# Lies My Teacher Told Me About The UVM: Basic Stimulus

Justin Refice

NVIDIA Corporation
Santa Clara, USA

www.nvidia.com

**ABSTRACT**

*Many users of the Universal Verification Methodology purchase books and other training materials which offer a simple description of how to setup your first testbench. These materials unfortunately tend to over-simplify, providing easy-to-read answers which scale poorly, provide minimal potential for reuse, and often omit important functionality.*

*This paper will demonstrate where the common view of stimulus breaks down and where the hazards in the simplified examples lie. Additionally a better approach will be presented leveraging familiar concepts from networking models, starting with a new perspective on the lowly sequence item.*

# Table of Contents



# Table of Figures



# Table of Tables

# 1. Introduction

The Universal Verification Methodology ("UVM") [1] provides basic structures for the creation of stimulus in the active domain of the testbench:

- *Sequence Items* are an abstract representation of a transaction on a bus.
- *Drivers* are responsible for processing/executing sequence items.
- *Sequences* are responsible for generating a stream of sequence items, and providing them to the driver.
- *Sequencers* are responsible for arbitrating which sequence gets to send the next item to the driver.

While there is general consensus on what each of these constructs is used for, *how* each of the constructs is used varies greatly from company to company, and even from team to team within a company, often in unfortunately incompatible ways. Given the amount of variance, one begins to question the definition of "Universal" in this context.

Illustrating a generalized form for using these structures is more complex than most introductory tutorials are willing to dive. This means that we end up with a large amount of training materials which present a simplified common example, and while this example is easy to understand at first glance, it begins to break down rather rapidly in the real world.

This paper presents a robust generalized method which leverages the encapsulation of the TCP/IP communication model to scale well, adapt easily, and provide both VIP developers and end users with a functional, consistent and familiar pattern for stimulus generation. In other words: A *Universal* pattern for stimulus generation.

# 2. That looks familiar...

The first step in generalizing our stimulus flow is to re-evaluate the interaction between the sequence and driver.

Sequences are "transient" in nature, meaning that a sequence can be created at any time during simulation, and can be destroyed just as easily. The driver on the other hand is "static" in nature, which is to say that once a driver has been constructed, it will continue to remain in existence for the entirety of simulation. The driver is constantly waiting for new requests to be initiated by sequences, at which point those requests will be acknowledged, serviced and potentially responses will be sent.

To those developers familiar with networking, this pattern should seem strikingly familiar, as it is the classic client-server model for communication. In this model, *clients* are responsible for initiating new sessions with the *server*, which constantly awaits new requests. A common real-world example of this model is the World Wide Web, with each web browser acting as a client, initiating new requests to web servers. In fact, most of the common protocols used on the internet (Web, File Transfer, Email, etc.) are client-server based.

By mapping UVM stimulus generation to the classic networking models, we can begin to build a foundation which will allow us to handle all forms of stimulus:

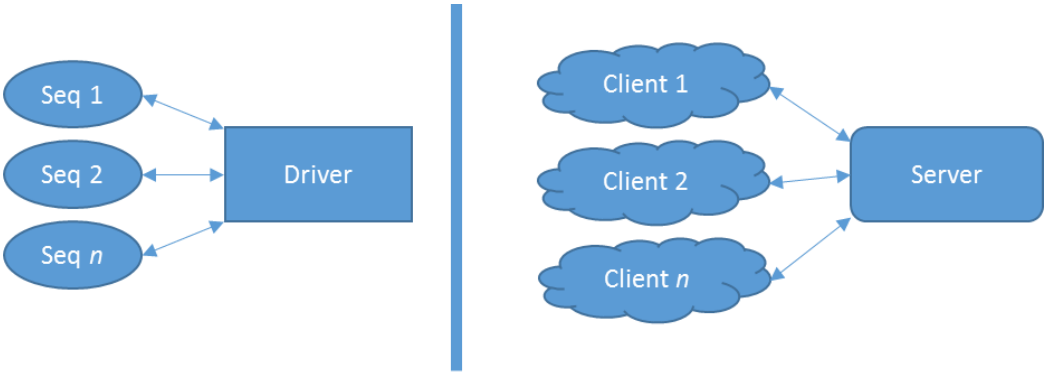


**Figure 1 – UVM Sequence/Driver vs. Client/Server Protocol**

It is fully intentional that the diagram omits the UVM Sequencer, this is because in the client/server view of sequences and drivers, it simply does not exist.

## 3. Layers of communication

A fundamental detail to remember when implementing any network communication (UVM Interface VIP included), is to separate "what" is being communicated from "how" it is being communicated.

Within computer networking, the internet protocol suite (generally referred to as "TCP/IP") is a common mechanism for describing the differences between these ideas. The model is constructed of 4 layers: Application, Transport, Internet and Link [2]. Each successive layer is capable of fully encapsulating the previous layer, providing clear abstraction of services. By clearly separating these layers, a typical user can browse the web without needing to know if their packets are being routed using Ethernet, Wi-Fi or carrier pigeon… that information is hidden from them by TCP/IP.

The following diagram illustrates the encapsulation and layering provided by the TCP/IP [3]:

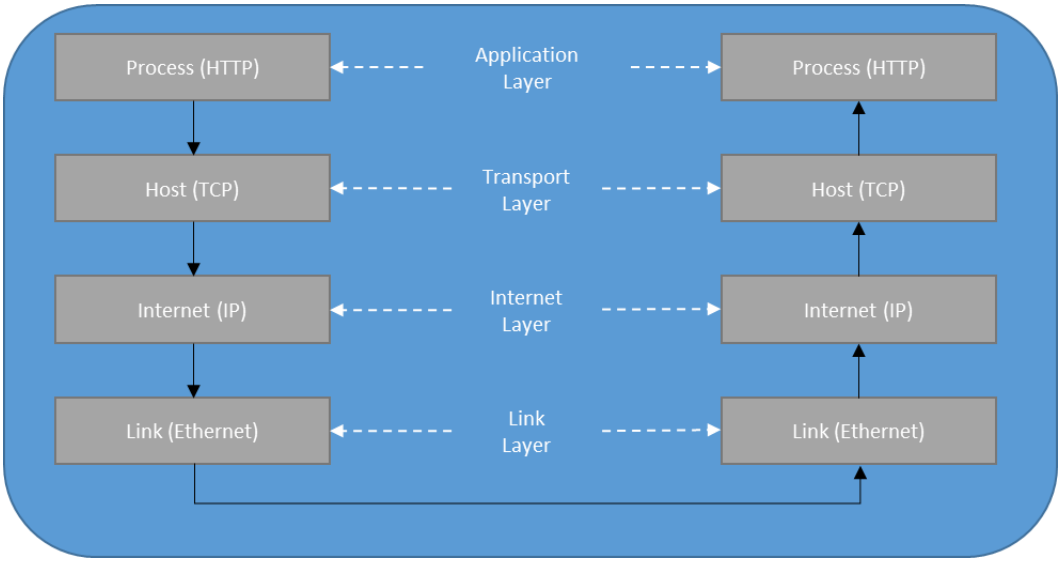


**Figure 2 – TCP/IP Data Flow**

In the TCP/IP stack the client and server processes communicate to one another using HTTP, however they wrap that information within a basic TCP packet which is routed over the network. The TCP protocol does not explicitly understand the HTTP protocol, it's just a mechanism for sending it. This is because TCP lies at the *transport* Layer, whereas HTTP lies within the *application* Layer.

While not as robust as the TCP/IP model, the same separation of concerns can be applied when handling the UVM sequence communication. Instead of having the 4 layers presented by TCP/IP, the UVM Sequence/Driver data flow contains only two: Application and Transport. The following diagram illustrates the flow of stimulus passing through the UVM sequencer:

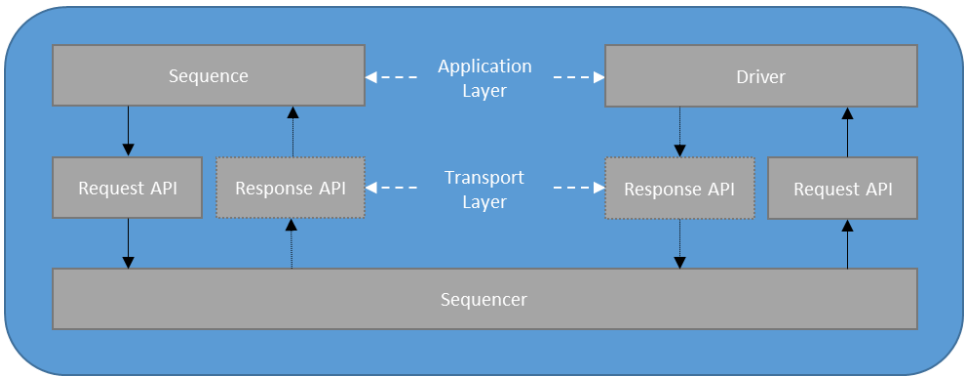


**Figure 3 – UVM Sequence / Driver Data Flow**

Just as the TCP layer is capable of sending HTTP (Web), SMTP (Email), and many other protocols, the sequencer *transport* layer can be used to send many different types of *application* layer requests and responses.

## 4. UVM Sequencer Transport Layer

There are two separate, albeit related, communications which are occurring at the "transport" layer of UVM stimulus: The sequence is interacting with the sequencer, and the driver is interacting with the sequencer. Both of these are implemented using direct method calls, and the entire protocol is strictly defined by the UVM. During this communication, the interactions can be separated into three basic stages:

- Alignment – During this stage, the sequence, sequencer and driver are "lining up" with one another. The sequence initiates a new request, and will be blocked if the driver is not ready to accept, or the sequencer determines that it is another sequence's turn, or if the sequence itself determines that the request is not immediately relevant. If the driver is expecting a new request, then it will be blocked until a sequence is available.

- Request Transmission – Upon entering this stage, the sequence, sequencer and driver are all in alignment, and the sequence *must* transmit the request to the driver without consuming any additional time. The zero-time requirement exists because it's possible that if time progresses, then the sequence or sequencer could drift out of alignment again. After sending the request, the sequence is blocked in this stage until the driver acknowledges the item. While time *can* be consumed between the driver receiving the item and acknowledging the item, the driver is only allowed to have one un-acknowledged request outstanding, so any sequences in the *alignment* stage will continue to be blocked during this time.

- Response Handling – Technically optional, this stage may be skipped depending on the nature of the request which was sent to the driver. If the request required additional information beyond the basic "acknowledgement" from the driver, then that information is provided in the form of additional responses. Any number of sequences can be in this stage simultaneously.

Notice how none of the stages explicitly mention the bus/interface which is being driven. This is intentional, as interacting with the actual interface is considered "higher level". The transport layer stages simply deal with passing requests and responses back and forth between the sequence and the driver.

The following diagram illustrates the three stages of the transport layer interactions, and the methods which are used during each:

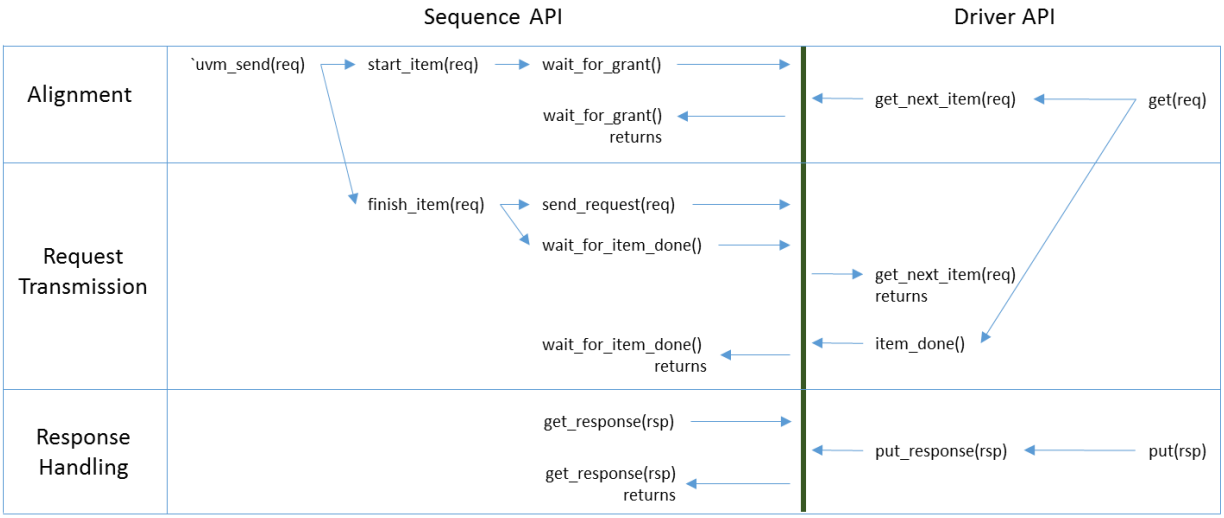


**Figure 4 – UVM Sequencer Transport Layer Protocol**

Note: `wait_for_grant()` and `get_next_item(req)` are not strictly ordered, either call could happen first.

## 5. UVM Sequence/Driver Application Layer

The interaction between the sequence and the driver is considered to be the *application* layer, as opposed to the *transport* layer provided by the sequencer, and the UVM does surprisingly little to describe the nature of this interaction. In fact, there are only a few requirements for the *application* layer:

1) Interactions between the sequence and the driver must be in the form of requests initiated by the sequence, bound for the driver.
2) The driver must acknowledge receipt of each request.
3) The driver is allowed to optionally send responses to those requests, bound for the originating sequence.
4) A single request may result in multiple responses.
5) Requests and responses must extend the UVM sequence item class.

Aside from those requirements, the UVM is silent on the exact nature of the sequence/driver interaction. This is not an oversight of the methodology, as each interface/bus/etc. has application-specific requirements which can't be generically expressed in a more specific manner than "requests" and "responses".

## 6. Confusing the Application and Transport Layers

While it's possible to map high level meanings to the low level interactions, developers do so at their peril, leading to an unnecessary amount of work for the test writers. Unfortunately, this is one of the most common mistakes which UVM VIP developers (and tutorials) make, confusing the low level interactions with the sequencer and the high level VIP developer intent (ie.

confusing the *transport* layer with the *application* layer). In order to illustrate this confusion, let's start with two reasonably simple interfaces: APB [4] and AHB-Lite [5].

The APB interface is non-pipelined, which is to say that the interface can only handle one request at a time. As such, a VIP developer may be inclined to produce the following item and example sequence:

```
class apb_item extends uvm_sequence_item;
  // 32 bit address
  rand logic [31:0] addr;

  // 1 indicates a WRITE
  // 0 indicates a READ
  rand logic        write;

  // Request data
  // Set by sequence in a write
  // Set by driver in a read
  rand logic [31:0] data;

  // Constructor, macros, etc...
endclass : apb_item

class apb_sequence extends uvm_sequence#(apb_item);
  // Constructor, macros, etc...

  virtual task body();
    apb_item tmp;

    // Read address 0x500
    `uvm_create(req)
    req.addr = 32'h0000_0500;
    req.write = 0; // READ
    `uvm_send(req)

    // Save off the response
    tmp = req

    // Write inverted data back
    `uvm_create(req)
    req.addr = 32'h0000_0500;
    req.write = 1; // WRITE
    req.data = !tmp.data;
    `uvm_send(req)
  endtask : body
endclass : apb_sequence
```

Unlike the APB interface, AHB-Lite is pipelined and supports multiple requests on the interface simultaneously. As such, our same VIP developer may be inclined to produce an alternative item and example sequence:

```
class ahb_item extends uvm_sequence_item;
  // 32 bit address
  rand logic [31:0] addr;

  // 1 indicates a WRITE
  // 0 indicates a READ
  rand logic        write;

  // Request data
  // Set by sequence in a write
  // Set by driver in a read
  rand logic [31:0] data[];

  // burst, lock, prot, size, etc...

  // Constructor, macros, etc...
endclass : ahb_item

class ahb_sequence extends uvm_sequence#(ahb_item);
  // Constructor, macros, etc...

  virtual task body();
    apb_item tmp;

    // Read address 0x500
    `uvm_create(req)
    req.addr = 32'h0000_0500;
    req.write = 0; // READ
    req.size = 2; // DWORD
    `uvm_send(req)
    get_response(rsp);

    // Write inverted data back
    `uvm_create(req)
    req.addr = 32'h0000_0500;
    req.write = 1; // WRITE
    req.size = 2;
    req.data[0] = !rsp.data[0];
    `uvm_send(req)
    get_response(rsp)

  endtask : body
endclass : ahb_sequence
```

Both examples would be perfectly functional, but there's one minor issue: Our VIP developer has forced the test writer to interact with the sequencer differently, depending on the nature of the interface. While it may make sense for the test to understand the pipelining characteristics of the interface, there's no reason why the low level interaction needs to be aware of these high level details.

If instead of relying on `item_done()` to indicate that the request was completed, the VIP developer had sent that information up the response path, then both the APB and the AHB-Lite VIPs would present an identical interaction model even though their high level interfaces support different capabilities. In other words, *what* we're telling the VIP to do can and should remain orthogonal to *how* we're telling the VIP to do it.

## 7. You talkin' to me?

In addition to the confusion caused by mixing *application* and *transport* layer communication, the VIP developer has an additional hurdle to jump in order to properly route response information back to the originating sequence.

When sending a response back to a sequence, the driver must provide routing information to the sequencer. This information is provided via the `set_id_info()` method:

```
// Copy the routing info from the request
rsp.set_id_info(req);
// Send the response
seq_item_port.put(rsp);
```

This "ID" allows the sequencer to simultaneously handle multiple outstanding requests. Each request has a unique identifier, and that identifier is all the sequencer needs to ensure that responses make their way back to the originating sequence. There's just one problem... what happens if the originating sequence had multiple requests outstanding?

For example:

```
virtual task body();
  REQ req1, req2;
  fork : multi_req
    `uvm_do(req1)
    `uvm_do(req2)
  join_none : multi_req
  get_response(rsp);
  // Is this rsp to req1 or req2!?
endtask : body
```

Fortunately, the UVM provides the ability to connect responses to their originating requests within a sequence via the `get_transaction_id()` method. If the sequence writer wants to

explicitly wait for a specific response, they can pass the transaction id to the `get_response()` method:

```
  get_response(rsp, req1.get_transaction_id());
```

Care needs to be taken though, as the sequence mechanism can deadlock if the sequence expects a different order than the driver provides:

```
virtual task body();
  REQ req1, req2;
  fork : multi_req
    `uvm_do(req1)
    `uvm_do(req2)
  join_none : multi_req
  get_response(rsp, req1.get_transaction_id);
  get_response(rsp, req2.get_transaction_id);
  // What happens if req2 is responded to first?
endtask : body
```

A safer mechanism is to check the transaction ids *after* receiving the responses... preventing a deadlock, and allowing for better debug information to be captured:

```
virtual task body();
  REQ req1, req2;
  fork : multi_req
    `uvm_do(req1)
    `uvm_do(req2)
  join_none : multi_req
  get_response(rsp);
  case (rsp.get_transaction_id())
    (req1.get_transaction_id()): //... req1
    (req2.get_transaction_id()): //... req2
    default: //... error!
  endcase
endtask : body
```

It's also worth noting that a sequence should not request a response to a request which hasn't been sent yet, as the act of sending the request is what triggers the transaction ID to be set.

## 8. Walking the walk

While the specifics of the application layer interface between the sequence and driver can't be generalized, there is one response type which is often desired at this level, and that is the indication that the driver has completed a request. Using this type of response, we can show a simple example of how to communicate between the sequence and driver in a consistent and scalable fashion.

For our example, the VIP supports the following high level session states:

- ACCEPTED – The driver has received the request, but not yet begun sending it on the bus
- BEGIN_REQ – The driver has started sending the request on the bus
- END_REQ – The driver has completed sending the request on the bus

As the request reaches each of these states, the driver will send a response back up to the originating sequence.

Remember that these states are provided as an example only... requests could have any number of states. While this grouping is a common subset, so common in fact that it even shows up in TLM 2.0 [6], the VIP developer is ultimately responsible for determining what states make sense to propagate back to the originating sequence. Since the driver is allowed to send any number of responses back for a given request, the state space is highly scalable.

Our sequences can use this high level state to wait for a request to complete:

```
virtual task body();
  REQ req1, req2;
  // Generate a random write
  `uvm_do_with(req1, {addr > 32'h0000_F000;
                      rw == WRITE;})
  // Wait for the request to complete
  do
    get_response(rsp);
  while (rsp.req_state != END_REQ);
endtask : body
```

Since the sequence is using the response path to retrieve high level information, the specifics of the driver implementation are now abstracted away from the low level interaction.

The driver could be implemented without a pipeline:

```
task my_driver::run_phase(uvm_phase phase);
  forever
    begin : get_and_drive
      seq_item_port.get(req);
      // Prepare the response
      rsp = RSP::type_id::create("rsp", this);
      rsp.set_id_info(req);
      // Mark as ACCEPTED while waiting for the
      // bus to be ready
      rsp.req_state = ACCEPTED;
      seq_item_port.put(rsp);
      // Wait for bus to be ready
      wait_for_ready();
```

```
      // Once the bus is ready, mark as BEGIN_REQ
      rsp.req_state = BEGIN_REQ;
      seq_item_port.put(rsp);
      drive_request(req);
      // Once the request is complete, mark as
      // END_REQ
      rsp.req_state = END_REQ;
      seq_item_port.put(rsp);
    end : get_and_drive
endtask : run_phase
```

Or it could be pipelined:

```
task my_driver::run_phase(uvm_phase phase);
  forever
    begin : get_and_drive
      // Wait for pipeline space to be available
      wait (num_requests < max_requests);
      // Retrieve the next request
      seq_item_port.get(req);
      num_requests++;
      // Prepare the response
      rsp = RSP::type_id::create("rsp", this);
      rsp.set_id_info(req);
      // Mark as ACCEPTED
      rsp.req_state = ACCEPTED;
      seq_item_port.put(rsp);
      // Fork off the processing
      fork : pipeline_fork
        // Save off the request/response
        automatic REQ forked_req = req;
        automatic RSP forked_rsp = rsp;
        // Handle the pipe
        begin : request_process
          // Wait for bus to be ready
          wait_for_ready(forked_req);
          // Once the bus is ready, mark as BEGIN_REQ
          forked_rsp.req_state = BEGIN_REQ;
          seq_item_port.put(forked_rsp);
          // Drive the request on the bus
          drive_request(forked_req);
          // Once the request is complete, mark as
          // END_REQ
          forked_rsp.req_state = END_REQ;
          seq_item_port.put(forked_rsp);
          num_requests--;
        end : request_process
```

```
      join_none : pipeline_fork
    end : get_and_drive
endtask : run_phase
```

Pipelined or single-threaded, our VIP developer is free to implement the driver as they wish, and at no point in time does the test writer interaction with the sequencer need to change.

## 9. Why not…?

While the coding patterns in section 8 may work generically, there are a few variations available for drivers which developers may wish to use. While these variations exist, they're not always as capable as the patterns, and often lead to unnecessary confusion. We will address a few of those alternatives here.

Alternative #1: Why not use `get_next_item`/`item_done` instead of `get`?

Due to the evolutionary process of the UVM (and the OVM [7] before it), the driver's sequence item port provides multiple mechanisms for accomplishing the same exact goal, as is shown in the chart below:

**Table 1 – Sequence Item Port Methods**

| Method "A" | Method "B" |
|---|---|
| `get(req);` | `get_next_item(req);`<br>`item_done(req);` |
| `if (has_do_available())`<br>`  get(req);`<br>`else`<br>`  req = null;` | `try_next_item(req);`<br>`item_done(req)` |

While the two methods are virtually identical in end result, the explicit necessity for an `item_done()` call in method "B" may tempt the developer to map some special meaning to `item_done()`... potentially confusing the high and low layers. If the VIP developer never uses `item_done()`, they're unlikely to map any special meaning to it.

Alternative #2: Why not use the built in events?

Since sequence items derive from `uvm_transaction`, they have certain built in events which can be used for expressing the current status: `begin_event` and `end_event`. One could be tempted to use the pass-by-reference implementation of the sequencer to bypass using the response path, and instead use these events to communicate the same information.

Unfortunately, these events are tied strictly to the recording interface, which means they may not align correctly with the VIP developer's intent.

For example, an interface may support read requests which have associated protocol responses. In this case, the session states may include BEGIN_REQ, END_REQ, BEGIN_RSP and

END_RSP. The `end_event` for the request would likely be triggered when the session reaches the END_REQ state... however there is more information that the sequence needs to wait for.

While this isn't the layering violation described in section 7, the cause is essentially the same: overloading the definition of one interface in an attempt to map another.

It's worth noting that even the UVM Register Layer implementation makes this particular mistake, relying on `end_event` to signal the response-ready completion of a request. This unfortunate implementation artifact has been the bane of many a VIP developer and register test writer.

Alternative #3: What about `uvm_push_driver`?

An often overlooked element of the UVM is the alternative "push" mode sequencer and driver. These constructs are intended to be used to provide a push-based flow, wherein instead of the driver calling `get(req)` on the specialized `uvm_seq_item_pull_port`, the sequencer would call `put(req)` on a classic `uvm_tlm_blocking_put_port`. These alternate-mode sequencers and drivers are yet another artifact of the UVM's evolutionary process, providing further areas of duplication within the standard.

While the push mode is entirely capable of being used as a transport layer, it does have one consequence which may not be desired. The underlying implementation of the push mode sequencer is constantly attempting to send new requests, similar to this example code:

```
forever
  begin : push_loop
    // Arbitrate next item
    this.get_next_item(req);
    // Push to driver
    req_port.put(req);
    // Mark item as done
    this.item_done();
  end : push_loop
```

While this may seem harmless enough, it's possible that the `put(req)` call may be blocked in the driver because the driver was not ready to handle a new request. In other words, while the sequence and sequencer were aligned, the driver was not. Unfortunately, if the sequence drifts out of alignment before the driver is ready, then the sequencer will end up transmitting an invalid request. This can happen if the sequence changes from relevant to not relevant, or if a higher priority sequence comes online.

It is worth noting that this alternate-mode mechanism is another great example of the difference between the application and transport layers. Sequences are generally unaware of which transport layer implementation the sequencer and driver have chosen to implement.

## 10.Conclusions

While the UVM provides basic structures for the creation of stimulus in the active domain of the testbench, it falls short of illustrating how those structures can be used in a consistent manner by VIP developers, leading to diverging implementations and training materials throughout the industry.

By leveraging the communication models used in modern networks, we can properly encapsulate VIP developer and test writer intent within higher level requests and responses, keeping that information separate from the low level details of how those requests and responses are transported between the sequence and the driver. This solution provides both VIP developers and end users with a functional, scalable, consistent and familiar pattern for stimulus generation.

## 11.References


[1] Universal Verification Methodology (UVM), "1.2 Class Reference," Accellera Systems Initiative, 2014.

[2] Network Working Group, "RFC-1122," Internet Engineering Task Force (IETF), 1989.

[3] "Internet protocol suite," 16 June 2015. [Online]. Available: https://en.wikipedia.org/wiki/Internet_protocol_suite. [Accessed 9 July 2015].

[4] ARM Limited, "AMBA 3 APB Protocol Specification," 2004.

[5] ARM Limited, "AMBA 3 AHB-Lite Protocol," 2006.

[6] IEEE 1666, "Open SystemC Language Reference Manual," 2011.

[7] Open Verification Methodology (OVM), "Class Reference," Cadence Design Systems, Inc. and Mentor Graphics, Corp., 2011.