There's something wrong between Sally Sequencer and Dirk Driver – why UVM sequencers and drivers need some relationship counselling

Mark Peryer
Mentor Graphics (UK) Ltd.,
Rivergate, London Road
Newbury, Berkshire, England
mark_peryer@mentor.com

Abstract **One of the drivers behind verification methodology is to allow engineers to solve complex problems with simple methods. However, the pace of technological change means that a system that is considered complex today will be viewed as a simple building block in a few years time. In order to keep up with this ever increasing complexity verification methodology has to evolve to ensure that working products can continue to be delivered by the electronics industry. The development of the Universal Verification Methodology (UVM) has been a welcome development and should enable a more coherent verification ecosystem. At the core of the UVM is a legacy stimulus generation architecture based on the use of sequence objects with sequencer and driver components. This paper makes a critical examination of this architecture and proposes an alternative based on TLM2 non-blocking transfers which have the potential to cope better with future verification needs.**

Keywords-UVM; stimulus; sequences; drivers; testbench architecture; TLM;

I. INTRODUCTION

The growing use of Transaction Level Modeling (TLM) in testbenches is one means by which verification engineers have been able to solve ever more complex problems by abstracting themselves away from the detail of signal level activities. TLM is used for both stimulus and observation, but the focus of this paper is on stimulus. There are various ways in which TLM stimulus can be implemented, but they all have the same characteristics:

- A pin level transfer is abstracted as a transaction data object
- A driver component receives transactions and converts them into patterns of pin level activity
- The stimulus process involves streaming transaction data objects to the driver

The measure of a good TLM methodology is that the stimulus writer should not need to know the detail of how the driver is implemented and that there is a clear and unambiguous API which, once learnt, can be reused across all environments. The only thing the stimulus writer should have to concern themselves with is the content of the transaction object and how that relates to the protocol. Unfortunately, with the UVM stimulus generation architecture this is not necessarily the case.

II. UVM STIMULUS GENERATION ARCHITECTURE

In the UVM class library, the uvm_driver component is responsible for converting abstract transactions into concrete patterns of pin level transfers on an interface of the DUT. The transactions that the driver receives are generated by a uvm_sequence and are referred to as sequence items. A sequence is a class which is generated on the fly and executes a time consuming task that either creates and starts other sequences or it generates sequence items. A sequence communicates with the driver through an intermediate component called the sequencer. The sequencer implements various TLM interfaces which the driver and the sequence use to communicate.

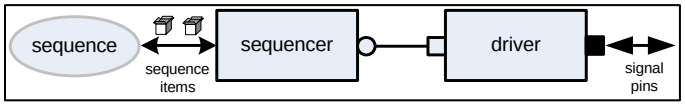

Figure 1 - UVM Stimulus Generation Architecture

The sequencer contains an arbitration mechanism that allows several sequences to be running concurrently, each one sending sequence items to the driver.

Historically, this stimulus generation architecture has its roots in the eRM[1] sequence architecture which was transposed to the OVM[2] and then inherited by the UVM[3]. As with all legacies, this architecture should be regularly challenged to determine whether there are better alternatives available. This paper will examine the architecture and its use models and then conclude with a discussion of an alternative based on TLM 2.

III. STIMULUS USE MODELS

In order to analyze the UVM stimulus architecture it is necessary to understand the sequence-sequencer-driver API and to explore a number of use cases.

A. The Sequence-Driver API

The UVM sequence-driver API probably provides adequate semantics for generating stimulus for around 80% of the interfaces that users encounter in practice. The API has two halves, the sequence side and the driver side. The

sequence uses two blocking methods to transfer request sequence items to the driver:

- **start_item(<item>)** – This requests the sequencer to have access to the driver for the sequence item and returns when the sequencer grants access.
- **finish_item(<item>)** – This results in the driver receiving the sequence item. When the finish_item() method unblocks is determined by an API between the driver and the sequencer.

The sequence handles responses in one of three ways:

- Using the handle to the <item> - since this is shared by the driver and the sequence
- **get_response(<item>)** – This is a blocking call which returns a response item explicitly returned by the driver
- **response_handler()** – A call-back that can be enabled to handle response items returned by the driver

The driver has a TLM uvm_sequence_item_pull_port which is connected to a corresponding export in the sequencer. The driver-sequencer API has a number of methods which are used to pull request transactions from the sequence via the sequencer and to return response transactions:

- **get_next_item()** – From the drivers perspective this is a blocking call which returns with a sequence item which it then translates into a pin level transfer. A get_next_item() call must be followed by an item_done()
- **item_done()** – The driver uses this non-blocking call to signal to the sequencer that it can unblock the sequences finish_item() method, either when the driver accepts the sequences request or it has executed it, or at some other point convenient to the driver.
- **try_next_item()** – This is a non blocking variant of the get_next_item() method. If there is a sequence item available it will return with its handle, otherwise it will return with a null handle.
- **get()** – This is a blocking call which has the same effect as calling get_next_item() and item_done() in one go. It unblocks the sequences finish_item() method immediately.
- **put(<item>)** – This is a non-blocking call which returns a response item to the sequence.

The sequence, sequencer and driver API is quite complex and knowing which call to use requires the user to be familiar with the underlying implementation of the driver and the sequencer.

B. Use Cases For the Sequence Driver API

The sequence driver API works fine for two use cases which probably allow users to model about 80% of the interface protocols that they will encounter – unidirectional and bidirectional transfers.

1) Unidirectional Transfers

In this use case, the sequence sends sequence items to the driver and the driver executes pin level transfers which require no response information to be returned to the sequence. The code example shows how the sequence uses start_item() and finish_item() to send sequence items to the driver and how the driver throttles the generation of further sequence items by the sequence by not calling item_done() until it has completed the transfer. This use model works well for simple interfaces.

```
// Unidirectional sequence example
class unidir_seq extends uvm_sequence #(uni_item);

task body;
  uni_item req = uni_item::type_id::create("req");

  start_item(req); // Blocks until sequencer ready
  if(!req.randomize()) begin
    `uvm_error("body", "Randomization failure")
  end
  finish_item(req); // Blocks until item_done()**
endtask: body
endclass: unidir_seq

// Unidirectional driver example
class unidir_driver extends uvm_driver #(uni_item);

task run_phase(uvm_phase phase);
  uni_item req;

  forever begin
    get_next_item(req); // Item from sequence via sequencer
    // do something with req
    item_done(); // ** Unblocks finish_item() in sequence
  end
endtask: run_phase
endclass: unidir_driver
```

Figure 2 - Unidirectional Sequence and Driver Code

2) Bidirectional Transfers

With bidirectional transfers, the overall control flow for the use model is the same as for the unidirectional use case, but the difference is that there is response data that needs to be transferred from the driver back to the sequence. In order to get to the response information, the sequence waits until the finish_item() method has unblocked and then accesses the properties within the sequence item which the driver has updated.

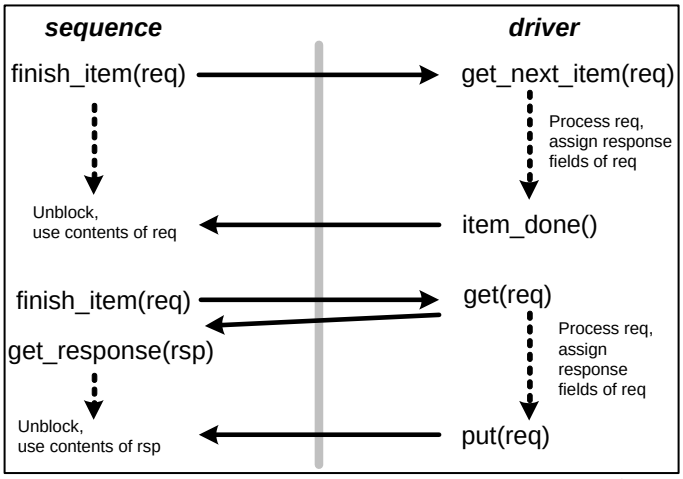

Figure 3 - Bidirectional use case sequence driver API control flow

Another implementation variant for the bidirectional use model is to use get() and put() in the driver to get the request sequence item and to return a response sequence item to and from the sequence. This approach requires the code in the sequence to an have additional get_response() call which blocks until the put() call is made by the driver. In order to accommodate the possibility that there might be multiple sequences running concurrently, the driver code also has to

clone() the original response item from the request item and set the sequence item id to ensure that it is returned to the right sequence. This adds complexity to the driver, but at least it is not evident to the sequence writer.

The salient point here is that there are multiple ways in which drivers can be implemented, and this means that the sequence writer has to know which use model to follow, violating the principles of TLM encapsulation

C. Use Cases That Stretch The Sequence Driver API

There are a number of more advanced use cases that are difficult to support using the blocking API between the UVM Sequence, Sequencer and Driver. These include pipelined transfers and pipelined transfers with out of order responses, examples of common protocols with these characteristics are the AMBA AHB bus and the AMBA AXI bus.

1) Modelling Pipelined Transfers

A pipelined transfer protocol is designed to maximise the bandwidth of an interface by executing multiple bus phases in parallel, it also has the advantage that it relaxes the timing requirements for a target to respond to a request. In the case of the AHB bus, the address phase for one bus cycle overlaps the data transfer for the previous one, but other pipelined protocols could have several phases overlapping each other.

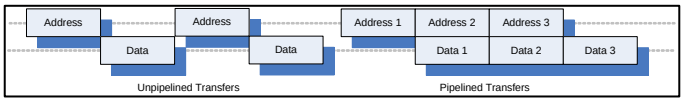

Figure 4 - Unpipelined vs. pipelined bus transfers

It is possible to model a single pipelined transfer using the same approach as a bidirectional transfer, by sending a sequence item to the driver and using item_done() when the bus cycle completes. However, this has the major disadvantage that the pipelined bus is effectively unpipelined since only one transfer can occur at a time. In order to fill the pipeline and have more than one sequence item being processed by the driver a change to the established UVM sequence, sequencer and driver use model has to take place.

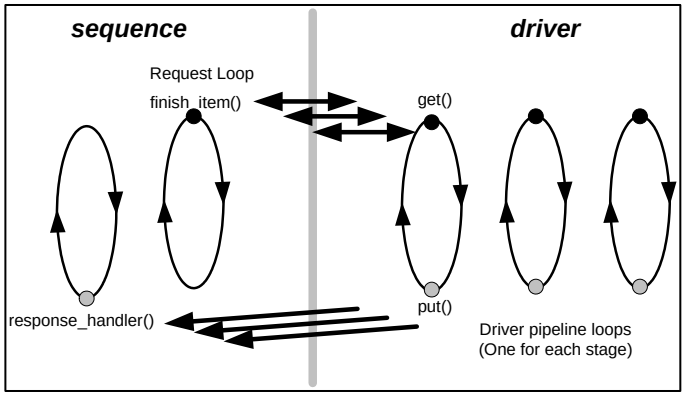

Figure 5 - Pipelined sequence and driver processes

The driver has to be changed so that it can get and handle the execution of multiple sequence items in parallel. An implementation that facilitates this is for the driver to have a bus_cycle() method that implements a complete bus transfer as represented by a sequence item, and then for the driver to spawn multiple threads executing that method concurrently. The method uses the get() method to acquire the next sequence item, this allows the sequences finish_item() method to unblock in order to send the next sequence item to keep the pipeline full. The bus_cycle() method uses a semaphore to lock access to the get() method, and to the bus resources associated with the initial phase of the protocol. The number of parallel bus_cycle() methods that the driver needs to run is equal to the number of pipeline stages in the protocol.

The sequence implementation also has to be modified, to have two loops – one for stimulus generation and the other for handling responses. In the stimulus loop, each finish_item() call will be unblocked as soon as the driver calls its get() method and the loop repeats to generate a new sequence item for the next bus cycle to keep the pipeline full. Note that a new sequence item does need to be generated since reusing the original sequence item will result in both the driver methods and the sequence working with the same object. In parallel, the response_handler() call-back can be used to handle responses.

```
// From the pipelined sequence:
task body;

mbus_seq_item req =
mbus_seq_item::type_id::create("req");

  // Enable response handler call-back:
  use_response_handler(1);
  // Generate Stimulus:
  for(int i=0; i<10; i++) begin
    assert($cast(req_c[i], req.clone()));
    start_item(req_c[i]);
    assert(req_c[i].randomize() with {…});
    finish_item(req_c[i]);
  end
endtask: body

function void response_handler(uvm_sequence_item response);
  mbus_seq_item rsp;

  if(!$cast(rsp, response)) begin
    `uvm_error("response_handler",
    "Failed to cast response to mbus_seq_item")
    return;
  end
  // Handle the response

endfunction: response_handler

// From the pipelined driver:
semaphore pipeline_lock = new(1); // Intialised as unlocked

// Spawn two threads - one for each pipeline stage
task run_phase(uvm_phase phase);
  fork
    do_pipelined_transfer;
    do_pipelined_transfer;
  join
endtask

task automatic do_pipelined_transfer;
  mbus_seq_item req;

  forever begin
    pipeline_lock.get();
    seq_item_port.get(req);
    // Do command phase
    // - unlock pipeline semaphore
    pipeline_lock.put();
    // Complete the data phase
    // Return the request as a response
    seq_item_port.put(req);
  end
endtask: do_pipelined_transfer
```

Figure 6 - Code example for pipelined sequence and driver implementation

So far, the normal UVM API has sufficed, albeit with some careful implementation. However, the fundamental problem is that the sequence does not know when the driver has completed a bus cycle and that it is safe to retrieve response information from the handle to the relevant sequence item. This can only be achieved by using the put() method in the driver to return responses and to signal the completion of

the bus cycle. Any number of pipeline stages can be modeled this way, but at the cost of an increasingly more complex implementation.

An alternative way to support further phases is to extend the sequence item by adding events to it. The events are then used to signal between the driver and the sequence that a particular phase is complete.

2) Modelling Pipelined Out Of Order Transfers

More advanced bus protocols, such as AMBA AXI, support pipelining with out of order responses. In other words, an initiating master makes a request for a transfer, but does not wait for the response before making other requests. The target slave responds when it is ready and the order in which the responses occur is not necessarily the same as the request order.

In the case of the AXI bus, there are separate channels for read and write transfers. This means that at any one time there could be a read request, a write request, a write data transfer, a write response and a read response phase all occurring in parallel.

Most of the techniques used for handling pipelined transfers are applicable to out of order transfers. However, an ID field is added to the bus so that responses can be tallied against their originating requests. The ID field is added to the sequence item for generation and this property helps the driver and the sequence track responses.

The driver implementation has to be enhanced with extra methods to implement the read and write request cycles, and the read and write response cycles. These are run in parallel threads with multiple read and write request cycle methods, according to the number of phases in the bus protocol. Request sequence items are put into a data structure once they have been executed on the bus. The read and write response methods monitor the response channels of the bus and retrieve the request sequence item from the data structure based on the ID of the response. They populate the response fields based on the bus response data and then send the response to the sequence.

As with the pipelined use model, the sequence is organised into a generation loop and a response handling loop. The generation loop does not change, but the response loop has to keep track of responses using an array of expected response sequence items since there are no guarantees about the order of the responses.

3) Real Breakdown – Handling The Unexpected

So far, the UVM Sequence, Sequencer and Driver API has been shown to work with the various use cases, albeit by requiring the sequence writer to have to understand how the driver is implemented or by extending the API to cope with the decoupled responses of pipelined protocols. However, the implicit assumption has been that the transfers are error free and suffer no disruptive events. In practice, a DUT may contain a protocol error or the verification plan may call for certain protocol errors to be modelled in order to check that the DUT can cope with them. In the case of a hardware reset or a protocol error, there may be a need for the driver to abort a bus cycle during execution and for the sequence to be able to handle this eventuality. In time, developments in UVM phasing may also introduce further requirements to handle disruptive changes during phase transitions.

Handling these events is not necessarily easy to implement and they need to be considered when the sequence and the driver are designed. Adding the capability to cope with these features to the pipelined use model implementations can be particularly problematic. The sequencer does contain methods which allow a sequence to be aborted, but these can easily create a dead-lock with a driver attempting to return responses to a sequence that the sequencer no longer recognizes, and a sequence which has become disconnected from the driver expecting responses.

Essentially, the only way that disruptive events can be handled is for the driver to contain code that handles them as an exception to be layered on top of the API by encoding a response status field in sequence items to indicate that a reset or a protocol error has resulted. The sequence also has to contain exception code to handle an aborted transfer. This adds further complexity to the API.

IV. An Alternative Stimulus Generation Architecture

Since the whole point of using sequences to send transaction level sequence items to drivers is to abstract difficult behavior for testbench users, the arcane nature of the UVM sequence-driver API suggests that there may simpler ways to approach the problem. In this section, an alternative implementation using TLM 2 is considered.

A. Using A TLM 2 Implementation

The UVM implements the TLM 2 protocol originally developed by OSCI for SystemC[4]. In SystemC TLM 2 differs from TLM 1 in several ways, the most important of which is that transfers are bidirectional since it is a reference to the transaction that is passed between the initiator and target socket. In SystemVerilog it is a handle to a transaction object that is passed, so even in TLM 1 there is an implicit return path via the handle to the transaction.

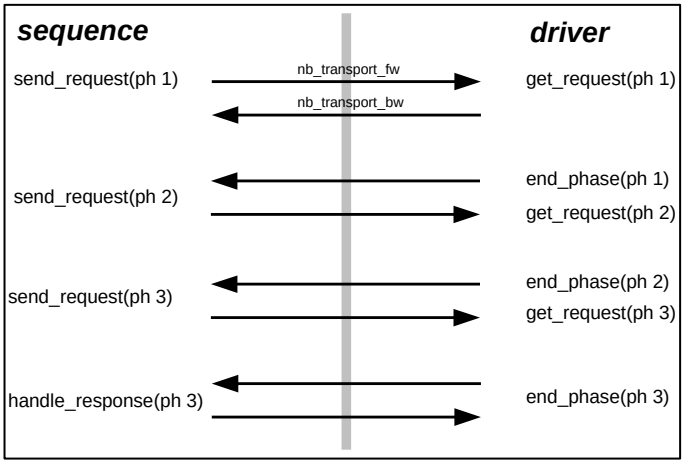

Figure 7 - TLM 2.0 Non-blocking sequence-driver implementation model

As part of its bidirectional transfer semantic, TLM 2 supports blocking and non-blocking transfers. Using blocking transfers as the basis of a sequence-driver implementation would require less implementation and user overhead for the basic use models. However, TLM 2 blocking transfers would

still suffer from the same draw-backs as the current UVM sequence, sequencer and driver API implementation since they rely on the unblocking of the method to indicate completion.

Using TLM 2 non-blocking transfers offers an alternative approach whereby it is possible to transfer a transaction between initiator and target sockets several times using phase and response fields to indicate a progression through a series of states. There is a forward transport method implemented in the target socket, and a backward transport method implemented in the initiator socket. The target can call the backward transport method independently of the initiator multiple times and can return different phase and status flags depending on the nature of the protocol, effectively implementing a state machine. This capability gives scope to handle more complex protocols.

```
// From TLM2 sequence base class:
virtual class tlm2_nb_seq_base #(type T = uvm_tlm_generic_payload)
extends uvm_object;

// Initiator Socket Handle
uvm_tlm_nb_initiator_socket #(tlm2_nb_sequencer #(T), T)
initiator_skt;

// Sequencer handle:
tlm2_nb_sequencer #(T) m_sequencer;
// Start method
virtual task start(tlm2_nb_sequencer #(T) parent_sequencer);
  m_sequencer = parent_sequencer;
  parent_sequencer.source_sequence = this;
  initiator_skt = m_sequencer.initiator_skt;
  body();
endtask: start

// For sending a transaction to the driver
virtual task do_item(T item);
  uvm_tlm_time delay = new;
  uvm_tlm_phase_e phase ;
  uvm_tlm_sync_e sync;

  sync = initiator_skt.nb_transport_fw(item, phase, delay);
  m_sequencer.cmd_lock.get(); // End of cmd phase
endtask: do_item

// For handling responses
virtual function void response_handler(uvm_object t);

endfunction: response_handler
```

```
// From TLM2 sequencer - T is the type parameterization
semaphore cmd_lock = new(0);
semaphore data_lock = new(0);
T respQ[$];
uvm_object rspQ[$];

tlm2_nb_seq_base #(T) source_sequence;

uvm_tlm_nb_initiator_socket #(tlm2_nb_sequencer #(T), T)
initiator_skt;

// Called from target
function uvm_tlm_sync_e nb_transport_bw(T t,
           ref uvm_tlm_phase_e p, uvm_tlm_time delay);
  case(p)
    END_REQ: cmd_phase_ended();
    END_RESP: data_phase_ended(t);
  endcase

  return UVM_TLM_ACCEPTED;
endfunction: nb_transport_bw

function void tlm2_nb_sequencer::cmd_phase_ended();
  cmd_lock.put();
endfunction: cmd_phase_ended

function void tlm2_nb_sequencer::data_phase_ended(T t);
  respQ.push_back(t);
  data_lock.put();
endfunction: data_phase_ended

task tlm2_nb_sequencer::run_phase(uvm_phase phase);
  uvm_object rsp;
  forever begin
    data_lock.get();
    wait(respQ.size > 0);
    rsp = respQ.pop_front();
    source_sequence.response_handler(rsp);
  end
endtask: run_phase
```

```
// From the driver:
virtual class tlm2_nb_driver #(type T = uvm_tlm_generic_payload)
extends uvm_component;

// Item Queue:
T itemQ[$];

// Target socket using defaults:
uvm_tlm_nb_target_socket #(tlm2_nb_driver #(T), T) target_skt;
// Called to accept a command:
function uvm_tlm_sync_e tlm2_nb_driver::nb_transport_fw(T t, ref
uvm_tlm_phase_e p, uvm_tlm_time delay);
  itemQ.push_back(t);
  p = BEGIN_REQ;
  return UVM_TLM_ACCEPTED;
endfunction: nb_transport_fw

// Called to signal command completion
function void tlm2_nb_driver::request_done(T t);
  uvm_tlm_phase_e p = END_REQ;
  uvm_tlm_time delay = new();
  uvm_tlm_sync_e sync;

  sync = target_skt.nb_transport_bw(t, p, delay);
endfunction: request_done

// Called to return a response:
function void tlm2_nb_driver::return_response(T t);
  uvm_tlm_phase_e p = END_RESP;
  uvm_tlm_time delay = new();
  uvm_tlm_sync_e sync;

  sync = target_skt.nb_transport_bw(t, p, delay);
endfunction: return_response

// Called to get a request:
task tlm2_nb_driver::get_request(output T t);
  wait(itemQ.size > 0);
  t = itemQ.pop_front();
endtask: get_request
```

```
// From the sequence:
task body();
  repeat(8) begin
    assert($cast(req_c, req.clone()));
    assert(req_c.randomize());
    req_c.id = wr_id;
    req_c.address = current_address;
    wr_req[wr_id] = req_c;
    do_item(req_c);
    case (req_c.opcode)
      SINGLE: current_address += 4;
      BURST4: current_address += 16;
      BURST8: current_address += 32;
    endcase
    wr_id++;
  end

function void response_handler(uvm_object t);
  // Cast object to sequence_item
  // Handle response
endfunction
```

Figure 8 - Experimental TLM 2 Sequence, Sequencer, Driver Code

Unfortunately, there are several issues with the current UVM (UVM 1.1a) implementation of TLM 2. The first is that the initiator and target sockets are uvm_components and therefore have to be created during the build phase and have to be bound together during the connect phase. The second is that the target backward transport method has to be implemented inside a component. This means that for a particular protocol, the backward transport method would have to be implemented inside the equivalent of the existing sequencer, and an alternative API implemented between the sequence and the sequencer so that the sequence can use the initiator socket.

The example code in figure 8 illustrates how a pipelined protocol might be modelled using TLM 2 in the UVM given the current limitations of the implementation. For the sake of brevity and clarity, the example has been constrained to only support the execution of a single sequence stream on the

driver. Nether the less, this experimental implementation shows that the API can be simplified for the sequence and the driver writer to just a few methods with a consistent interface.

With more complex protocols the sequencer will most likely require an extended implementation to take into account additional phases and transfers. The handling of disruptive events such as resets, protocol errors and UVM phase changes could be coded into the non-blocking transfer protocol so that the driver could enter a recovery state whilst returning an exception response status to the sequences from which it is currently processing transactions. An exception handler in the sequences could then flush their event queues and safely decouple themselves from the driver. The VIP developer would handle this additional complexity, leaving the sequence writer with a straight-forward API.

A more elegant way to code the transfer protocol between the sequence and the driver would have been for the sequence to have an initiator socket that could be bound directly to the drivers target socket. This implies that the sequences initiator socket could be created and bound dynamically during any phase, something which is not currently possible with the UVM. The sequence side initiator socket backward transport method would handle the protocol and the relevant methods would be in a TLM 2 sequence base class.

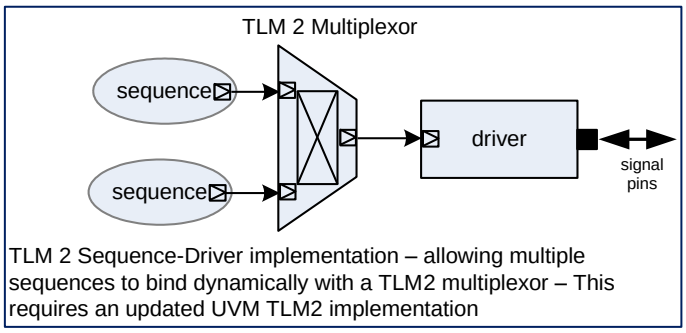

Figure 9 - TLM2 Sequence-Driver Architecture

In practice, there will also be a requirement for several concurrent sequences to be communicating with the driver. This implies a need for an intermediate sequencer-like component to act as a multiplexor. The multiplexor would contain an initiator socket that is bound to the driver target socket, and a dynamic array of target sockets. With this architecture, when a sequence started, it would create its initiator socket and add a target socket to multiplexors target socket array and then bind the two together. The multiplexor would be responsible for arbitrating between, and routing, the sequence items to the driver and returning any backward response back to the sequence via the various intermediate sockets. The arbitration algorithm used by the multiplexor could be defined using a policy class. In order to generate stimulus for a layered protocol, sequences could be layered using layers of TLM 2 intermediate sockets which modify or pack sequence item content into alternative sequence items.

The TLM 2 functionality in the UVM is relatively immature and is not completely aligned with SystemC in terms of socket functionality. Since the UVM TLM2 base classes have not been widely adopted by users, there is some justification for reconsidering their implementation and that of TLM (1 & 2) ports generally. This would allow the proposed sequence, multiplexor, driver architecture to be implemented as proposed.

V. Conclusion:

Although adequate for most stimulus generation problems to date, the UVM sequence generation architecture is complex and requires that the sequence writer has to understand the way in which the driver has been implemented. Some of the inevitable confusion and complexity could be taken away by an alternative approach based on TLM 2 non-blocking transfers which simplifies the stimulus generation API and improves the modularity of the code. This all anticipates some future work by the UVM developers which would allow TLM2 sockets to be dynamically created and bound in any phase.

Acknowledgments

I should like to thank my colleagues Adam Rose, Adam Erickson, Rich Edelman and Gordon Allan of the Mentor Graphics Verification Methodology Team for their tolerance and encouragement of a healthy debate about alternative stimulus architectures.

References

[1] e Reuse Methodology (eRM) Developer Manual – Version 4.3.1 – Verisity Design Inc - 2004

[2] OVM 2.1.2 Users Guide. Mentor Graphics and Cadence Design Systems – 2011

[3] UVM 1.1a Users Guide. Accellera – 2011

[4] OSCI TLM 2.0 Language Reference Manual – Software Version 2.0.1 OSCI - July 2009

[5] Meyer, A., 2009. Overview of Sequence Based Stimulus Generation in OVM 2.0 – Application note. Mentor Graphics Corporation

[6] Edelman, R., 2008. Sequences in SystemVerilog – Proceedings DVCon 2008

[7] OVM/UVM Methodology Cookbook – Mentor Graphics Corporation – 2011 - http://verificationacademy.com/uvm-ovm