complexity of a real protocol. We have chosen only two protocols for the example, but this concept can be extended to more complex protocols.

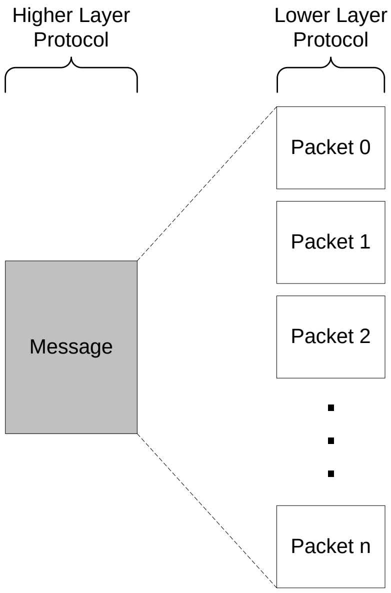


Figure 2 - Multi-Layered Packet to Message Format

Each packet in our example has routing and control information along with a data payload, as shown in Figure 3.

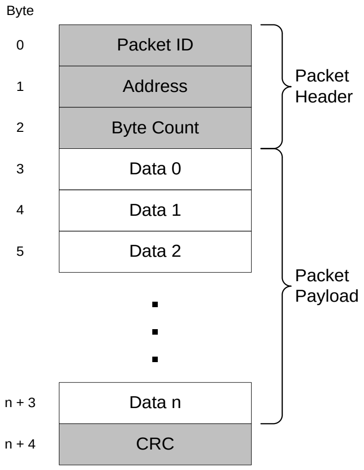


Figure 3 - Example Packet

The message resides in the payload of multiple packets as shown in Figure 4. The message has a unique ID that allows it to be reassembled in order by a receiver. Once all the packets for a given message have been received, it is then processed as a message.

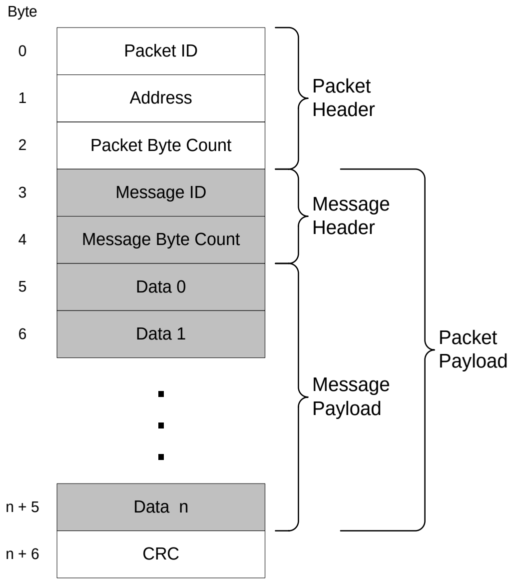


Figure 4 - Packet with Message Payload

When one protocol encapsulates another, it may be important to control, monitor, and check both protocols simultaneously. Protocol layering is one way of achieving this. As is common for protocol generation within OVM, each layer is created using a sequence, sequencer, and driver. In the case of Figure 4, the outer sequence generates a Message and sends it to a Message sequencer. The Message sequencer sends messages to a conversion sequence instead of a Message driver. From there it is sent to the Packet's sequence, sequencer, and driver setup. This continues until the Message protocol can be recognized and processed by the DUT.

4. PAST EXPERIENCE

We have been working with System Verilog (IEEE 1800) since before it became a major extension of the established IEEE 1364 Verilog language and the industry's first unified hardware description and verification language (HDVL) standard. Our first simulation environment was created in 2005 using a subset of

System Verilog since many features were still unsupported by commercial simulators. It had some deficiencies, primarily due to using a design-centric approach to a verification problem. Random stimulus was created without fully understanding abstraction levels and their effect on stimulus generation. The environment would create an object on the fly when requested by the DUT, but there was no interaction between the various protocol layers. This made it difficult to effectively control stimulus.

Another side-effect of using a design-centric approach was the tendency to try to stay in lockstep with the design for stimulus generation and results checking. This proved to be problematic for several reasons; signals change as the design evolves, many false errors are caused by timing differences between the DUT and verification models, and reuse is minimal due to the environment being tied to the design implementation.

In January of 2008, we started using Mentor's Advanced Verification Methodology (AVM), the precursor to OVM. We created stimulus using a custom "sequence-like" approach. A transaction was created that would be passed around to the various configuration components. These configuration components would peel the information that they needed from the transaction before passing it on to another component. After the entire configuration was complete, the transaction would be given to the component that actually created transactions to be sent to the DUT. This made any modification of the flow difficult due to the nature of the object; any change in RTL or verification would force us to modify a number of different objects. Another drawback in this type of fixed environment is that configuration and stimulus generation always occur in the same order. Various random delays were put into the configuration components to help alleviate this, but the environment only passed things down as needed and so they were of limited value.

By the start of our latest project, early in 2009, OVM 2.0 was released. Building on what we understood from our previous projects and wanting to improve on our previous attempts, we decided to take full advantage of the features in OVM. For example, using sequences for each layer of the protocol, only passing configuration information downward, and using factories to instantiate error sequences for certain levels of the protocol without affecting other levels.

5. LAYERED PROTOCOL STIMULUS GENERATION EXAMPLE

In our simplified example we demonstrate where we used the abstraction. For brevity, code examples shown may exclude code that is not necessary to explain the topic (e.g. new () functions).

Figure 5 shows all the sequences that are to be used to create stimulus for the example shown in Section 3. The Message and Packet Sequences are the basic building blocks for this architecture. To create a standard message as shown in Section 3, the Message Sequence component generates the message.

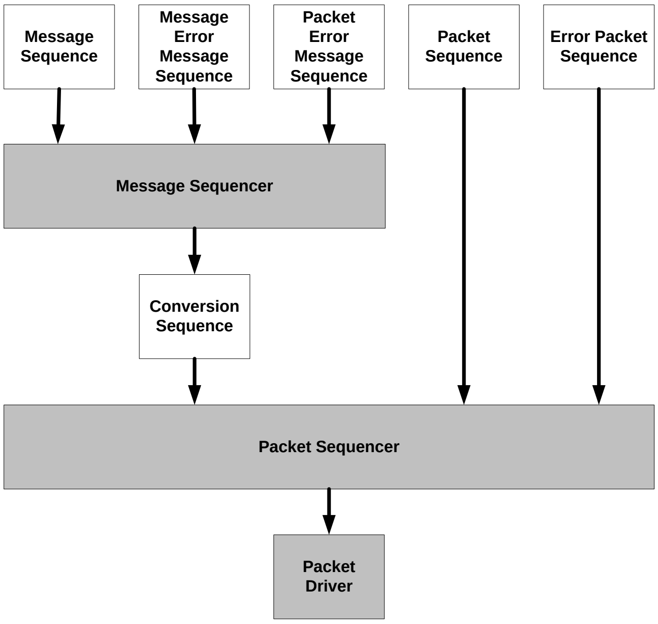


Figure 5 - Layered Stimulus Hierarchy

There are two other sequences that generate Message transactions, Message Error Sequence and Packet Error Message Sequence. These are two derived classes from the base Message Sequence and are just to show the various types of the other sequences that could drive the Message Sequencer. The Message Sequencer arbitrates between the Message Sequences to determine which message to forward to the Conversion Sequence.

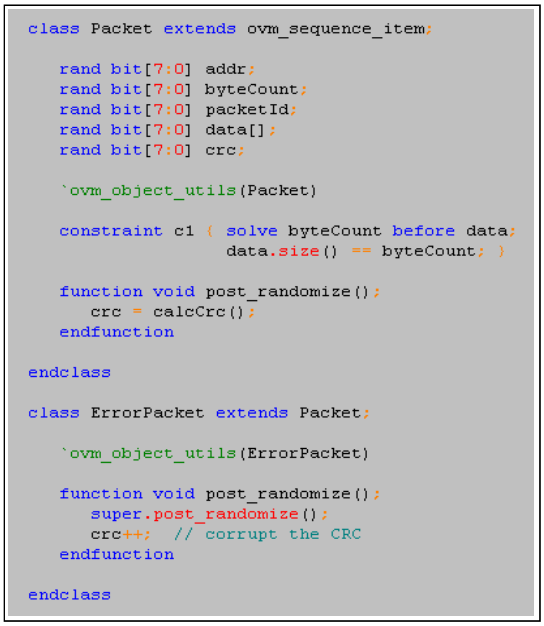
The Conversion Sequence creates packets from the message that the Message Sequencer passes on for service. One or more packets will be generated in the format shown in Figure 4.

These packets are then forwarded to the Packet Sequencer. Again like the Message Sequencer, this object chooses which sequence at a packet level to service and so forth until the transaction is at the lowest level of abstraction nearest the actual DUT interface.

To briefly touch on self-checking with this approach, a monitor is used to observe transactions. An analysis port is provided by the monitor and is typically connected to a scoreboard. Monitors and scoreboards are implemented at each layer of the protocol that requires checking according to the verification requirements.

With this hierarchy, it is simple to generate errors at any layer. Simply add another sequence that generates the error condition that is desired and hook it to the corresponding sequencer.

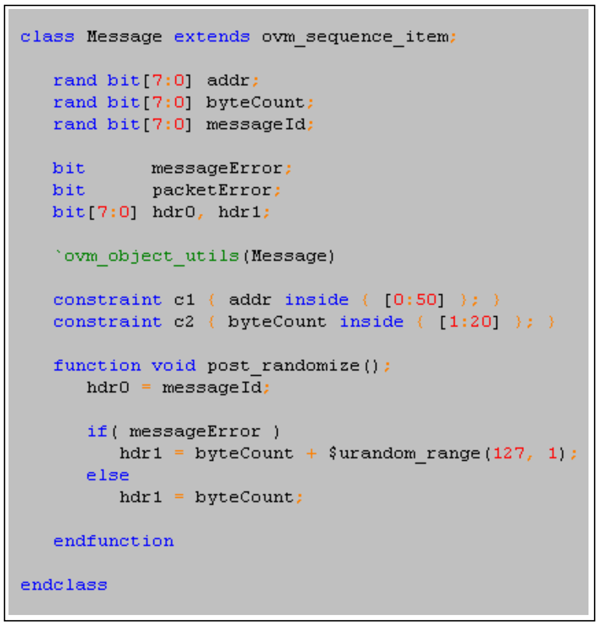
The code in Figure 6 shows the implementation of a packet from Figure 3. It extends ovm_sequence_item so we can utilize the sequence methodology provided by OVM. An ErrorPacket is also shown in Figure 6. A simple ErrorPacket will corrupt the CRC field to create packet errors.



```
class Packet extends ovm_sequence_item;

   rand bit[7:0] addr;
   rand bit[7:0] byteCount;
   rand bit[7:0] packetId;
   rand bit[7:0] data[];
   rand bit[7:0] crc;

   `ovm_object_utils(Packet)

   constraint c1 { solve byteCount before data;
                   data.size() == byteCount; }

   function void post_randomize();
      crc = calcCrc();
   endfunction

endclass

class ErrorPacket extends Packet;

   `ovm_object_utils(ErrorPacket)

   function void post_randomize();
      super.post_randomize();
      crc++;  // corrupt the CRC
   endfunction

endclass
```

Figure 6 - Example Packet

The implementation of a message is shown in Figure 7. Similar to Packet, it extends ovm_sequence_item. This class contains two control bits for creating errors. The messageError bit determines if an error will be generated at the message level, which is created by corrupting the message header. The packetError bit determines if an error will be generated at the packet level created by corrupting the CRC by utilizing the ErrorPacket class. Packet error injection will be described in more detail below.



```
class Message extends ovm_sequence_item;

   rand bit[7:0] addr;
   rand bit[7:0] byteCount;
   rand bit[7:0] messageId;

   bit       messageError;
   bit       packetError;
   bit[7:0]  hdr0, hdr1;

   `ovm_object_utils(Message)

   constraint c1 { addr inside { [0:50] }; }
   constraint c2 { byteCount inside { [1:20] }; }

   function void post_randomize();
      hdr0 = messageId;

      if( messageError )
         hdr1 = byteCount + $urandom_range(127, 1);
      else
         hdr1 = byteCount;

   endfunction

endclass
```

Figure 7 - Example Message

The heart of the solution for creating layered stimulus is the use of a conversion sequence as shown in Figure 5. The role of the conversion sequence is to take the higher-layer transactions (messages) and convert them to lower-layer transactions (packets). The conversion is done in a sequence so the packets on the output side are sent to a standard sequencer interface, which allows other packet traffic (not associated with messages) to be driven to the same interface. This approach allows for flexible stimulus generation at each protocol layer with the use of reusable components based on OVM.

The conversion sequence acts like an ovm_driver on the input side and pulls messages from a message sequencer by using an ovm_seq_item_pull_port. The caveat is that the conversion sequence cannot create or connect the ovm_seq_item_pull_port since sequences are not components. The solution is to create the port somewhere in the component hierarchy on behalf of the conversion sequence and assign it a handle in the conversion sequence. In our example, the ExampleTest component creates the msgConvPort. It connects the port to the message sequencer and assigns it to the msgPort handle in ConversionSequence during the connect phase. Please see Figure 8 & Figure 9.

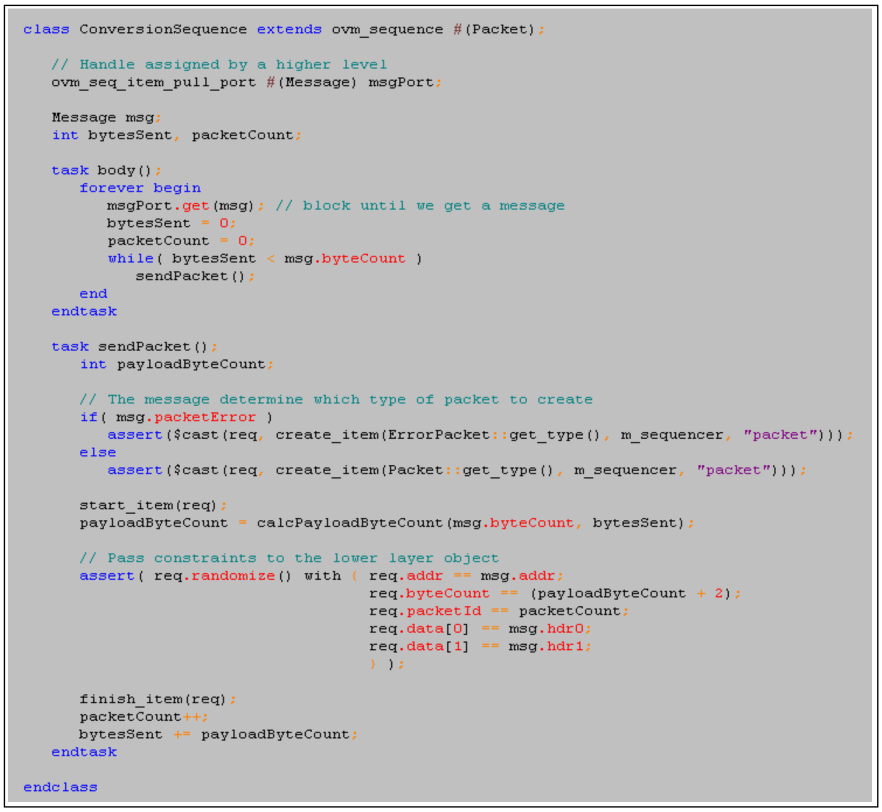
The ConversionSequence can be thought of as a “static” sequence that is started at the beginning of simulation and runs forever. It continuously monitors the msgPort for an incoming message from the message sequencer. Once it receives a message, it creates one or more packets as specified by the message byte count. Each packet is randomized with additional constraints required from the message. In our example, the Message does not impose any constraints on the data payload. Therefore the Message object has no data payload. It is created on the fly during randomization of each Packet.

This implementation does not allow for the absolute latest generation of Messages – once a message is pulled into the ConversionSequence, some time may elapse before the packet request is granted by the packet sequencer. However, the OVM sequence API provides the capability to access and use the latest generation of the higher-level object if needed.

The ConversionSequence uses the packetError flag from the Message to determine if a normal Packet or an ErrorPacket should be created. Additional types of packet errors could be easily inserted by creating a new class that extends ErrorPacket and creating a factory override.

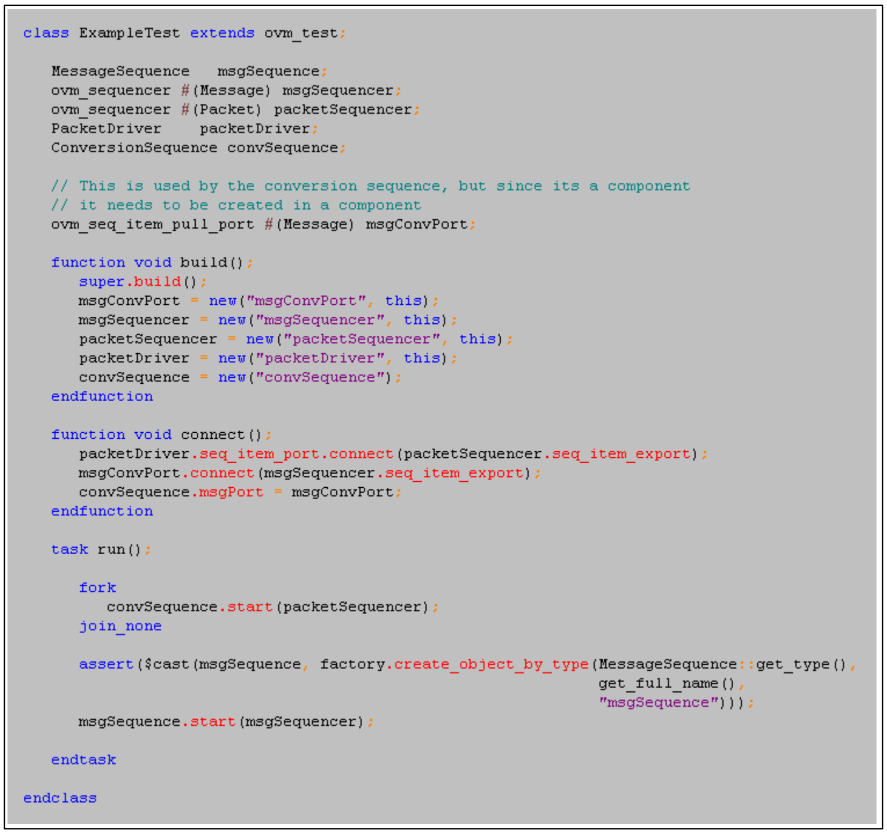
The high-level sequences used in our example are shown in Figure 10. The MessageSequence creates one message and sends it to the sequencer. The MessageErrorMessageSequence creates one message that has a message error. It uses the pre_do() callback to set the messageError flag in the request object (Message) before it is randomized. Similarly, the PacketErrorMessageSequence creates one message that has a packet error and sets the packetError flag in the pre_do() task. Factory overrides can be used to determine which sequence to use from the top level.

.



```
class ConversionSequence extends ovm_sequence #(Packet);

   // Handle assigned by a higher level
   ovm_seq_item_pull_port #(Message) msgPort;

   Message msg;
   int bytesSent, packetCount;

   task body();
      forever begin
         msgPort.get(msg); // block until we get a message
         bytesSent = 0;
         packetCount = 0;
         while( bytesSent < msg.byteCount )
            sendPacket();
      end
   endtask

   task sendPacket();
      int payloadByteCount;

      // The message determine which type of packet to create
      if( msg.packetError )
         assert($cast(req, create_item(ErrorPacket::get_type(), m_sequencer, "packet")));
      else
         assert($cast(req, create_item(Packet::get_type(), m_sequencer, "packet")));

      start_item(req);
      payloadByteCount = calcPayloadByteCount(msg.byteCount, bytesSent);

      // Pass constraints to the lower layer object
      assert( req.randomize() with { req.addr == msg.addr;
                                     req.byteCount == (payloadByteCount + 2);
                                     req.packetId == packetCount;
                                     req.data[0] == msg.hdr0;
                                     req.data[1] == msg.hdr1;
                                     } );

      finish_item(req);
      packetCount++;
      bytesSent += payloadByteCount;
   endtask

endclass
```

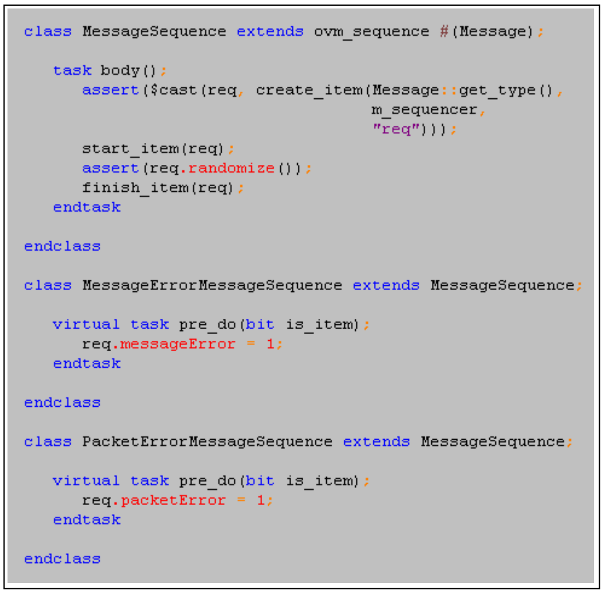
Figure 8 - Example Conversion Sequence



```
class ExampleTest extends ovm_test;

   MessageSequence   msgSequence;
   ovm_sequencer #(Message) msgSequencer;
   ovm_sequencer #(Packet) packetSequencer;
   PacketDriver    packetDriver;
   ConversionSequence convSequence;

   // This is used by the conversion sequence, but since its a component
   // it needs to be created in a component
   ovm_seq_item_pull_port #(Message) msgConvPort;

   function void build();
      super.build();
      msgConvPort = new("msgConvPort", this);
      msgSequencer = new("msgSequencer", this);
      packetSequencer = new("packetSequencer", this);
      packetDriver = new("packetDriver", this);
      convSequence = new("convSequence");
   endfunction

   function void connect();
      packetDriver.seq_item_port.connect(packetSequencer.seq_item_export);
      msgConvPort.connect(msgSequencer.seq_item_export);
      convSequence.msgPort = msgConvPort;
   endfunction

   task run();

      fork
         convSequence.start(packetSequencer);
      join_none

      assert($cast(msgSequence, factory.create_object_by_type(MessageSequence::get_type(),
                                                              get_full_name(),
                                                              "msgSequence")));
      msgSequence.start(msgSequencer);

   endtask

endclass
```

Figure 9 - Example Test

These objects allow us to generate various messages and packets simply and quickly. We have control over the creation of large amounts of different stimuli and the distribution of the errors for these structures. We also have the ability to create background packets that have nothing to do with messages to better test the DUT. By controlling the constrained random stimuli, we can reach our coverage requirements.



```
class MessageSequence extends ovm_sequence #(Message);

   task body();
      assert($cast(req, create_item(Message::get_type(),
                                    m_sequencer,
                                    "req")));
      start_item(req);
      assert(req.randomize());
      finish_item(req);
   endtask

endclass

class MessageErrorMessageSequence extends MessageSequence;

   virtual task pre_do(bit is_item);
      req.messageError = 1;
   endtask

endclass

class PacketErrorMessageSequence extends MessageSequence;

   virtual task pre_do(bit is_item);
      req.packetError = 1;
   endtask

endclass
```

Figure 10 - Message Sequences

6. RESULTS

Going to a higher abstraction level allowed us to trade grey-box testing for more black-box testing which helped reduce the number of bugs introduced in the simulation environment due to timing. The creation of more complicated stimulus with less code was another benefit of going to a higher abstraction level.

We use a metric called SLOC, or significant lines of code. Our first project was approximately 98,000 verification SLOCs, while a similar project using AVM was 45,000 verification SLOCs and more thoroughly tested the DUT. We are on target with our latest project with OVM to come in below 45,000 SLOCs.

We compared the verification of two designs that have nearly identical multi-layered protocol interfaces. Even though the second project had additional verification requirements, we realized significant improvements in the verification effort by using the layered stimulus approach outlined in this paper. The number of engineering development hours was reduced by 30%, the SLOC count was reduced by 55%, and the number of bugs found in verification components was reduced by 73%. In addition to these numbers, our verification components are more reusable and allow us to stress the DUT much more thoroughly than our previous solution.

Another benefit lies in determining when verification is complete. For our first project, we used a brute force method. We developed over 200 test cases to verify every feature. Once all 200 test cases passed, we considered verification complete. At the time, we lacked a quantitative measure testing thoroughness. In subsequent projects we utilized code and functional coverage to better measure the thoroughness of our verification. On our latest completed design, we were able to quantitatively get over 90% code and functional coverage for the FPGA we were verifying.

7. CONCLUSION

Choosing the proper abstraction level is instrumental in effectively generating stimulus for a layered protocol. Part of that abstraction choice is what methodology to utilize in creating your simulation environment. A predefined methodology like OVM has many built-in features to make it easier to create environments involving different protocol layers.

8. ACKNOWLEDGMENTS

Thanks to Andy Meyers of Mentor Graphics, Steve Nelson of Lockheed Martin, and the many editors without whom this paper would not have been possible.

7. REFERENCES

[1] Glasser, Mark. Open Verification Methodology Cookbook. Springer, 2009

[2] Meyer, Andreas. Principles of Functional Verification. Newnes, 2004