Stimulating Scenarios in the OVM and VMM

JL Gray
Verilab, Inc.
Austin, TX
jl.gray@verilab.com

Scott Roland
Verilab, GmbH
Munich, Germany
scott.roland@verilab.com

ABSTRACT

The Open Verification Methodology (OVM) and Verification Methodology Manual (VMM) libraries used to augment the capabilities of the SystemVerilog language introduce advanced stimulus generation capabilities suitable for designing large testbenches and verification IP in the form of sequences and scenarios. However, many verification teams struggle to fully utilize these techniques, and end up with testbenches that either only support directed tests, or support randomization while being difficult to maintain and enhance. In this paper, advanced stimulus generation concepts, architecture, and motivation will be described. Tips for a successful stimulus generation implementation will be provided, and solutions from the VMM and OVM libraries will be compared and contrasted.

Specifically, we will cover the basic components of a stimulus generation solution in the OVM and VMM, including both standard and multi-channel stimulus, and deal with the issue of resource allocation via the `grab/ungrab` API. Next, we will review techniques required for building stimulus suitable for modeling both active (master) and reactive (slave) testbench components. We will close with a discussion of push vs. pull models of driver development.

Categories and Subject Descriptors

B 6.3 **[Hardware]:** Logic Design – *simulation, verification.*

D.3.3 **[Programming Languages]:** Language Constructs and Features – *frameworks, patterns.*

D.2.2.2 **[Software]:** Design Tools and Techniques – *software libraries, object oriented design methods.*

General Terms

Algorithms, Languages, Theory, Verification.

Keywords

SystemVerilog, OVM, VMM, Scenarios, Sequences, Stimulus, Verification

1 INTRODUCTION

Before the advent of EDA tools, early chip layouts were inspected by hand for bugs. In fact, in the early 1960s, it took 16-18 weeks to create the layout for a module for the room-sized computers of the day, and an additional 10 weeks to debug the completed board. [1]. Therefore, it was critical to find bugs as early as possible in the design process. Later, directed test software simulation techniques were applied to increase the speed at which bugs could be found. However, over time chips have become too large and complex for such approaches. Traditional test writing styles, while still valid in many instances, have not scaled well to meet the challenges of new categories of devices built to take advantage of an ever increasing number of transistors on a single chip.

The Open Verification Methodology (OVM) and Verification Methodology Manual (VMM) libraries used to augment the capabilities of the SystemVerilog language provide advanced stimulus generation capabilities suitable for designing large testbenches and verification IP in the form of sequences and scenarios. However, many verification teams struggle to fully utilize these techniques, and end up with testbenches that either only support directed tests, or support randomization while being difficult to maintain and enhance. In this paper, advanced stimulus generation concepts, architecture, and motivation will be described. Tips for a successful stimulus generation implementation will be provided, and solutions from the VMM and OVM libraries will be compared and contrasted.

Specifically, we will cover the basic components of a stimulus generation solution in the OVM and VMM, including both standard and multi-channel stimulus, and deal with the issue of resource allocation via the `grab/ungrab` API. Next, we will review techniques required for building stimulus suitable for modeling both active (master) and reactive (slave) testbench components. We will close with a discussion of push vs. pull models of driver development.

2 CONCEPTS

Advanced stimulus generation as implemented in the OVM and VMM is composed of five major data structure types. Each library uses a different set of terms to describe these components.

- transaction
- driver
- sequence (collection of transactions)
- sequencer (contains library of sequences, drives sequences)
- virtual sequences

To simplify matters, the terminology from the OVM will be used throughout the paper to components written in either methodology. However, the VMM terminology will be used as needed when both libraries are discussed together. In this section, each of the five components of stimulus generation will be introduced, along with code examples showing how to use each one.

Examples in this section will be derived from the basic testbench architectures shown in Figure 1 and Figure 2.

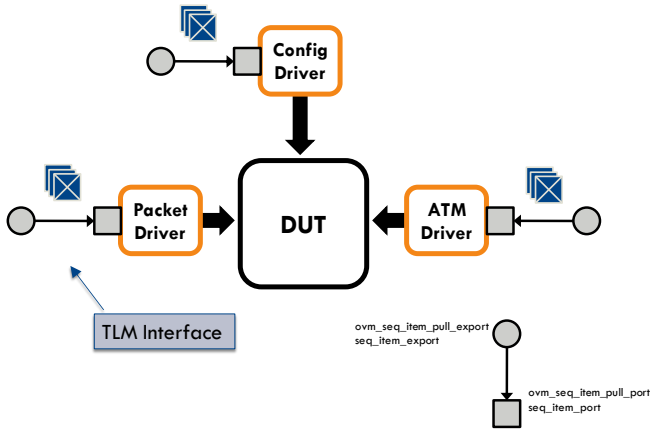

Figure 1: OVM Testbench Basics

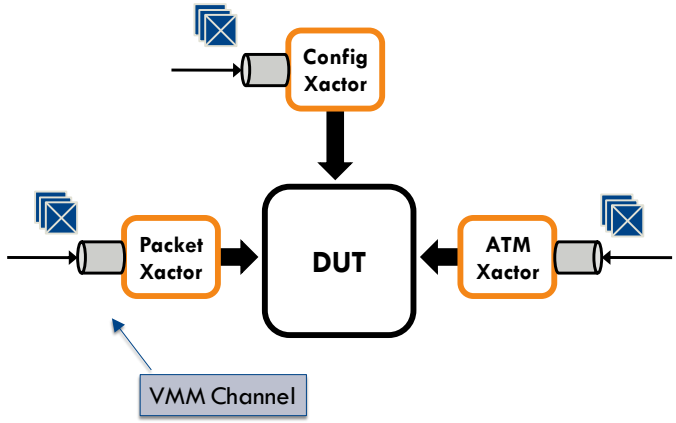

Figure 2: VMM Testbench Basics

2.1 Terminology

The OVM and VMM support many of the same stimulus generation concepts but use different terminology to describe them. Here is a mapping of the terms that will be used throughout this paper.

Table 1: Terminology Mapping Between the OVM and VMM

OVM	VMM	Definition[1]
ovm_sequence_item	vmm_data	Transaction
Sequence	Scenario	An object that defines a set of transactions to be executed and/or controls the execution of other scenarios on a single interface
Virtual Sequence	Multi-stream Scenario	Similar to a sequence, but can control transactions and sequences from multiple interfaces.
Sequencer	Scenario Generator / Multi-stream scenario generator	A verification component that provides transactions to another component.
Driver	Transactor	A component responsible for executing or otherwise processing **transaction**s, usually interacting with the device under test (DUT) to do so.

[1] Definitions were taken from [2] where applicable.

Throughout the document, OVM source code will be displayed in a box with a solid border.

```
class packet extends ovm_sequence_item;
```

VMM code is displayed with a dashed border.

```
class packet extends vmm_data;
```

2.2 Transaction

Before the advent of constrained random testing, generating "stimulus" involved writing directed tests to wiggle pins of a Device Under Test (DUT) or to fill a memory with machine code bytes. However, it soon became apparent that it was more convenient to think at higher levels of abstraction. For example, a test writer might want to execute a register read or write transaction. The transaction encapsulates all of the information needed to wiggle the pins required to execute the command and relieves the test writer of the tedium of thinking about the low level details of particular operations.

Basically, a transaction is "a class instance that encapsulates information used to communicate between two or more components." [2] Transactions are the basic building blocks of any verification environment. Transactions can be used to describe a variety of common testbench data types, such as:

- Ethernet packets
- CPU instructions
- ATM cells
- OCP transactions
- Registers

In the past, testbench data may have been passed as parameters to function calls. For example, someone attempting to write to a register may have simply said:

```
write(0xA0, 0xFF001FF0);
```

This Verilog task call writes the value 0xFF001FF0 to the register at address 0xA0. Similarly, to read register data, a user may have called a task similar to the following:

```
read(0xA0, result);
```

`result` would end up populated with a value such as 0xF0F0F01. Experienced verification engineers will recognize that this information is only marginally useful. A typical 32-bit register is made up of one or more fields of varying bit-widths. Optimally,

these fields could be referenced by name without knowledge of the underlying register structure or widths. In Verilog, I could have created a struct to represent these fields. Using a struct solves one problem, but it does not allow a user to create customized methods such as those described in Table 2. Specialized transaction-specific base classes help automate the process of incorporating specialized functionality into data structures.

Support for transactions is included in all modern verification methodology libraries, including the OVM and VMM. When building a verification environment, it is critical to utilize the built-in transaction base classes to ensure compatibility with the stimulus generation capabilities inherent in the libraries. These classes also free the user from having to deal with the implementation of many commonly used operations:

Table 2: Common operations for transactions

ovm_sequence_item	vmm_data
copy	copy
pack	pack
unpack	unpack
compare	compare
print	psdisplay
record	record
--	save
--	load

Transactions are defined similarly in both the OVM and VMM. OVM transactions are based on the `ovm_sequence_item` base class. Customization macros are used to enable users to copy, compare, print, etc. without having to implement these methods manually.

```
class packet extends ovm_sequence_item;
   rand bit [47:0] src;
   rand bit [47:0] dst;
   rand bit [15:0] type_len;
   rand unsigned int delay;
   constraint short_delay {delay < 100; };
   `ovm_object_utils_begin(packet)
     `ovm_field_int(src, OVM_ALL_ON)
     `ovm_field_int(dst, OVM_ALL_ON)
     `ovm_field_int(type_len, OVM_ALL_ON)
   `ovm_object_utils_end(packet)
   ...
endclass : packet
```

The same transaction can be defined in VMM as shown below. This code is similar (but not the same) between the libraries.

```
class packet extends vmm_data;
   rand bit [47:0] src;
   rand bit [47:0] dst;
   rand bit [15:0] type_len;
   ...
   rand unsigned int delay;
   constraint short_delay { delay < 100; };
   ...
   `vmm_data_member_begin(packet)
     `vmm_data_member_scalar(src, DO_ALL);
     `vmm_data_member_scalar(dst, DO_ALL);
     `vmm_data_member_scalar(type_len, DO_ALL);
   `vmm_data_member_end(packet)
endclass : packet
```

Once transactions are available in an environment, it is possible to create directed or fully random tests by creating a transaction and passing it to a component (usually a `vmm_xactor` or an `ovm_component`) which will drive the transaction on the physical interface to the Device Under Test (DUT). Within a directed test one could also create a well-defined series of transactions (like opening a TCP connection or describing a for-loop in an assembly program). These types of series can be extremely valuable if reused in other tests in the user's environment.

Modern methodologies such as the VMM and OVM allow reuse of series of transactions via the use of sequences. Before discussing sequences, it is important to understand how the drivers shown in Figure 1 work with the other components of the stimulus generation solution.

2.3 Driver

As described in Table 1, a driver is a component responsible for executing or otherwise processing transactions, usually interacting with the device under test (DUT) to do so. While this definition applies to drivers created in both the OVM and VMM, there are some implementation differences. Stimulus in the VMM is generated using "push" mode. The "push" style of stimulus generation requires a driver that can accept transactions passed to it via one of its standard interfaces. In the VMM, transactions would be passed in via either a `vmm_channel` or a Transaction Level Modeling (TLM) blocking or non-blocking transport.

When an OVM driver is operated in push mode, transactions are passed in via a TLM export. However, push mode is not the standard mechanism for generating stimulus in the OVM. Instead, "pull" mode is used. Both pull and push modes are described more thoroughly in section 3.2 below.

Figure 3 provides a step-by-step overview of the general flow of a pull-mode driver.

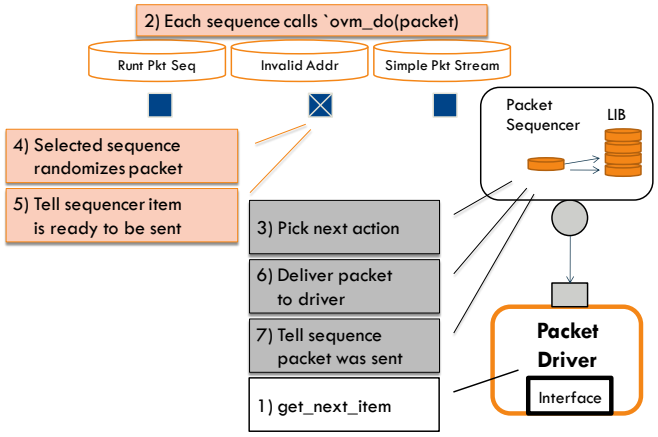

Figure 3: OVM Pull Mode

Unlike push-mode stimulus in the VMM and OVM, the `run()` task of an OVM driver written to be used in pull mode must be implemented as shown below in order to function correctly.

```
class packet_driver extends ovm_driver #(packet);
  function new (string name,
                ovm_component parent);
```

```
    super.new(name, parent);
  endfunction : new

  `ovm_component_utils(packet_driver)

  task run();
    packet item;
    forever begin
      @(...);
      // User must implement logic to get next item
      // from sequencer as shown.
      seq_item_port.get_next_item(item);
      ovm_report_info(get_type_name(),
         "driving packet");
      seq_item_port.item_done();
    end
  endtask : run

endclass : pkt_driver
```

With the appropriate drivers in place, it is possible to start constructing sequences, sequencers, and virtual sequences.

2.4 Sequence

Once testbench creators started thinking in terms of individual transactions, the next logical progression was to imagine what to do if you were to have a collection of transactions. For example, what if you wanted to send a particular stream of read or write register commands to program the DUT? What if you want to open a TCP/IP connection using a stream of Ethernet packets? Higher level collections of transactions can be modeled using the concept of sequences.

Sequences define useful streams of transactions. According to Accellera, a sequence is an "object that procedurally defines a set of transactions to be executed and/or controls the execution of other sequences."[2] For example, imagine trying to test a network device under the scenario where a user needs to open a TCP/IP connection. The desired stream of packets might look something like this:

- Send a short packet to IP address 192.168.0.1
- Send a long packet to IP address 192.168.0.1

The stream could be captured as a directed test, but ideally we'd like to run this stream intermixed with a variety of other randomly selected streams of packets. Sequences (scenarios in the VMM) can be used to accomplish this goal. To code a sequence in the OVM, simply extend the `ovm_sequence` class:

```
class pkt_seq extends ovm_sequence #(packet);
  rand reg [47:0] addr;

  `ovm_sequence_utils_begin(pkt_seq, pkt_sqr)
    `ovm_field_int(addr, OVM_ALL_ON)
  `ovm_sequence_utils_end

  function new(string name="x_seq");
    super.new(name);
  endfunction

  virtual task body();
    // Send packet 1
    `ovm_do_with( ... )
    // Send packet 2
    `ovm_do_with( ... )
  endtask : body

endclass : my_packet_seq
```

In the example above, the `pkt_seq` sequence is derived from an `ovm_sequence` specialized based on the `packet` data type.

In the VMM, create a VMM scenario. Starting with the VMM 1.2, use the parameterized VMM scenario base class as the basis for all user-generated scenarios.

```
class packet_scenario extends vmm_ss_scenario #(packet);
   ...
endclass: packet_scenario
```

Now, create a user-defined scenario by implementing the `apply()` method of `my_packet_scenario`. `my_packet_scenario` is extended from `packet_scenario`:

```
class my_packet_scenario extends
                              packet_scenario;
  ...
  virtual task apply(...);
     // Ideally, use VMM factory instead
     // of direct instantiation...
     packet pkt1 = new;
     packet pkt2 = new;

     // Send packet 1.
     pkt1.randomize();
     channel.put(pkt1)

     // Send packet 2
     pkt2.randomize();
     channel.put(pkt2)

     ...
     channel.put( ... )
  endtask : apply
  ...
endclass : my_packet_scenario
```

Once a single sequence has been created, it makes sense to create a collection of sequences.

Once a collection of sequences exists it quickly becomes apparent that a testbench component must be created in order to manage the collection of sequences. This component is called a sequencer.

2.5 Sequencer

The sequencer has several functions within a testbench. First, it manages the collection of sequences relevant to the particular sequencer type and instance. Individual sequencers can be customized by test writers and environment integrators to behave as needed for the application at hand.

Second, the sequencer manages the interaction between sequences and the underlying driver. The sequencers must also provide a way to control which sequences will be selected. Both the VMM and OVM make it possible to create a set of weighted constraints that can be applied to this process. As shown in Figure 4, the OVM uses the concept of a default sequence. The default sequence contains the

constraints and procedural code required to choose the next sequence to be selected.

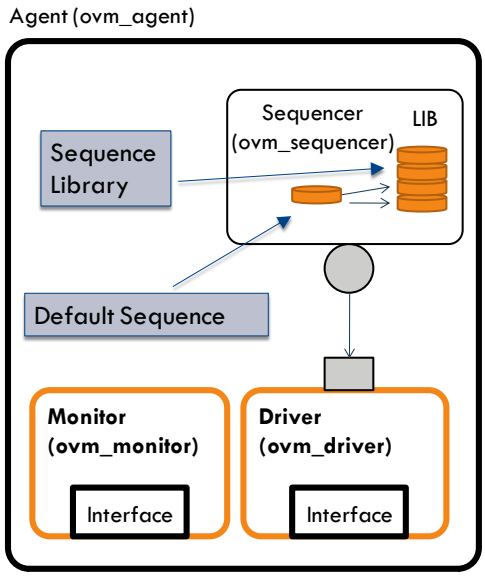

Figure 4: Default Sequence in an OVM Sequencer

As shown in Figure 5 in the VMM a voter is used to select the next scenario. The voter could be configured to select a single scenario that could then be used as a "default" scenario if behavior similar to the OVM is desired.

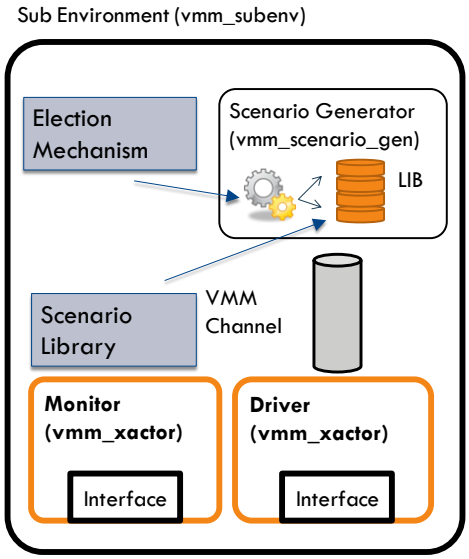

Figure 5: Voter in a VMM Scenario Generator

The sequencer serves other purposes as well. It serves as an anchor to hold the customized set of sequences and underlying configuration parameters for sequences in the sequence library. Additionally, in OVM the sequencer is also a resource that can be grabbed by other testbench components (sequences or virtual sequences). In the VMM the channel serves as the lockable resource, but the scenario generator is responsible for maintaining a registry of scenarios that can be run in the context of the scenario generator.

An OVM sequencer can be created as shown.

```
class pkt_sqr extends ovm_sequencer #(packet);

  function new (string name="pkt_sqr",
                ovm_component parent);
    super.new(name, parent);
    `ovm_update_sequence_lib_and_item(packet)
  endfunction : new

  `ovm_sequencer_utils(pkt_sqr)

endclass : x_sqr
```

The VMM scenario generator and associated channel classes are generated using VMM macros.

```
// Instantiate the channel & scenario generator
// using transaction "packet".
`vmm_channel(packet)
`vmm_scenario_gen(packet, "packet class")
```

2.6 Virtual Sequences

Let's review what has been covered so far. First, the concept of a transaction was introduced to save us from having to deal with the underlying DUT behavior. Next, we created a collection of sequences and the sequencer required to encapsulate important system use cases/behaviors and allow us to randomly select from these during a simulation. However, something is still missing. What happens if users want to coordinate what happens on several sequencers together? For example:

- Configure the DUT (configuration sequencer)
- Send an ATM cell (ATM sequencer)
- Open a TCP/IP connection (packet sequencer)

To do this, the concept of the virtual sequencer is needed. The virtual sequences executed by the virtual sequencer are capable of executing sequences from any other sequencer or virtual sequencer in the verification environment. Virtual sequences provide the user with fine-grained control of testbench activities and allow for coordination between multiple disparate interfaces.

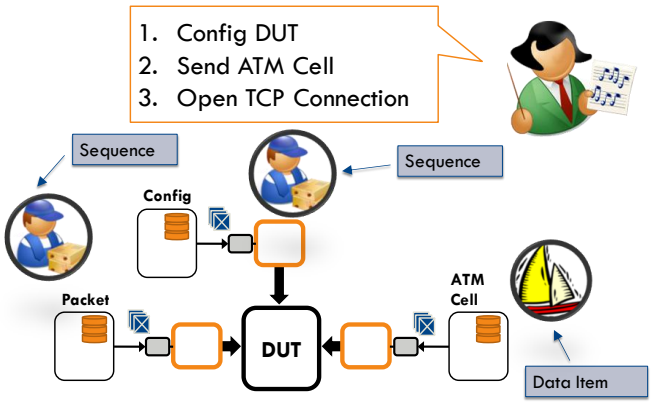

Figure 6: Virtual Sequences Overview [3]

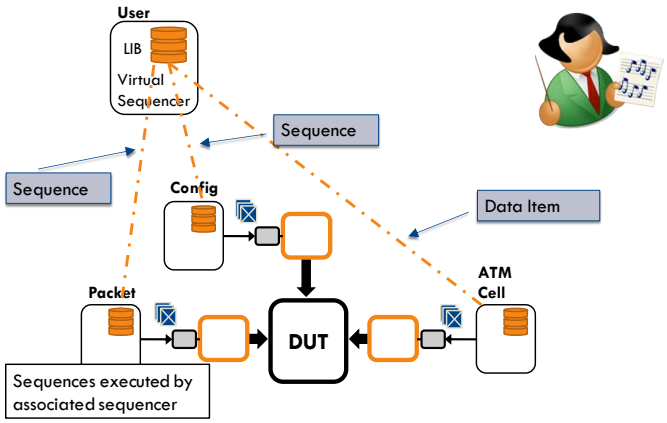

Figure 7: OVM Virtual Sequences Controlling Sequences [3]

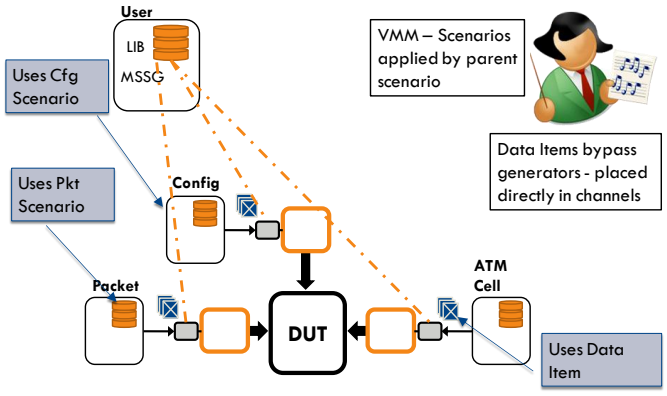

Figure 8: VMM Multi-Stream Scenarios Controlling Scenarios

Virtual sequences in the OVM are created using sequencers and sequences that have no associated sequence item. Otherwise, they are created in the same fashion as regular sequences.

In the VMM, virtual sequences are known as multi-stream scenarios (MSS). Sequencers are known as multi-stream scenario generators (MSSG). Unlike virtual sequencers in the OVM, MSSGs are not created using the same data structures as scenario generators in the VMM. MSSGs are described in more detail in [4] and [5].

2.7 Grab/Ungrab

In a VMM testbench we use a channel to pass stimulus information to a transactor. In an OVM testbench, stimulus is passed to drivers via a sequencer. In each case, data could be sent to the transactor/sequencer from different streams (threads). Data sent from multiple streams could end up being interleaved, but this is not always desirable. For example, an algorithm might call for an uninterrupted sequence of commands. It may also be desirable to lock multiple resources (channels in the VMM, sequencers in the OVM) for a given set of operations such that other stimulus streams do not interfere with the flow of data. This could occur during an arbitration scenario where the activities on multiple interfaces are coordinated to create specific traffic timing patterns for a period of time.[4]

Resource sharing in both the OVM and VMM is handled via the concepts of `grab` and `ungrab`. Grabbing a resource means that the grabber has exclusive rights to send transactions through it. Other testbench components attempting to access the resource will be blocked until the grabber "ungrabs" the resource using the `ungrab` command.

The problem of resource sharing cannot be resolved simply by having individual sequences grab channels for exclusive use. Hierarchical stimulus, where a sequence or virtual sequence can have any number of child sequences makes it necessary to have a scheme that allows the children to obtain a grabbed resource for use from their parent or another ancestor sequence. Both the OVM and VMM use similar algorithms to determine who may `grab` or `ungrab` any given sequence or channel. For more information about how this is handled in the VMM, see [4] and [5]. For information on how this is handled in the OVM, see [6] and [7].

3 USE CASES

In section 2 we covered the basic concepts required to implement stimulus in the OVM and VMM. Next, we will demonstrate the concepts in a real system context where both master and slave drivers are present. Slaves are especially interesting because of the feedback path from the driver to the sequence. The examples in this section show how to implement each component using both the OVM and VMM.

3.1 Open Core Protocol (OCP)

The following examples are based on a production Verification IP for the Open Core Protocol™ standard [8]. The goal was to take a real piece of VIP with code that has been used in production and distill useful examples from it, rather than creating simply theoretical code. The reader should be able to understand the examples even without any particular knowledge of OCP.

The OCP defines a point-to-point interface between two components. For each OCP instance there is one master that initiates request transactions and one slave that responds to the transactions. If two components wish to communicate in a peer-to-peer fashion, then they need to be connected with two OCP instances, with each

component being a master on one instance and a slave on the other. It is possible to connect multiple components together in a variety of arrangements, using as many OCP instances as desired, each instance having a single master and slave.

OCP defines a set of dataflow signals that are used for the primary communication between the components via read and write transfers. OCP also defines a set of sideband signals that are used for control information. The dataflow and sideband signals are allowed to change asynchronously with respect to each other.

3.2 Push vs. Pull

As mentioned in section 2.3, there are two variants of sequencer-driver interaction: Push and Pull. Both strategies can be used to create complex stimulus appropriate for any environment.

In the case of a "push" sequence, the sequence initiates the action by putting[2] a transaction into the sequencer, which then gives the transaction to the driver. For a "pull" sequence, the driver initiates the action by getting a transaction from the sequencer, which will request the transaction from the sequence.

It is often desirable to determine the contents of a transaction based on conditions up to and until the driver is ready to use it. Pull sequences make it easier to adapt a transaction to the system at the time it will be used. For example, if a transaction were being sent to indicate the current fill level of a FIFO or if a sequence were trying to meet a certain coverage goal. The delaying of generation of transactions until they are ready to be used is sometimes called "late randomization" because the randomization occurs as late as possible.

Drivers are often required to consider interface arbitration rules before starting the next transaction. An OCP master driver cannot start driving a transaction on the request phase signals if the signals are currently in use by a different transaction. When using a pull driver, an OCP master would not get the next transaction from the sequence until it was ready to be used.

Since a push sequence initiates the action that eventually happens in the driver, it is possible, and common, for the driver to delay using an already created transaction until a later time. However, it is possible to avoid this by having the push mode sequence communicate with the push driver and delay generation of the transactions until when they are ready to be used. This allows a push sequence to also achieve late randomization. An OVM example of this will be shown in section 3.4.1, while a VMM example of this will be shown in section 3.4.2.

3.3 OCP Sideband Driver

The OCP sideband signals include signals dedicated to conveying interrupt and error information. Normally, a component experiences a condition and then drives the appropriate sideband signal, such as the master interrupt.

Either push or pull mode could be used since the source of the information decides when the transaction should be sent. The sideband signal driver does not need to worry about arbitration or protocol semantics to decide when it can drive the signals.

3.3.1 Push Sideband Driver (OVM)

In the OVM, a sideband sequence is created by extending from the `ovm_sequence` class. This class has a parameterized member variable (`req`) that can be used for storing each transaction. The sequence `ocp_minterrupt_sideband_seq` below issues a single sideband transaction, which indicates that the master interrupt (`minterrupt`) signal should be driven.

```
class ocp_minterrupt_sideband_seq extends
                              ovm_sequence#(ocp_sideband);
  `ovm_sequence_utils(ocp_minterrupt_sideband_seq,
                      ocp_sideband_sequencer)
  function new(string name="…");
    super.new(name);
  endfunction

  rand bit minterrupt_value;
  virtual task body();
    `ovm_create(req)
    req.drive_minterrupt = 1;
    `ovm_rand_send_with(req,
                    {req.minterrupt==minterrupt_value;})
  endtask : body
endclass : ocp_minterrupt_sideband_seq
```

The `` `ovm_rand_send_with `` macro will randomize the transaction and send it to the OVM sequencer. The sequencer will then push the transaction to the driver with a call to the `put()` task.

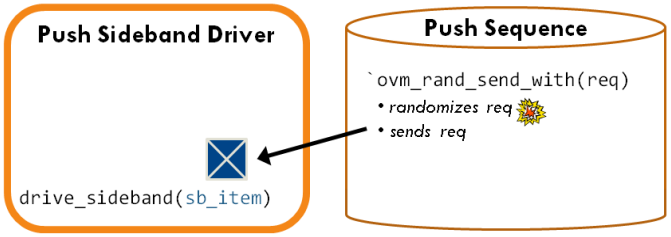

Figure 9: OVM Push Driver (OCP Sideband)

A simplified version of the OVM-based push driver is shown below. The transaction is driven on the OCP interface once received by the driver.

```
class ocp_sideband_driver extends
       ovm_push_driver#(ocp_sideband);
  …
  task put(ocb_sideband sb_item);
    drive_sideband(sb_item);
  endtask : put
endclass : ocp_sb_driver
```

It would also be possible to use a pull driver with pull sequences for controlling the OCP sideband signals. When using OVM, *pull drivers are recommended and are a good default choice*.

3.3.2 Push Sideband Driver (VMM)

In the VMM, a sideband sequence is created by extending the `vmm_ss_scenario` class. The sequence `ocp_minterrupt_sideband_seq` below issues a single sideband

[2] In the OVM, sequence items are "put" into a TLM interface to the sequencer. In the VMM, data items are placed into the channel.

transaction, which indicates that the master interrupt (minterrupt) signal should be driven.

```
class ocp_minterrupt_sideband_seq extends
                          vmm_ss_scenario#(ocp_sideband);
  rand bit minterrupt_value;
  virtual task apply(…);
    ocp_sideband sb_item = new();
    sb_item.drive_minterrupt = 1;
    if (!sb_item.randomize() with
              {sb_item.minterrupt==minterrupt_value;}) …
    channel.put(sb_item);
  endtask: apply
endclass : ocp_minterrupt_sideband_seq
```

The channel.put() call will push the transaction through the channel to the driver.

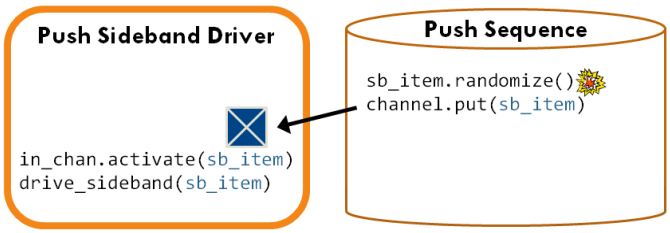

Figure 10: VMM Push Driver (OCP Sideband)

A simplified version of the VMM-based push driver is shown below. When the driver receives the transaction from the input channel, it drives it on the OCP interface. The driver also does some standard VMM management of the channel, but these actions are not important for understanding the execution flow of the driver.

```
class ocp_sideband_driver extends vmm_xactor;
  virtual task main();
    fork
      super.main();
      forever begin
        ocp_sideband sideband_item;
        wait_if_stopped_or_empty(in_chan);
        in_chan.activate(sideband_item);
        void'(in_chan.start());
        drive_sideband(sideband_item);
        void'(in_chan.complete());
        void'(m_in_chan.remove());
      end
    join
  endtask : main
endclass : ocp_sideband_driver
```

3.4 OCP Dataflow Master

The OCP dataflow signals are used for the read and write transfers between components. These signals must follow several protocol rules which specify when a new transfer can begin. A simple push driver could be used for these signals, but that would mean that the sequence could randomize the transactions before they were actually used by the driver.

The next set of examples show OCP dataflow master drivers. The drivers initiate requests transactions on the interface. They are architected to ensure that the request transactions are only randomized at the time they will be used.

3.4.1 Pull Dataflow Master (OVM)

The OVM implementation of the OCP master uses the recommended OVM pull driver. Since this is a pull driver, it will determine when to initiate the get of the next transaction. Only when the OCP interface is ready to accept the next request transaction will the master attempt to do a get_next_item() from the sequencer.

```
class ocp_master extends ovm_driver#(ocp_txn);
  virtual task drive_interface();
    forever begin
      @(...);
      if (interface_is_available() &&
          seq_item_port.has_do_available())) begin
        seq_item_port.get_next_item(req);
        drive_request(req);
        seq_item_port.item_done();
      end
    end
  endtask : drive_interface
endclass : ocp_master
```

The sequencer does not have to provide a new transaction every time the master asks for one. The master first checks with the sequencer to see if it is ready to provide a new transaction. The sequencer might be waiting for an external event, such as a FIFO reaching a certain threshold, before providing the next transaction to master. If the sequencer is not ready to provide a new transaction this cycle, then the master will leave the interface idle and check again the next cycle. This allows the sequence to have control over when things happen, even though it is in pull mode and does not initiate the activity.

A simplified view of the flow of control is shown in Figure 11. For a higher level view of the flow, including the sequencer, refer to Figure 3.

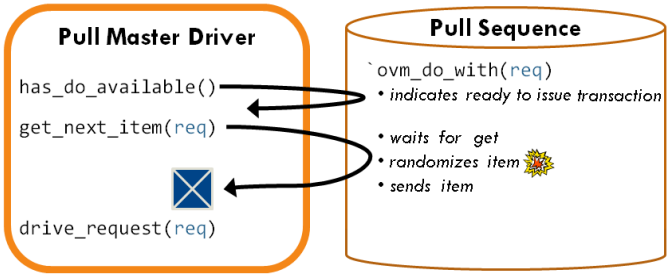

Figure 11: OVM Pull Master (OCP Dataflow)

Below is the example of a simple pull sequence that issues a single read transaction, but only after the FIFO has enough space for the read it will be sending.

```
class ocp_simple_read_request_seq extends
                                  ovm_sequence #(ocp_txn);
  virtual task body();
    wait_for_fifo_space();
    `ovm_do_with(req, {req.cmd ==RD; ... } )
    get_response(rsp);
  endtask : body
endclass : ocp_simple_read_request_seq
```

The `ovm_do_with macro hides a few actions and is basically equivalent to the following:

```
  `ovm_create(req)
  start_item(req); // blocking
  if(!req.randomize() with {req.cmd ==RD; ...}) ...
  finish_item(req);
```

The `start_item()` call is blocking. First it indicates that the sequence is ready to provide the next transaction, and then it waits until the sequencer actually requests it. This ensures that the transaction is only randomized at the time it is needed.

3.4.2 "Pull" Dataflow Master (VMM with Notifier)

The VMM implementation of the OCP master uses a VMM push driver to emulate pull mode. To avoid the push sequence randomizing a transaction before the driver is ready to use it, a VMM notifier is connected between the OCP master and sequence. The `READY` indicator will be set when the master is ready for a transaction from the sequence. This simple approach does not handle the case where multiple sequences are trying to drive transactions to the master driver.

```
class pull_indications extends vmm_notify;
  typedef enum {READY} indications_e;
  ...
endclass: pull_indications
```

The example sequence will issue read requests until stopped. The sequence first waits on the `READY` indicator before randomizing the transaction and putting it in the channel to send to the master.

```
class ocp_txn_seq extends vmm_ss_scenario #(ocp_txn);
  pull_indications indications;
endclass : ocp_txn_seq

class ocp_simple_read_seq extends ocp_txn_seq;
  ...
  virtual task apply(ocp_txn_channel channel,
                     ref int unsigned n_insts);
    ocp_txn req;
    forever begin
      req = new();
      indications.wait_for(pull_indications::READY);
      if (!req.randomize() with {req.cmd==RD;}) ...
      channel.put(req); // blocks until removed
      n_insts++;
    end
  endtask : apply
endclass : ocp_simple_read_request_scenario
```

The sequence takes advantage of the fact that this channel only has room for a single entry and that the `put()` call will block after inserting the transaction until the channel is empty. This means that the sequence will be delayed until the master has responded and removed the transaction from the channel.

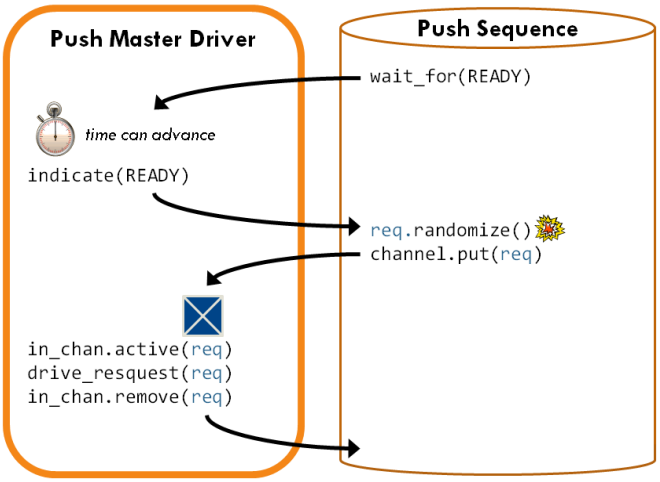

Figure 12: VMM Push Master w/ Notifier (OCP Dataflow)

The OCP master observes the OCP interface and when it is available, it will check for a transaction from the sequence. The master checks to see if the `READY` indicator is being waited on by the sequence, before setting the indicator and getting the next transaction from the sequence. This prevents the master from being blocked when the sequence has no transaction to provide.

```
class ocp_master extends vmm_xactor;
  ...
  virtual task main();
    fork
      super.main();
    join_none
    forever begin
      ocp_txn req;
      @(...);
      wait_if_stopped();
      if (interface_is_available() &&
          indications.is_waited_for(pull_indications::
                                    READY)) begin
        indications.indicate(pull_indications::READY);
        in_chan.activate(req);
        indications.reset(pull_indications::READY);
        ...
        void'(in_chan.start());
        drive_request(req);
        void'(in_chan.complete());
        void'(in_chan.remove());
      end
    end
  endtask : main
endclass : ocp_master
```

The removal of the transaction from the channel allows the sequence's `put()` call to complete, as mentioned previously. The master resets the `READY` indicator before removing the transaction, to ensure that the sequence will block on the next `wait_for(READY)` call. This ensures that the next transaction is not randomized until the master is ready for it.

3.5 OCP Dataflow Slave

OCP dataflow slave drivers are responsible for responding to requests on the OCP interface with responses. The final set of

examples show implementation of OCP dataflow slave drivers in OVM and VMM.

Since the slave drivers are reactive, they will ensure that the response transactions are generated after the received request transactions.

3.5.1 Pull Dataflow Slave (OVM)

The OVM implementation of the OCP dataflow slave uses an OVM pull driver. The flow of control is shown in Figure 13.

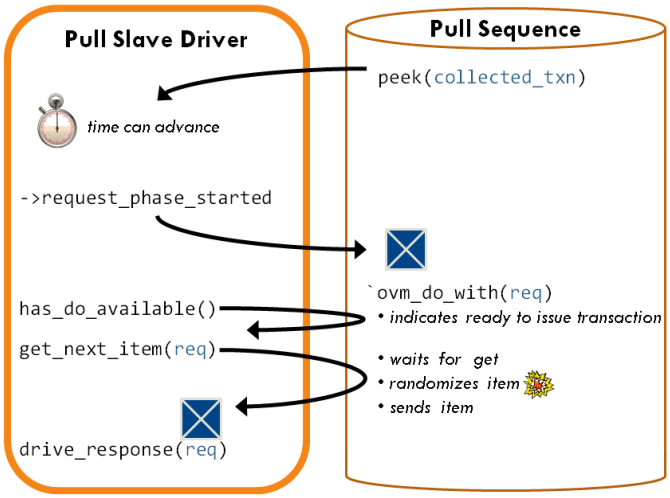

Figure 14: OVM Reactive Pull Slave (OCP Dataflow)

This example uses the simple response sequence below, which generates a response transaction for ever request transaction.

```
class ocp_simple_response_seq extends
                                ovm_sequence #(ocp_txn);
  virtual task body();
    forever begin
      p_sequencer.request_phase_peek_port.peek(
                                        collected_txn);
      `ovm_do_with(req,
                   {req.addr == collected_txn.addr;...})
    end
  endtask : body
endclass : ocp_simple_response_seq
```

The sequence loops forever in order to be able to respond to as many requests as needed. The first thing it does is to call the blocking `peek()` task to get the collected request transaction from the slave. This hands over timing control to the driver, allowing it to decide when it is ready for the next transaction from the sequence.

When the driver has collected the next request transaction, it allows the `peek()` to complete and returns the collected transaction to the sequence. The sequence then calls the `` `ovm_do_with `` macro, which: waits for the driver to ask for the next transaction, randomizes the response and sends it to the driver. (The variable called `req` is a built-in OVM variable for storing the transaction generated by a sequence. The name is confusing here because we are using it to store a response transaction, but don't let the name obscure the local usage.) Notice in the flow of control that the randomization of the response transaction only occurs after the slave has called `get_next_item()`.

The code for the slave is shown below. First, it observes a request on the OCP interface and stores the collected request into the member variable `m_current_request`. Then it triggers an event, which allows the blocking `peek()` method to return the observed request to the sequence. Next, the driver checks to see if the sequence is ready to provide a response transaction. If a response transaction is available, the driver will get it from the sequence and drive it on the OCP interface.

```
class ocp_slave extends ovm_driver#(ocp_txn);
  event   m_request_phase_started;
  ocp_txn m_current_request;
  task peek(output ocp_txn collected_txn);
    @m_request_phase_started;
    collected_txn = m_current_request;
  endtask : peek
  virtual task drive_interface();
    forever begin
      @(...);
      if (request_is_ready()) begin
        m_current_request = collect_request();
        -> m_request_phase_started;
        if (seq_item_port.has_do_available()) begin
          // Sequencer response via TLM port
          seq_item_port.get_next_item(req);
          drive_response(req);
        end
      end
      if (response_is_finished()) begin
        seq_item_port.item_done();
      end
    end
  endtask : drive_interface
endclass : ocp_slave
```

The `has_do_available()` check in the slave prevents the slave from being blocked if the sequence is not yet ready to provide a response transaction. A pull response sequence could examine the collected request and decide not to respond immediately, in which case it would not perform the `` `ovm_do_with() `` and the slave would repeat the entire process the next cycle.

3.5.2 "Pull" Dataflow Slave (VMM with Notifier)

The VMM implementation of the OCP dataflow slave uses a VMM push driver to emulate pull mode.

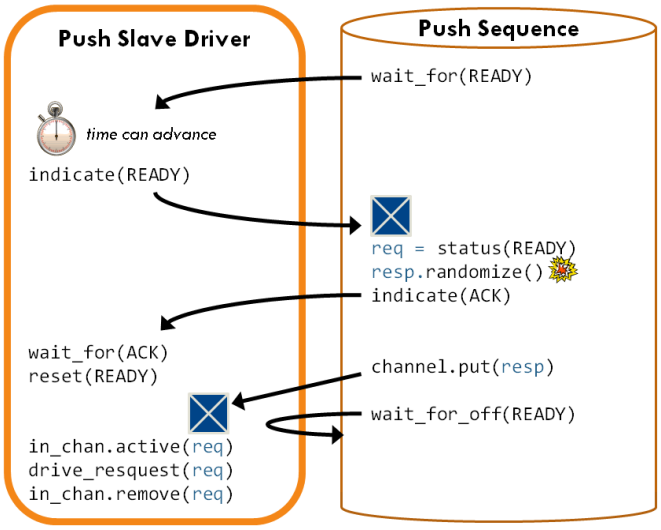

Figure 15: VMM Reactive Push Slave (OCP Dataflow)

Similar to the VMM dataflow master in section 3.4.2, a VMM notifier is connected between the OCP slave and sequence. The notifier is used for the handshaking required between the two and for providing the sequence with the collected request transaction. The reactive slave needs an additional ACK indicator, to acknowledge receipt of the READY indicator. This is explained in more detail when the driver is discussed later.

```
class pull_indications extends vmm_notify;
  typedef enum {READY, ACK} indications_e;
  ...
endclass: pull_indications
```

This example uses the response sequence below.

```
class ocp_txn_seq extends vmm_ss_scenario #(ocp_txn);
  pull_indications indications;
endclass : ocp_txn_seq

class ocp_simple_response_seq extends ocp_txn_seq;
  ...
  virtual task apply(ocp_txn_channel channel,
                     ref int unsigned n_insts);
    ocp_txn req;
    ocp_txn resp;
    bit     responding;
    forever begin
      resp = new();
      indications.wait_for(pull_indications::READY);
      dataflow_req = ocp_txn'(
           indications.status(pull_indications::READY));
      responding = want_to_respond(req);
      if (responding) begin
        if (!resp.randomize() with
            {resp.addr==req.addr;}) ...
      end
      indications.indicate(pull_indications::ACK);
      if (responding) channel.put(resp);
      indications.wait_for_off(pull_indications::READY);
      indications.reset(pull_indications::ACK);
      n_insts++;
    end
  endtask : apply
endclass : ocp_simple_read_request_scenario
```

The sequence loops forever in order to be able to respond to as many requests as needed. It first waits on the READY indicator from the slave, which tells the sequence when the slave has collected a request transaction. A VMM indicator can also include an associated "status" transaction. In this case, the READY indicator's status transaction is the request collected by the slave. Once the sequence gets the collected request from the notifier, the sequence decides if it is going to provide a response now, or wait to respond in a later cycle. If the sequence decides to generate a response, then response transaction is randomized and put() into the channel to the slave driver.

The sequence will always set the ACK indicator to tell the driver that it has processed the request transaction, even if the sequence decides not to respond this cycle. Finally, the sequence waits until the READY indicator is cleared by slave before it clears the ACK indicator, thereby ensuring that the slave saw the ACK indicator.

The slave driver code is shown below. First, it observes a request on the OCP interface and checks to see if the sequence is waiting for a request. This allows the slave to not be blocked if the sequence is not ready to handle a request. If the sequence is ready, then the request is collected into the variable req and associated with the READY indicator that is set for the sequence to observer.

```
class ocp_slave extends vmm_xactor;
  ...
  virtual task main();
    fork
      super.main();
    join_none
    forever begin
      ocp_txn req;
      ocp_txn resp;
      @(...);
      wait_if_stopped();
      if (request_is_ready() &&
          indications.is_waited_for(pull_indications::
                                    READY)) begin
        req = collect_request();
        indications.indicate(pull_indications::READY,
                             req);
        indications.wait_for(pull_indications::ACK);
        indications.reset(pull_indications::READY);
        if (resp_chan.level() > 0) begin
          resp_chan.activate(dataflow_resp);
          void'(resp_chan.start());
          drive_response(dataflow_resp);
          void'(resp_chan.complete());
          void'(resp_chan.remove());
        end
      end
    end
  endtask : main
endclass : ocp_slave
```

Next, the driver waits for the ACK indicator from the sequence and then clears the READY indicator. This provides a complete handshake with the sequence. During the process, the sequence has decided if it wants to reply to the request. The driver can tell if the sequence has replied by checking to see if there is a response transaction in the channel. If a response transaction is available, the driver will get it and drive it on the OCP interface.

If the sequence decided not to issue a response this cycle, then the driver can continue without blocking on trying to pull from the channel.

4 CONCLUSION

Stimulus is an integral part of every verification environment, including those created in SystemVerilog using the OVM or VMM. Both methodologies use different terminology and seemingly different data structures to facilitate creation of a stimulus model. When examined more closely, though, it is easy to see that each library supports similar capabilities in almost every respect. The concepts of sequences, virtual sequences, sequencers, and drivers are common to both the OVM and VMM. We described the purpose of each of the aforementioned components and showed how they could be used. We then demonstrated how to create both push and pull-mode sequencers in the OVM and VMM using a real-life verification component based on the Open Core Protocol (OCP).

5 REFERENCES

[1] JL Gray. (2009, August) The Past, Present and Future of the Design Automation conference. Blog. [Online]. http://www.coolverification.com/2009/08/dac-past-present-future.html

[2] Accellera, *Verification Intellectual Property (VIP) Recommended Practices v1.0*., August 2009. [Online]. http://www.accellera.org/activities/vip/VIP_1.0.pdf

[3] JL Gray. (2009, July) Zero to Sequences in 30 Minutes. Presentation.

[4] Jason Sprott, JL Gray, Sumit Dhamanwala, and Cliff Cummings, "Using the New Features in VMM 1.1 for Multi-Stream Scenarios," 2009.

[5] Synopsys. (2009) VMM 1.2 Documentation.

[6] Cadence Design Systems and Mentor Graphics. (2009) OVM 2.0.3 User Guide.

[7] Cadence Design Systems and Mentor Graphics. (2009) OVM 2.0.3 Reference Manual.

[8] OCP-IP Association. (2007) Open Core Protocol Specification 2.2.

