MASTERING REACTIVE SLAVES IN UVM

Mark Litterick, Jeff Montesano

Verilab

Munich (Germany), Austin (USA)

www.verilab.com

ABSTRACT

In most interface protocols a component can either be a master, which initiates the transactions or a slave, which responds to these transactions. Generating constrained-random request transactions in a proactive master agent using sequences is fairly straightforward in the UVM; however, implementing a reactive slave is much more complicated, especially for relatively inexperienced users. The implementation can be further complicated if the slave has a storage component or we are required to synchronize high-level scenarios with slave traffic.

This paper outlines the roles and responsibilities of a reactive slave and proactive master and then explores different architectures for reactive slave implementation, highlighting their suitability for a protocol depending on the decoding of the transactions in the monitor.

All aspects of reactive slave operation are illustrated with code examples, including architecture, sequence items, forever sequences, TLM interconnection, storage API and synchronization agents.

Table of Contents

MASTERING REACTIVE SLAVES IN UVM 1
1. Introduction 3
2. Proactive Masters and Reactive Slaves 3
    2.1 Proactive Master Architecture 5
    2.2 Reactive Slave Architecture 6
    2.3 Alternative Slave Architecture 7
    2.4 BFM Implementation 9
3. Reactive Slave Operation 9
    3.1 Sequence 10
    3.2 Monitor 10
    3.3 Sequencer 11
    3.4 Driver 12
    3.5 Agent 12
    3.6 Test Environment 13
4. Additional Features 14
    4.1 Reactive Slave With Memory/Storage 14
    4.2 Reactive Slave With Control Agent 16
    4.3 Error Injection In Slave Responses 19
    4.4 Bus Topology Considerations 21
5. Conclusions 22
6. References 22

1. Introduction

In most interface protocols a component can be classified as either a master, which initiates the transactions on the bus, or a slave, which responds to these transactions. From a verification point of view, a reactive slave is an agent that drives response signals only when a transfer is initiated on its interface by the Device Under Test (DUT) master. It is an active component as it affects the state and flow of the simulation by driving signals on the interface to specific values at specific times. Such reactive slaves are sometimes referred to as "responders".

Generating constrained-random request transactions in a proactive master agent using sequences is fairly straightforward in the Universal Verification Methodology (UVM) [1]; however, implementing a reactive slave is much more complicated, especially for relatively inexperienced users. This is primarily because the slave sequence needs to autonomously generate sequence items on-demand depending on the requests from the DUT which are observed at unpredictable times. The implementation can be further complicated if the slave has a storage component or we are required to synchronize high-level scenarios with slave traffic.

This paper outlines the roles and responsibilities of a reactive slave and proactive master and then explores different architectures for reactive slave implementation. All aspects of reactive slave operation are illustrated by code examples; including architecture, sequence items, forever sequences, TLM interconnection, and use cases for tests. In addition we also demonstrate how and where to connect storage while providing the user with a flexible API to prefill, validate or interfere with the storage. Synchronization requirements for higher-level scenario sequences that need to wait on slave transactions from the DUT are also discussed and a flexible solution using control agents is provided . Lastly, we touch upon the topic of error injection as it relates to reactive slave implementation.

This version of the paper has additional text, clarifications and code fixes compared to [4].

2. Proactive Masters and Reactive Slaves

Most interface protocols can be decomposed into master and slave roles, where the master initiates a transaction on the interface and the slave responds to it. In a verification component (UVC), the classification of the role depends on the associated behavior of the DUT on the interface:

- UVC master initiates transactions for a DUT slave
- DUT master initiates transactions for a UVC slave

From a testbench point of view, if the slave agent drives any signals whatsoever into the DUT (e.g. ready, clear-to-send, data, response, error, etc.), then the slave's role is considered active. The master agent is of course responsible for initiating traffic to the DUT and is always performing an active role. In practice most slave protocols are active (in that the slave drives signals back to the master), with a few notable exceptions being a display interface, or other write-only interfaces that have no feedback at all.

It is important to avoid confusing the master and slave roles on an interface protocol with the active/passive mode of operation of a verification component. A verification component operating in active mode drives signals to the DUT or otherwise affects the stimulus, whereas a verification component operating in passive mode only observes DUT behavior without affecting signal values or timing. Hence if we observe traffic on an internal interface of the DUT (where the master and slave roles are both performed by RTL components) then we typically connect a verification component configured for passive operation; this passive agent could be connected to either the master or the slave end of the internal interface protocol (or indeed multiple passive agents could be used and connected to all endpoints).

This distinction between active and passive modes of operation in combination with master and slave roles is illustrated in Figure 1. It shows an environment with three different protocol UVCs, two of which (A and C) perform active roles (master and slave respectively) on external DUT interfaces, and one of which (B) performs passive monitoring of an internal protocol interface.

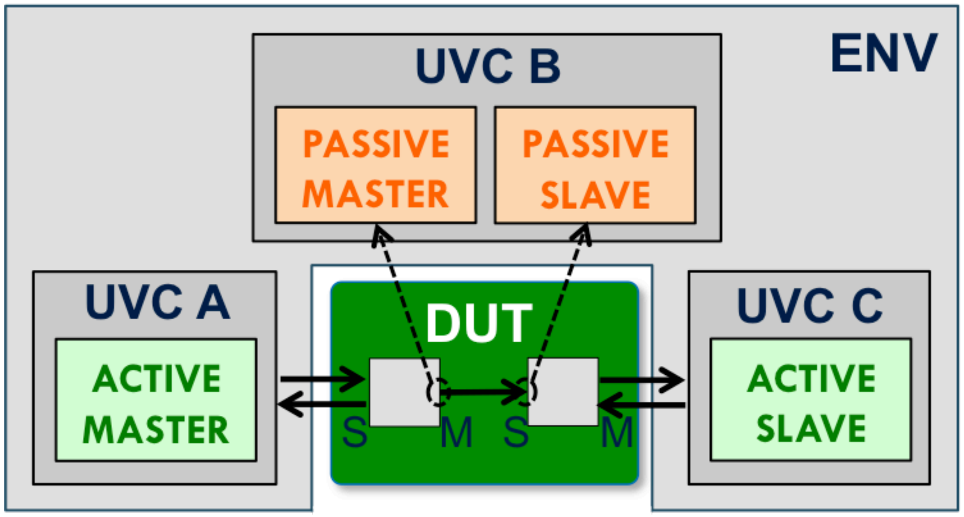

Figure 1 Active/Passive Operation of Master/Slave Agents

The responsibilities of the master and slave verification components can be further clarified by considering the stimulus flow. To generate traffic via the active master, the test must take charge and execute sequences as required, whenever it wants, in order to create interesting scenarios - this is *proactive* behavior. However, when the DUT assumes the role of master and the verification component is the slave, then we need to generate responses on-demand based on requests coming out of the DUT, whenever the DUT generates them - this is *reactive* behavior.

Note that although the DUT in a verification environment is provoked into generating the requests on the master interface output ports, it is often not possible to predict the timing of this traffic (for example where the DUT executes internal microcode on an embedded processing element which results in some subsequent traffic at some unpredictable time based on system state and current firmware version). Hence the slave needs to operate in an autonomous fashion and provide these responses when required without blocking the main stimulus flow for the test.

Verification components in a modern constrained-random setting can normally therefore be described as:

- **proactive master** which initiates traffic to the DUT using sequence-based stimulus
- **reactive slave** which responds to traffic from the DUT using sequence-based stimulus

In fact there are a few more variations than those listed above, but they are generally not appropriate. For example we could have an active slave that generates stimulus responses which have no relationship whatsoever to the traffic initiated by the DUT master (an ignorant active slave!) - but this would seriously compromise the ability to control stimulus and achieve meaningful scenarios for the verification requirements. Likewise we can also have a non-sequence based Bus-Functional Model (BFM) implementation whereby the response is automatic and not under the control of sequence-based constrained-random stimulus - this might be OK for very trivial slaves with only one response, but in general is strongly discouraged (refer to the section on BFM Implementation for more details). A sequence-based stimulus approach for reactive slaves is the most effective way to satisfy any non-trivial set of verification requirements. It allows the engineer to describe response sequence items along with constraints, and then leverage the power of randomization to achieve coverage closure.

The basic behavior and stimulus flow for a proactive master verification component is shown in Figure 2. In this case the stimulus is explicitly initiated by the test sequences (1), resulting in a request being sent to the DUT from the proactive master agent (2), and the DUT generating the response in accordance with the protocol (3).

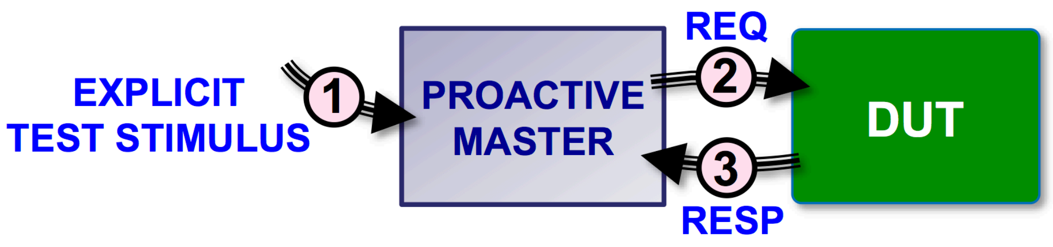

Figure 2 Proactive Behavior

The basic behavior and stimulus flow for a reactive slave verification component is shown in Figure 3. In this case the DUT is provoked into generating a request as a result of some remote stimulus or implicit behavior (1), the DUT generates the actual request whenever it wants (2), and the reactive slave agent generates a corresponding response in accordance with the protocol and verification requirements (3).

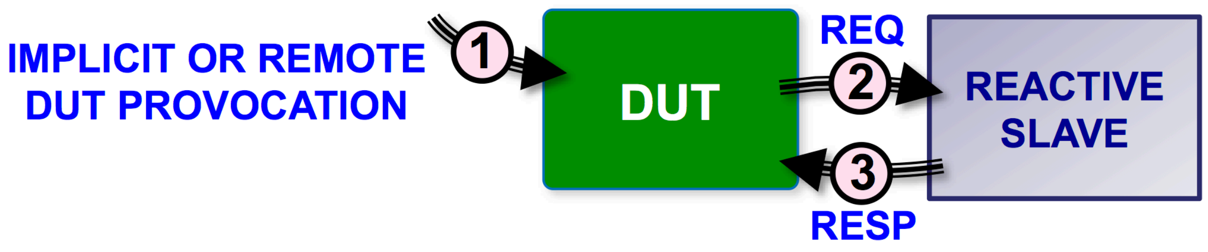

Figure 3 Reactive Behavior

2.1 Proactive Master Architecture

The most common type of verification component is the proactive master (where the user instructs the UVC to generate traffic for the DUT by executing one or more sequences in the context of a test scenario). The sequence flow is under the control of the user (via the test) but of course we may allow a varying degree of constraints to be placed upon the stimulus generation (configuration, sequences, scenarios). The basic UVM architecture for a proactive master component is shown in Figure 4.

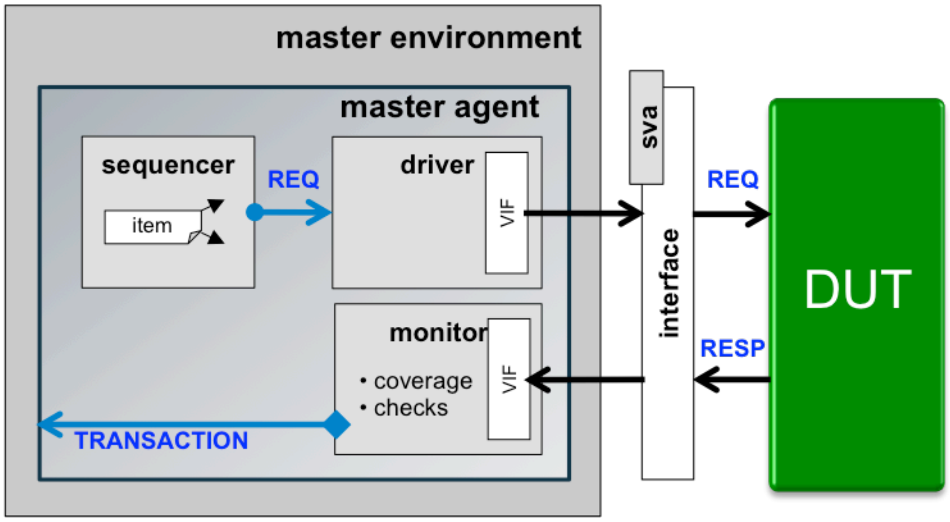

Figure 4 Proactive Master Architecture

(Actually this is the same architecture for all modern HVL-based constrained-random testbench languages, only the terminology changes.)

The key thing to bear in mind is that the structure supports all of what we want to do, uses sequence-based stimulus (as opposed to methods), and the test environment controls the stimulus flow and sequence order (allowing of course for randomization of content and permutations) rather than the DUT. So the sequences are started on the sequencer, sequence items are passed to the driver, time is consumed by the driver as it converts the sequence item to signal stimulus, once the item completes control is passed back to the sequencer, and then new sequences are started on demand. The DUT can of course influence the response timing, provide ready signals or clear-to-send flow control, and the UVC's driver must react to these correctly.

2.2 Reactive Slave Architecture

When the DUT is master of an interface and the UVC has to generate responses based on the observed requests, then the UVC behaves as a reactive slave. Typically the response information and flow control signals are required to be related to the requests provided by the DUT (e.g. for functional reasons such as when data belongs to a request tag or address, or for verification requirements such as we want to generate an error response on a particular address or type of transfer). The basic UVM architecture for sequence-based constrained-random stimulus for a reactive slave is shown in Figure 5.

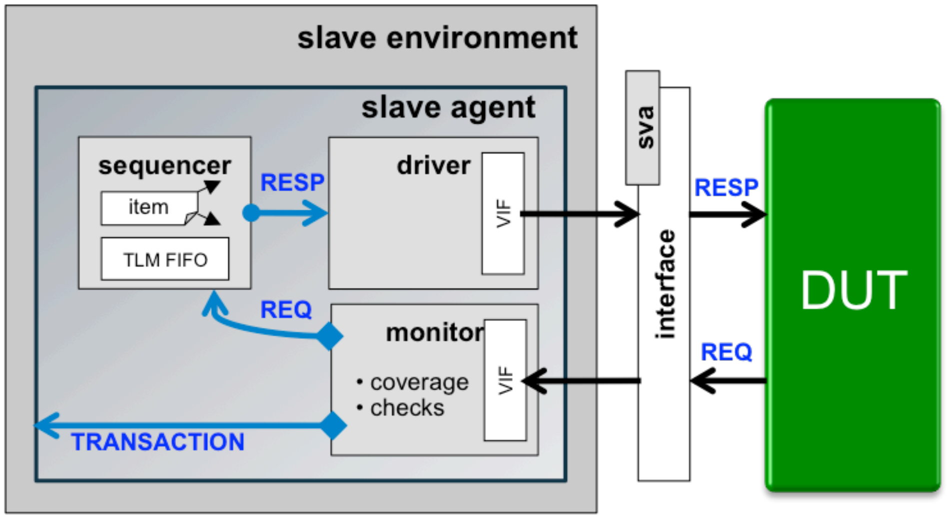

Figure 5 Reactive Slave Architecture

Note that the architecture is not so different from the proactive master (e.g. each agent has an active sequencer and driver, as well as a passive monitor, and the same agent and environment encapsulation). In this case there is additional communication from the monitor to the sequencer in order to generate sequence items as late as possible in reaction to requests from the DUT (i.e. we need to communicate the request information to a sequence for sequence item generation) via the TLM port connection and a TLM FIFO in the sequencer. Note that the FIFO is only there to provide blocking *get* capability for the TLM analysis port connection (it is not storing multiple transactions). Also, since we do not know when exactly the DUT is going to initiate the transaction we need to run a different kind of sequence. Operation of this architecture is described in detail in Section 3. ; although the infrastructure has become slightly more complicated, the basic communication between the sequences and driver is very straightforward as shown in Figure 6. Note that the driver operation is identical to a normal proactive master driver for this architecture.

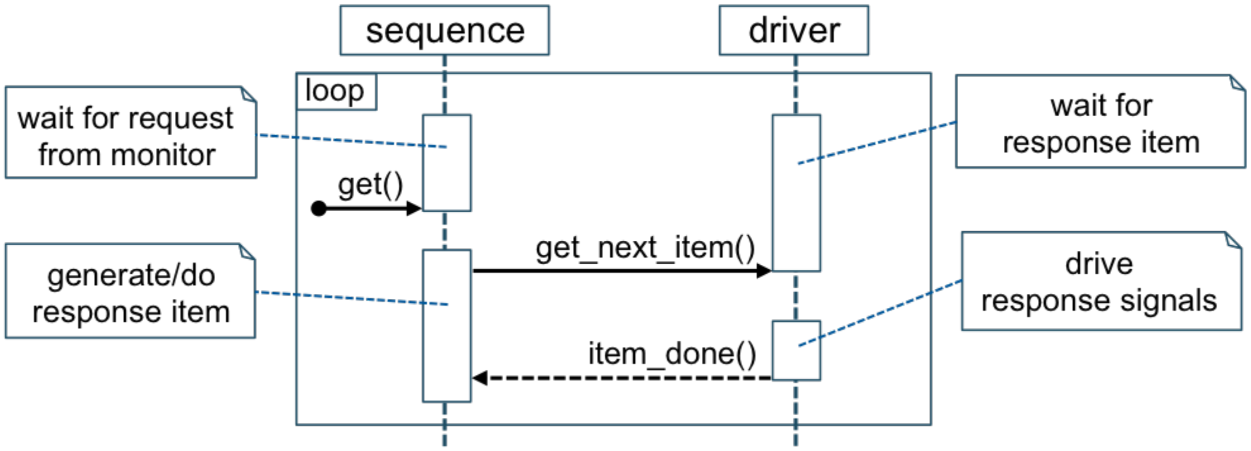

Figure 6 Reactive Slave Sequence-Driver Interaction

2.3 Alternative Slave Architecture

Note that there is more than one possible architecture for implementing a reactive slave. For example [2] describes an alternative implementation whereby the driver decodes the request (in parallel with the normal monitor operation) and communicates two times with the sequencer for every transaction (once to send the observed request, and once to get the generated response). The main advantage of this alternative setup is that the architectural structure is simpler, in fact it is identical to a normal proactive master architecture, as shown in Figure 7.

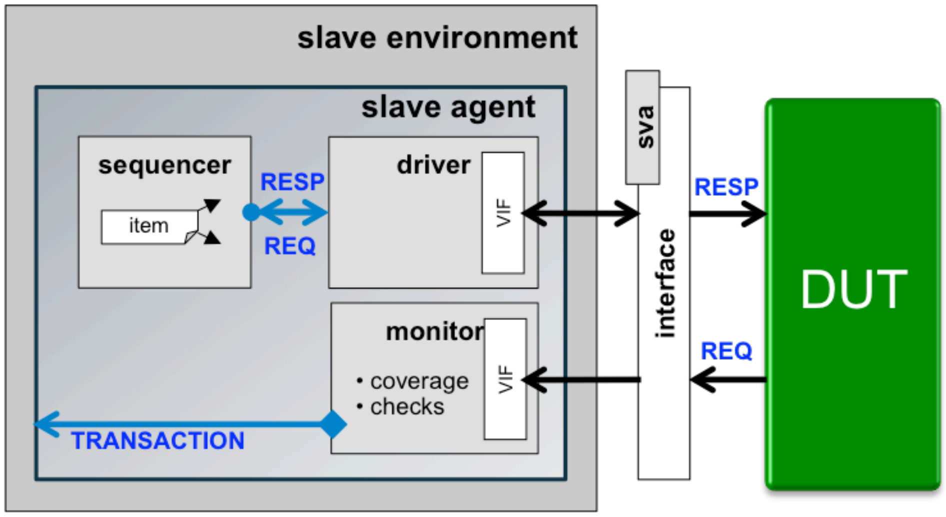

Figure 7 Alternative Slave Architecture

In this case the interaction between sequences and the driver is however more complicated, as shown in Figure 8, and the decoding logic for the request has to be duplicated in the driver (it is required in the monitor anyway).

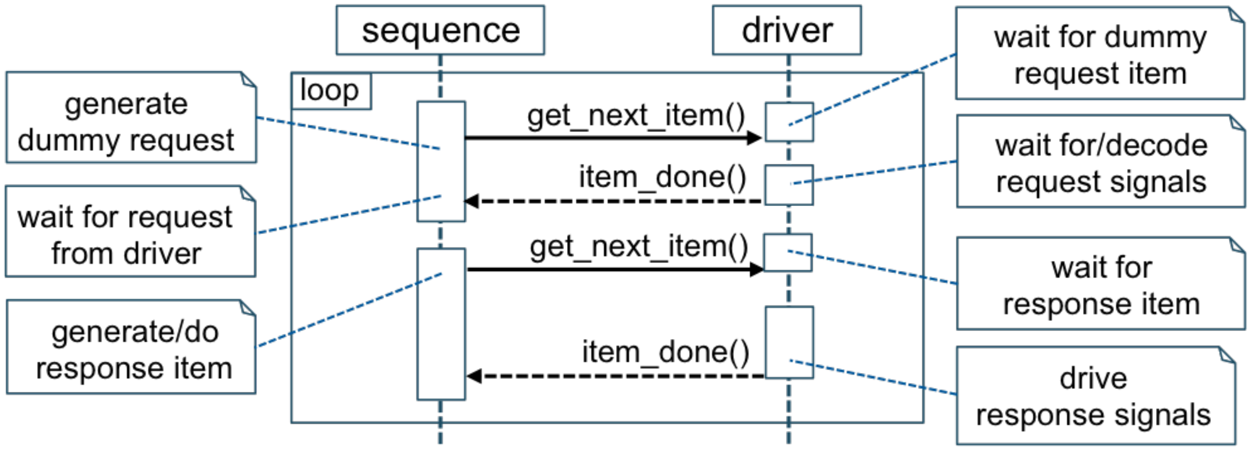

Figure 8 Alternative Slave Sequence-Driver Interaction

Although the infrastructure is slightly simpler in this case, the authors believe the additional complications introduced into the sequences and driver mean that this approach is less logical, more error-prone and harder to maintain. In addition, if there is any non-trivial decoding (e.g. number of start bits or gap time for a serial protocol), or complicated decoding (e.g. 8b10b symbol decoding), or arbitration (e.g. multiple masters contending for signal state) associated with the protocol, then it does not make sense to duplicate this logic in both the driver and monitor - remember the monitor always has to do the decoding in all these architectures in order for the verification component to work in passive mode.

The remainder of the paper is based around the architecture described by Figure 5, but could nonetheless be adapted to this alterative slave architecture by the reader if required.

2.4 BFM Implementation

Beginners, verification engineers from an HDL background (i.e. experienced with testbenches written in VHDL or Verilog) and lazy people (sorry, but true) are often tempted to implement the reactive slave entirely in a reactive driver which performs the role of a Bus-Functional Model (BFM). In this case all the advantages of sequence-based constrained-random stimulus are discarded in favor of simple RTL-like behavior (typically controlled via a configuration method API) as shown in Figure 9.

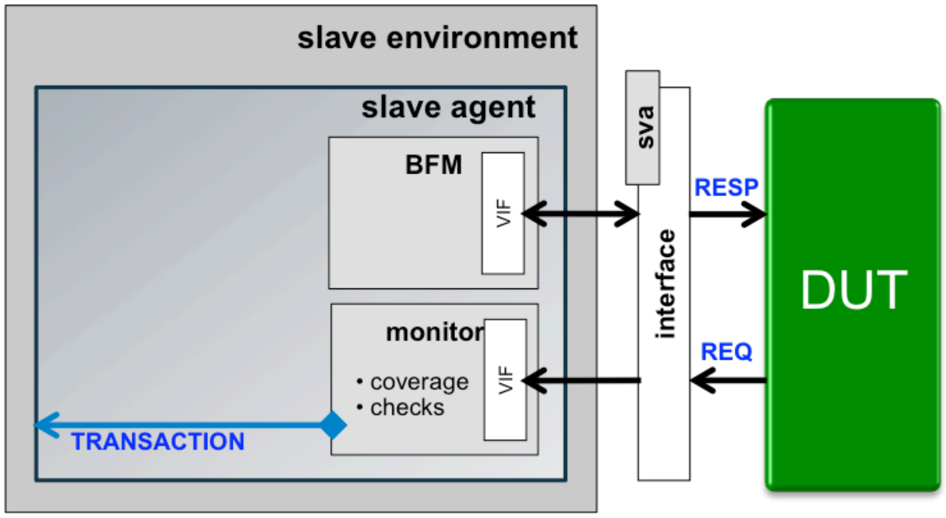

Figure 9 BFM Implementation

For trivial protocols this BFM solution might in fact be OK (e.g. where there is only ever one response, no wait state control, etc.). In fact it might also be OK as a dramatic temporary shortcut, but it is seldom the best solution for achieving user control over all of the responses in a constrained-random manner since the user would have to interfere with the driver behavior instead of just providing new scenario-specific sequence combinations. We would recommend strongly against this approach.

3. Reactive Slave Operation

An overview of the sequence-driver interaction for the reactive slave architecture of Figure 5 is provided by Figure 6. This section provides a more detailed explanation and UVM code examples for all aspects of slave operation, which can be summarized as:

- the sequencer runs a "**forever**" **sequence** waiting for and responding to requests from the DUT
- the monitor passively decodes all interface traffic but **publishes** transaction **requests** in addition to publishing complete transactions
- the **sequencer subscribes** to the published requests and sequences use this information to **generate response** sequence items
- the driver converts the sequence item into a **response** that is **driven** on the bus signals in accordance with the protocol requirements

Note that the code examples shown in the following sections assume that a base transaction is defined for the protocol, and that the sequence item extends this to add constraints and control variables (our preferred style); if you use only one item for both monitor and driver then adapt the code accordingly.

3.1 Sequence

As mentioned previously, the sequences for the reactive slave contain a forever statement. These sequences run forever in the allocated run-time phase and therefore they do not raise and drop objections. An example forever sequence is shown below:

```
class my_slave_response_seq extends uvm_sequence #(my_slave_seq_item);
  my_slave_seq_item m_item;
  my_transaction m_request;
  `uvm_object_utils (my_slave_response_seq)
  `uvm_declare_p_sequencer (my_slave_sequencer)
  function new(...)
  virtual task body();
    forever begin
      // wait for a transaction request (get is blocking)
      p_sequencer.request_fifo.get(m_request);
      // generate response based on observed request, e.g:
      case (m_request.m_direction)
        MY_DIRECTION_WRITE : begin
          `uvm_do_with(m_item,{
            m_item.m_resp_kind == MY_RESPONSE_ACK;
          })
        end
        MY_DIRECTION_READ : begin
          `uvm_do_with(m_item,{
            m_item.m_resp_kind == MY_RESPONSE_DATA;
            m_item.m_data == get_data(m_request.m_addr);
          })
         end
       endcase
     end
   endtask
endclass
```

Typically not many sequences are required to support all the slave functionality, but the user will probably have to provide additional sequences to do very specific things (for example generate an error response on the Nth access to a specific address). As a starter we would provide something like the following sequences as a template (actual sequences of course depend on protocol):

- my_slave_base_seq
- my_slave_response_seq - normal response sequence, non-error response for valid requests, error response for any invalid requests
- my_slave_error_seq - generate a trickle of error responses even for valid requests (e.g. random distribution of DUT master requests terminated with an error response)

Note: many slaves have to model storage (typically small memories), since the DUT could write to the slave and then read back later. The normal responses would make use of this storage accordingly as shown in Section 4. .

3.2 Monitor

The monitor for a reactive slave agent has one additional responsibility compared to a standard monitor - this is to publish the transaction request phase information in a timely manner so that the sequences can react and generate an appropriate response. This can either be done by publishing clearly marked partial transactions on the existing analysis port (which requires all users of the analysis port to filter the transactions accordingly, and is therefore not recommended), or to provide a separate port for publishing requests only (recommended).

It is recommended that the monitor publishes the requests via an analysis port since it is easier to handle passive operation of the UVC. The monitor does not care about the number of subscribers or the fact that one of them is a sequencer running a forever sequence, it just publishes the transaction request information (typically a partial transaction rather than a new transaction kind) on one analysis port when observed, and the full transaction on another analysis port at the end. Both analysis ports can handle zero or more observers - hence operation of the UVC is ensured, even when there are no drivers or sequencers in circuit (as is the case for passive operation).

```
class my_slave_monitor extends uvm_monitor;
  ...
  virtual my_interface m_vif;
  my_transaction m_transaction;
  uvm_analysis_port #(my_transaction) transaction_aport; // full
  uvm_analysis_port #(my_transaction) request_aport; // partial
  ...
  function new(string name, uvm_component parent);
    super.new(name, parent);
    transaction_aport = new("transaction_aport", this);
    request_aport = new("request_aport", this);
  endfunction
  task run_phase();
    fork
      monitor_reset();
      monitor_bus();
      ...
    join
  endtask
  task monitor_bus();
    ...
    forever begin
      // decode bus signals in accordance with protocol
      ...
      request_aport.write(m_transaction); // publish request part
      ...
      // collect response
      transaction_aport.write(m_transaction); // publish full
    end
  endtask
  ...
endclass
```

Note, it is also possible to connect the monitor and sequencer using a blocking *put* port, to peek export. In this case the monitor must not construct the port when the enclosing agent is in passive mode (since this port requires a one-to-one connection) and the agent must not attempt the connection (note the sequencer is not present at all in passive mode).

3.3 Sequencer

The main difference in the reactive-slave sequencer compared to a normal sequencer is that it has an additional TLM analysis export that is connected to the request transactions published by the monitor. In addition there is a TLM FIFO which allows the sequences to stall while waiting for request transactions to be published by the monitor. Note that we do not have to code the write method for the analysis export since it is connected to a TLM FIFO. So the sequencer has the following additional code:

```
class my_slave_sequencer extends uvm_sequencer #(my_slave_seq_item);
  ...
  uvm_analysis_export #(my_transaction) request_export;
  uvm_tlm_analysis_fifo #(my_transaction) request_fifo;
  ...
  function new(string name, uvm_component parent);
    super.new(name, parent);
    request_fifo = new("request_fifo", this);
    request_export = new("request_export", this);
  endfunction
  function void connect_phase(...);
    super.connect_phase(phase);
    request_export.connect(request_fifo.analysis_export);
  endfunction
endclass
```

3.4 Driver

The driver is identical in all respects to a normal proactive driver. Specifically, it receives sequence items of the appropriate type that describe all aspects of the stimulus (field values and metadata values such as delays) and it generates a response in accordance with the protocol.

```
class my_slave_driver extends uvm_driver #(my_slave_seq_item);
  my_slave_seq_item m_item;
  ...
  task run_phase(...);
    init();
    forever begin
      // Get the next response item from sequencer
      seq_item_port.get_next_item(m_item);
      // Drive the response onto the interface
      drive_item(m_item);
      // Consume the response item
      seq_item_port.item_done();
    end
  endtask
  task drive_item(input my_slave_seq_item item);
    // drive response signals to DUT in accordance with protocol
    // based on response item fields, e.g. sync to clock edge,
    // wait for delay, drive signal <= item field
  endtask
endclass
```

3.5 Agent

The only additional responsibility in the agent is to connect the additional TLM analysis ports between the monitor and the sequencer.

```
class my_slave_agent extends uvm_agent;
  uvm_analysis_port #(my_transaction) slave_aport; // full transaction
  ...
  function void connect_phase(...);
    super.connect_phase(phase);
    ...
    monitor.transaction_aport.connect(slave_aport);
    if (is_active == UVM_ACTIVE) begin
      driver.seq_item_port.connect(sequencer.seq_item_export);
      monitor.request_aport.connect(sequencer.request_export);
      ...
    end
  endfunction
endclass
```

3.6 Test Environment

Several possibilities exist for starting the required slave sequence, but the most generic approach is to set the *default_sequence* for the sequencer's main run-time phase to be the desired default slave operation. (Note that the *default_sequence* for the *sequencer* has been deprecated in UVM-1.2, so you should use the *default_sequence* in the *main_phase* instead.) Setting the *default_sequence* is normally done using the *config_db* from the highest environment layer (i.e. the common environment that is used by all tests):

```
class my_test_env extends uvm_env;
  ...
  my_uvc uvc_env;
  ...
  function void build_phase(...);
    super.build_phase(phase);
    ...
    uvm_config_db #(uvm_object_wrapper)::set(
      this,
      "uvc_env.slave_agent.sequencer.main_phase",
      "default_sequence",
      my_slave_response_seq::type_id::get());
    ...
  endfunction
endclass
```

This is all that is required to initialize the reactive slave and start execution of the corresponding forever sequence. Whenever the DUT initiates a request on the master port, the slave sequence will generate a sequence item for the driver to use in providing a corresponding response.

Alternatively tests can start explicit sequences for the slave agents directly from the test component (provided the *default_sequence* for the *main_phase* is left as *null* in the configured environment) as shown below:

```
class my_test expends uvm_test;
  `uvm_component_utils(my_test)
  my_test_env test_env;
  ...
  task run_phase(...);
    ...
    fork
      slave_seq.start(test_env.uvc_env.slave_agent.sequencer);
    join_none
    ...
    // other stimulus
    ...
  endtask
endclass
```

Tests can of course select non-standard behaviour from the reactive slave by choosing an alternative sequence (and starting it directly or by overriding the *config_db* settings for the *main_phase default_sequence*), synchronising to slave traffic, interfering with storage values or error injection as discussed in subsequent sections.

4. Additional Features

4.1 Reactive Slave With Memory/Storage

Most applications require that the reactive slave models some kind of memory or storage. This is required since the DUT master will probably expect values written out to be retained by the destination slave in the real application. The storage component can be added to the agent as shown in Figure 10.

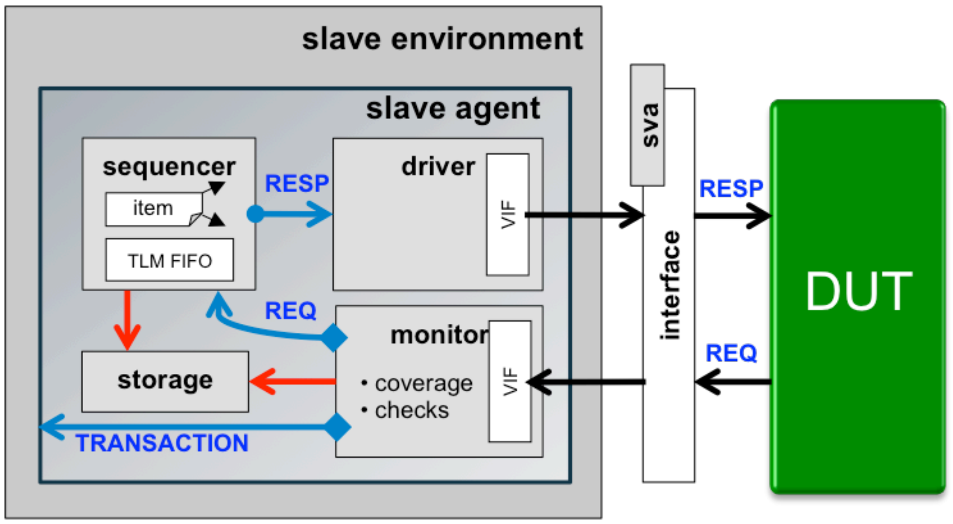

Figure 10 Reactive Slave With Storage

Figure 10 shows a slave **storage** component (typically a small or sparse memory array) that can be accessed by the monitor (for debug, even in passive mode) and by the stimulus via the sequencer (with a sequence or task-based API). Note that this slave memory is *not* part of the DUT and typically not part of the corresponding DUT register model - specifically it contains persistent data for another device in the system (the target slave) and is therefore modeled in isolation in order to provide realistic stimulus scenarios.

The storage API should provide the ability to perform the following operations:

- Write data to storage
- Read data from storage
- Initialize storage content

The following code shows a possible implementation of the storage API, which is provided by the storage component and accessible to any components or sequences which have a handle to this class:

```
class my_storage extends uvm_component;
  int mem[64];
  ...
  function void write(int addr, int value);
    mem[addr] = value;
  endfunction
  function int read(int addr);
    return(mem[addr]);
  endfunction
  function void init();
    bit [MY_DATA_WIDTH-1:0] rand_value;
    case (cfg.mem_init_type)
      MY_STORAGE_INIT_ZERO: begin
        for (int i = 0; i < size; i++) begin
          mem[i] = 0;
        end
      end
      MY_STORAGE_INIT_RANDOM: begin
        for (int i = 0; i < size; i++) begin
          void’(std::randomize(rand_value));
          mem[i] = rand_value;
        end
      end
    endcase
  endfunction
endclass
```

Note that for illustrative purposes we have specified the memory as a small array. For larger address spaces a sparse memory is preferable and can be implemented with an associative array,

The slave agent’s monitor is responsible for initializing the storage component upon detecting a reset, as well as for performing writes when it detects the DUT performing a write operation. It is important to note that these responsibilities are given to the monitor in order for the storage component to remain updated when the UVC is operating in passive mode.

```
class my_slave_monitor extends uvm_monitor;
  my_storage storage;
  ...
  task run_phase(...);
    ...
    fork
      forever begin
        // decode reset condition
        ...
        // reset condition detected
        storage.init();
      end
      forever begin
        // decode bus traffic
        ...
        // write transaction (from DUT) detected
        storage.write(addr, data);
        ...
      end
    join
  endtask
endclass
```

The forever sequence shown earlier (in Section 3. Sequence) should also be modified to read data from the storage component whenever it is required for a read response:

```
class my_slave_response_seq extends my_slave_base_seq;
  ...
  virtual task body();
    forever begin
      // wait for a transaction request (get is blocking)
      p_sequencer.request_fifo.get(m_request);
      // generate response based on observed request, e.g:
      case (m_request.m_direction)
        ...
        MY_DIRECTION_READ : begin
          `uvm_do_with(m_item,{
            m_item.m_resp_kind == MY_RESPONSE_DATA;
            m_item.m_data ==
              p_sequencer.storage.read(m_request.m_addr);
          })
        end
      endcase
    end
  endtask
endclass
```

In addition to the standard uses of the storage API described above, a test can choose to use it for operations like pre-filling the memory with data (e.g. based on the contents of an input file), or reading the memory contents out to a file for post-processing:

```
class my_test expends uvm_test;
  `uvm_component_utils(my_test)
  my_test_env test_env;
  ...
  task run_phase(...);
    ...
    meminit_seq.start(test_env.uvc_env.slave_agent.sequencer);
    ...
    // other stimulus
    ...
    memdump_seq.start(test_env.uvc_env.slave_agent.sequencer);
  endtask
endclass
```

4.2 Reactive Slave With Control Agent

So far we have described how the reactive slave generates responses only, which is typically enough for block-level environments. For practical high-level scenario generation in the presence of reactive slaves, we may need additional functionality to allow the test scenario stimulus to stall while waiting for the DUT to initiate a transaction on the corresponding slave interface, perhaps with a mechanism for validating any traffic on that interface or even interfering with the data in the reactive slave to provoke certain verification requirements (e.g. if the DUT writes a value out, we modify it on the fly before the DUT reads it back again). For this reason we typically add more functionality to the reactive slave environment to allow for operations such as:

- wait for a transaction from the DUT
- wait for a specific transaction kind and/or field value from the DUT
- wait for a specific transaction kind to a specific address from the DUT

A comprehensive API will enable the user to implement realistic constrained-random scenarios to validate overall operation and specific behavior of the DUT with respect to the reactive slave. In order to support this level of stimulus control using sequence-based stimulus, the reactive slave environment can be extended to perform these "tasks" inside a sequence API by adding a proactive scenario **control agent** to the UVC as shown in Figure 11.

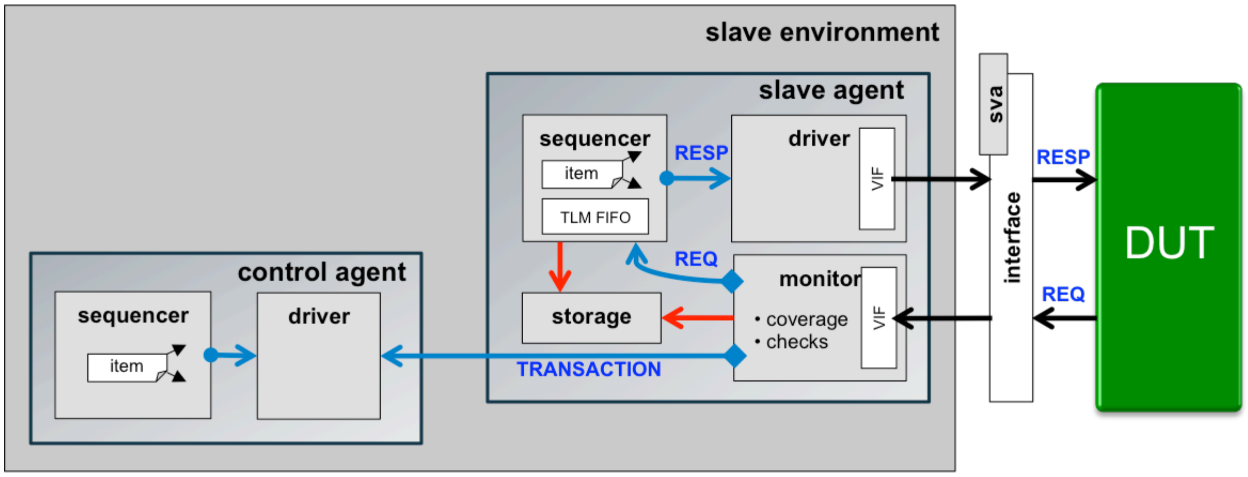

Figure 11 Reactive Slave With Control Agent

The control agent driver is split into two parts. The first gets sequence items from the sequencer which tell it what to wait for (e.g. any transaction, specific commands) and sits in a loop waiting for events and checking if the transaction content matches what is required. The second part of the driver just monitors the transactions from the analysis export and generates events when transactions are observed. The corresponding user sequence library for the control agent sequencer could contain sequences of the following type:

- wait_transaction_seq - wait for transaction from the DUT (read/write/any)
- wait_cmd_seq - wait for specific command (with control knobs for required fields)
- wait_addr_seq - wait for specific address (with control knobs for required fields)

We have used this architecture in a number of different UVCs for different clients and the flexibility and capability are always well appreciated by the testbench architects and test writers.

For the sake of brevity we do not present code examples of the sequence item and sequences that could be used to implement such a control agent; there is nothing novel about them, just standard UVM sequence items and sequences.

One possible implementation for the control agent shown in Figure 11 is for the driver to subscribe to transactions published by the reactive slave agent's monitor by connecting a UVM analysis export, and trigger required events from the write method of the analysis export implementation. The driver can then have a forever loop that processes these events based on sequence item requirements as shown below:

```
class my_control_driver extends uvm_driver#(my_control_seq_item);
  ...
  uvm_analysis_export #(my_transaction) transaction_aexport;
  my_transaction m_slave_transaction;
  event write_detected;
  event read_detected;
  ...
  task run_phase(uvm_phase phase);
    ...
    forever begin
      seq_item_port.get_next_item(m_item);
      drive_item(m_item);
      seq_item_port.item_done();
    end
  endtask
  task drive_item(input my_control_seq_item item);
    case (item.m_wait)
      WAIT_TRANSACTION: begin
        case (item.m_direction)
          WRITE: @write_detected;
          READ:  @read_detected;
          ANY:   @(write_detected or read_detected);
        endcase
      end
      WAIT_ADDRESS: begin
       case (item.m_direction)
          WRITE: @(write_detected iff
                     m_slave_transaction.m_addr == item.m_addr);
          READ:  @(read_detected iff
                     m_slave_transaction.m_addr == item.m_addr);
          ANY:   @(write_detected iff
                     m_slave_transaction.m_addr == item.m_addr,
                   read_detected iff
                     m_slave_transaction.m_addr == item.m_addr);
        endcase
      end
    endcase
  endtask
  function void write(my_transaction transaction);
    m_slave_transaction = my_transaction::type_id::create(...);
    m_slave_transaction.copy(transaction);
    case (transaction.m_direction)
      WRITE: ->write_detected;
      READ:  ->read_detected;
    endcase
  endfunction
endclass
```

A test component or sequence can make use of a scenario control sequence to wait for a specific event to occur, check the memory value, possibly choose to modify the value written to storage by the DUT, and continue the test from there as shown in the following code snippet:

```
class my_test_seq extends uvm_sequence;
  my_control_wait_addr_seq wait_addr_seq;
  ...
  // wait for the DUT to write to address 'h60
  `uvm_do_on_with(
     wait_addr_seq,
     test_env.uvc_env.control_agent.sequencer, {
       transaction == WRITE;
       addr        == 'h60;
     })
  // check the memory value provided by DUT
  m_data = test_env.uvc_env.slave_agent.storage.read('h60);
  if (m_data != `h1234)
    `uvm_error(...)
  // modify the memory value at address `h60
  test_env.uvc_env.slave_agent.storage.write('h60, 'h0000);
  // continue with other stimuli
  ...
endclass
```

4.3 Error Injection In Slave Responses

Error injection (i.e. stimulus sent to the DUT with deliberate errors introduced) for reactive slaves is much more complicated than it is for proactive masters since the test scenario does not know when the DUT will initiate the transaction. Note that simple distributions of slave response errors (e.g. randomly distributing a proportion of requests with an error response) are trivial to implement and can be enabled in the default sequence using an appropriate distribution constraint. However, this is typically not enough for higher-level test scenario generation, where for example we might want to inject an error on the first or last transfer in a burst, or on specific addresses.

One possible solution to accomplish this level of control from high-level test scenarios is explained in detail in [3] and summarized here. Counters are used to hold requests for error injection events and these are used by the sequences to provide the error responses on-demand whenever the DUT initiates a transaction request. These error injection counters (one for each type of error) can be conveniently stored in the configuration object of the UVC and modified via the control agent sequences as shown in the following code examples:

```
class my_slave_cfg extends uvm_object;
  int m_crc_error_count;
  int m_latency_error_count;
  ...
endclass
```

To allow testcase writers to increment the error injection counters, a sequence is added to the control agent for the UVC, with control knobs for each error type:

```
class my_control_increment_error_seq extends uvm_sequence;
  ...
  rand bit increment_crc_error;
  constraint default_crc_error_c {soft increment_crc_error == 0;}
  ...
  task body();
    if (increment_crc_error)
      p_sequencer.cfg.m_crc_error_count++;
    ...
  endtask
endclass
```

Additional error injection sequences can be added to the sequence library for the reactive slave (or existing sequences modified to be sensitive to error injection requests), which allow the slave-response mechanism to make use of the configuration's error counters as shown in the example below. If an error request counter is active in the configuration object when the response sequence item is generated, then error control flags are set in the sequence item in order to instruct the driver to inject the specified error into this current slave response. Hence the slave response containing errors is reacting to both scenario based error injection requests from the high-level test and protocol event timing controlled by the DUT.

```
class my_slave_response_error_seq extends uvm_sequence #(...);
  …
  virtual task body();
    forever begin
      // wait for a transaction request (get is blocking)
      p_sequencer.m_request_fifo.get(m_req);
      // generate response based on observed request and error flags
      `uvm_create(m_item)
      // randomize item
      if (!m_item.randomize() with {...})
        `uvm_error("RNDFLD", "randomization failed")
      // set the error flag after randomizing
      if (p_sequencer.cfg.m_crc_error_count > 0) begin
        m_item.m_inject_crc_error = 1;
        p_sequencer.cfg.m_crc_error_count--;
      end
      // execute the sequence item on the driver
      `uvm_send(m_item);
    end
  endtask
endclass
```

Finally, the user specifies this slave sequence as the default sequence for the test in which error injection is desired and calls the control sequence when required to set the counters:

```
class my_err_test extends uvm_test;
  ...
  my_uvc uvc_env;
  ...
  function void build_phase(...);
    super.build_phase(phase);
    ...
    uvm_config_db #(uvm_object_wrapper)::set(
      this,
      "uvc_env.slave_agent.sequencer.main_phase",
      "default_sequence",
      my_slave_response_error_seq::type_id::get());
    ...
  endfunction
  task run_phase(...);
```

```
    my_control_wait_addr_seq wait_addr_seq;
    my_control_increment_error_seq error_seq;
    ...
    wait_addr_seq.start(test_env.uvc_env.slave_agent.sequencer);
    error_seq.start(test_env.uvc_env.slave_agent.sequencer);
    ...
  endtask
endclass
```

4.4 Bus Topology Considerations

All of the analysis in this paper so far has focused on operation of a single slave agent, typically in the context of response generation for a DUT master where there is a simple point-to-point connection between the master and the slave. However, none of the arguments presented are limited to such simple UVC environments and can easily be extended to cover much more complex topologies. In fact the extension to complex topologies has no effect on the internal architecture of the slave agents, and only minimal effect on how we connect them into a bigger cluster.

In bus protocols where there are separate slave select signals (e.g. AHB, SPI) or allocated slave addresses (e.g. I2C) we would strongly recommend having one agent per slave to provide the maximum flexibility and control when creating response stimulus scenarios. With such topologies we would also recommend having a passive master agent to monitor the resolved bus traffic at the DUT interface. Only one scenario control agent for the reactive slaves is required, since this can subscribe to transactions published by each slave agent (i.e. we extend the control driver capability to subscribe to multiple transaction streams by adding analysis exports). A UVC environment supporting multiple reactive slave agents for a single-master protocol is shown in Figure 12.

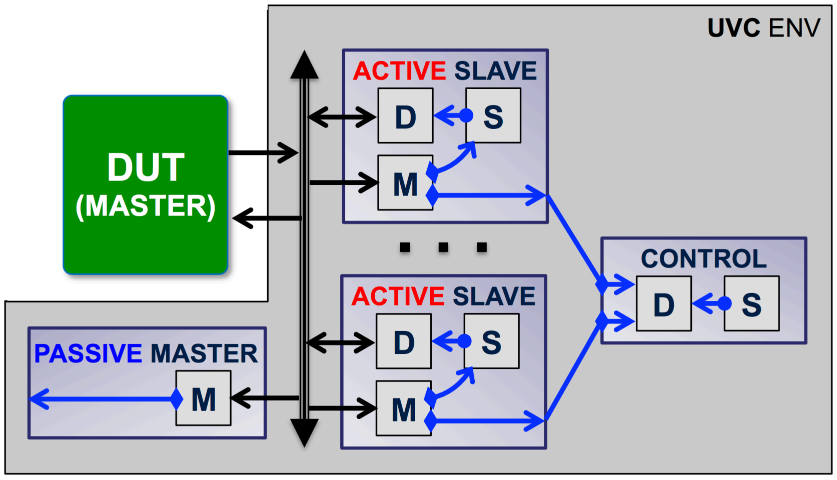

Figure 12 UVC Topology for Multiple Slaves with Single Master

For multi-master bus protocols, the requirements for reactive slaves do not change, but there are additional demands on the master side which are covered here for completeness. Specifically in addition to the passive master agent to monitor (check & cover) traffic at the DUT master port, we require one or more alternate active master agents in order to compete with the DUT for ownership of the bus and a separate bus monitor to publish all resolved traffic on the bus irrespective of what each end-point thinks is happening. Figure 13 shows such a multi-master bus topology for a UVC environment.

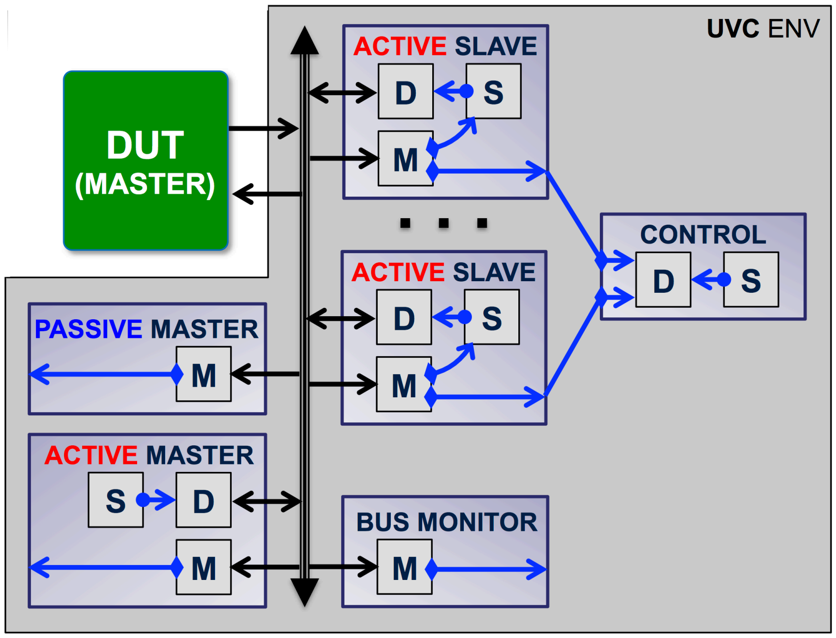

Figure 13 UVC Topology for Multiple Slaves with Multiple Masters

In summary, we would recommend setting up your UVC structure to cover all anticipated bus topologies from single agent (for point-to-point) to full bus topology required by protocol (i.e. single or multiple masters). This means the UVC can be built with:

- 0 or more agents of each type (master/slave)
- individual control over active/passive setting of each agent
- multi-master protocols should provide an independent bus monitor

5. Conclusions

This paper presented a discussion on the differences between proactive master and reactive slave verification components and demonstrated an architecture that has been used by Verilab engineers for many different projects and various interface protocols. The structural changes for the slave agent are minimal in order to achieve the basic constrained-random sequence-based response generation as a reaction to DUT traffic, and the architecture is easily extended to provide scenario-based synchronization to specific DUT traffic, test-controlled observation and manipulation of slave storage or memory, as well as controlling error injection for responses from high-level test scenarios. The detailed content presented here should enable intermediate level verification engineers to get consistent results producing reactive slave UVCs in UVM, and provide a reference implementation for consideration by advanced users.

6. References

[1] Accellera, UVM (*Universal Verification Methodology*), www.accellera.org

[2] Mentor Graphics, *UVM Sequences in Depth*, Web Seminar, www.mentor.com

[3] J. Montesano, M. Litterick, *UVM Sequence Item Based Error Injection*, SNUG 2012

[4] M. Litterick, J. Montesano, T. Reddy, *Mastering Reactive Slaves in UVM*, SNUG 2015