Seven Separate Sequence Styles Speed Stimulus Scenarios

Mark Peryer
Mentor Graphics (UK) Ltd
Rivergate, London Road, Newbury,
Berkshire, RG14 2QB., UK
mark_peryer@mentor.com

Abstract **Writing effective stimulus in UVM can prove to be challenging for various reasons, but not knowing about the relevant coding design patterns should not be one of them. There are various alternative techniques for writing sequences and choosing the right approach requires mastery of several styles. This paper describes seven common sequence design patterns which should prove useful to all UVM sequence writers. These patterns can be used stand-alone or combined to solve practical stimulus generation problems using UVM sequences.**

Keywords—UVM; sequences; stimulus generation; design patterns; sequence_items; configuration; sequence layering

I. INTRODUCTION

The UVM sequence is designed to be a transitory, or short-lived, class object that is constructed; used to generate stimulus; and then dereferenced, pending garbage collection. It contains a body() method, implemented as a SystemVerilog task, which is called when the sequence is started using its start() method. As the body() method executes it either generates and shapes sequence items, or it creates and starts other sequences. A sequence communicates with a driver via the sequencer component using sequence items.

In practice, most sequences are quite simple, generating a few sequence items before completing. However, more complex stimulus generation requires the use of more advanced techniques. The purpose of this paper is to describe a number of common sequence design patterns.

II. SEQUENCE BASICS

The main purpose of the UVM sequence is to generate sequence_items and to send them to a UVM driver component using an API that is implemented in the UVM sequencer component.

A. The sequence_item

The sequence_item is an object that contains transaction level data that the driver is responsible for interpreting and converting into pin level activity. An example sequence_item is shown in Fig.1. This particular sequence_item contains data items that can notionally be used to describe a bus interface transaction and will be used in most of the design pattern examples that follow. It contains stimulus fields which are rand so that they can be randomized and response fields which are not. Generally, the sequence_item also contains a set of convenience methods which can be used to copy(), clone(), compare(), print(), convert2string(), pack() and unpack() the transaction, but these have been omitted from the example for the sake of brevity.

```
class ex_seq_item extends uvm_sequence_item;

`uvm_object_utils(ex_seq_item)

// Stimulus data
rand bit[31:0] addr;
rand bit[31:0] wdata[16];
rand bit rnw;
rand bit[3:0] length;

// Response data
bit[31:0] rdata[16];
bit[1:0] resp[16];

// Field convenience functions omitted

endclass: ex_seq_item
```

Figure 1- Example sequence_item

B. The Sequence

In its simplest form, a sequence creates a sequence_item, assigns values to its data fields and then sends it to the sequencer. The code example in Fig.2 shows how this is typically done. The active part of the sequence is the body() task method. In this method, the sequence_item is created, then the blocking start_item() method is called which returns when the sequencer is ready to send the sequence_item to the driver. In this example, the sequence_item is then randomized and sent to the driver using the blocking finish_item() method. The sequence ends when the driver unblocks the finish_item() method.

```
class ex_seq extends uvm_sequence #(ex_seq_item);
`uvm_object_utils(ex_seq)

task body;
  ex_seq_item item = ex_seq_item::type_id::create("item");

  start_item(item);
  if(!item.randomize()) begin
    `uvm_warning("ex_seq::body", "randomization failure")
  end
  finish_item(item);
endtask: body
endclass: ex_seq
```

Figure 2 - Example sequence

There are many potential variations on this theme, such as randomizing the sequence_item using in-line constraints; sending a series of sequence_items; or spawning sub-sequences. Test cases for a particular verification environment are built up from a collection of sequences which are run in the context of the test function and the configured testbench topology. The design patterns that are described in this paper

address various common issues that are encountered when generating stimulus streams using sequences.

III. SEVEN SEQUENCE DESIGN PATTERNS

The example sequence in figure 2 generates stimulus by randomizing a sequence_item. However, in the absence of any constraints there is no control over the final value of the fields in the sequence_item. The process of adding control to allow the randomization of the field values is often referred to as shaping. One of the simplest sequence design patterns is therefore known as the shaped sequence.

A. The Shaped Sequence

Intent: To ensure that a sequence's stimulus generation behavior can be controlled programmatically.

Motivation: In order to reduce the overall number of sequences that have to be written, there is a need to generalize sequences so that they can be reused in different scenarios. This generalization can be used to shape the profile of the data content of the sequence item.

Applicability: The shaped sequence pattern is of general use to ensure that sequences are not hard-wired and can be used in more than one specific context. Examples of shaping include setting the address and data values in a bus access sequence; controlling the number of iterations of a generation loop; and defining bounds for in-line constraints.

Implementation: A sequence can be shaped by adding data fields as shown in the code example in Fig.3. The data fields can be assigned values, either directly or via methods implemented in the sequence, and these affect the way in which stimulus is generated within the sequence. In the example, the fields that control the stimulus side of the interface behavior are marked as rand allowing the value assignment to be made by calling the randomize() method of the sequence from a higher level sequence, together with any in-line constraints. The response fields of the sequence are not rand, and they can be assigned response information which can be accessed from outside the sequence. This allows a series of sequences to be chained together, using the result from one to guide the next.

Note the use of the local namespace in the example (e.g. local::addr) to avoid confusion between the addr variable in the item and the addr variable in the sequence.

```
class shaped_sequence extends uvm_sequence #(ex_seq_item);
`uvm_object_utils(shaped_sequence)

// Stimulus data
rand bit[31:0] addr;
rand bit rnw;
rand bit[3:0] length;
// Response data
bit[31:0] data[16];
bit[1:0] resp[16];

task body;
  ex_seq_item item = ex_seq_item::type_id::create("item");

  start_item(item);
  // Randomize with shaped data
  if(!item.randomize() with {addr == local::addr;
                             rnw == local::rnw;
                             length == local::length;})
  begin
    `uvm_warning("body", "randomization failure")
  end
  finish_item(item);
  // Get response data
  if(rnw)begin
    data = item.rdata;
  end
  else begin
    data = item.wdata;
  end
  resp = item.resp;
endtask: body
endclass: shaped_sequence
```

Figure 3 - Shaped Sequence Design Pattern Example

As the number of control fields required to shape or control a sequence increases, it becomes easier to wrap the variables in an object which is passed to the sequence to configure it. This is particularly useful if there are a number of related sequences which share a common set of variables. This form of shaped sequence is referred to as the configurable sequence.

B. The Configurable Sequence

Intent: To control the behavior of a complex generalized sequence using a data object.

Motivation: Sequences which have complex behavior or have a large number of control variables can more easily be configured using a data object, rather than using field assignments or configuration methods.

Applicability: The configurable sequence pattern can be used in the situation where there are many variables which affect the behavior of the sequence or the shape of the sequence_items it produces. The pattern is particularly useful if there are several sequences which share common configuration information and the some configuration object can be shared with all of them. An example application is a bus traffic modeling sequence where the bus agent driver is capable of generating transactions for the complete bus protocol but the master in question is only required to generate a sub-set of the protocol. In this case, the configuration object is used to constrain the generation of bus transactions to the supported sub-set without having to modify the bus agent driver.

```
class seq_config extends uvm_object;
`uvm_object_utils(seq_config)

// General stimulus shaping
rand bit[31:0] start_addr;
rand bit[31:0] wdata[16];
rand bit rnw;

// The protocol sub-set supported:
rand bit[3:0] burst_length[] = '{1, 4, 8, 16};
rand burst_type_e valid_bursts[] = '{INCR};
// ... Other protocol sub-set definitions
rand protection_e prot[] = '{CODE_SECURE, DATA_TRUST_ZONE,
                             DATA_USER};

endclass seq_config
```

Figure 4 - Sequence Configuration Object

Implementation: The configurable sequence pattern is implemented by pairing a sequence with a sequence configuration object. The sequence configuration object contains fields which are used by the sequence to either affect

the way in which it behaves or to shape the sequence items that it generates.

Fig.4 shows an implementation of a sequence configuration object for a bus orientated sequence. This contains a mix of variables which shape the generated bus traffic and variables that specify which sub-set of the bus protocol is supported by the bus master. The corresponding configurable sequence example is shown in Fig.5. Here, the bus sequence_item is randomized using constraints based on the shaping content of the configuration object and its protocol sub-set definition. The same configuration object could be used with other sequences to ensure that they only generate bus sequence_items that are constrained to be within the same protocol sub-set.

```
class configurable_sequence extends
                                uvm_sequence #(ex_seq_item);

`uvm_object_utils(configurable_sequence)

// Handle for sequence configuration object
seq_config cfg;

// Function to allow configuration to be set
function void set_seq_config(seq_config s_cfg);
  cfg = s_cfg;
endfunction: set_seq_config

task body;
  ex_seq_item item = ex_seq_item::type_id::create("item");

  // Item constrained to be within protocol sub-set
  // Driver supports the whole protocol
  start_item(item);
  if(!item.randomize() with
   {addr == start_addr;
    wdata == wdata;
    rnw == rnw;
    // Protocol sub-set constraints
    burst_length inside {cfg.burst_length};
    valid_bursts inside {cfg.valid_bursts};
    // ... Other protocol sub-set constraints
    protection inside {cfg.prot};}) begin
     `uvm_error("body", "Constraint error with ex_seq_item")
  end
  finish_item(item);

endtask: body
endclass: configurable_sequence
```

Figure 5 - Configurable Sequence Design Pattern Example

The configurable sequence pattern works well in the situation where the sequence is self-contained and does not need to refer to anything else in the testbench to control or shape the stimulus. However, there are many situations where access to an external resource, such as a register model, is required to support the stimulus generation process. This requirement leads to the next sequence design pattern, the resourced sequence.

C. The Resourced Sequence

Intent: To allow a sequence to access resources within the testbench hierarchy.

Motivation: In the UVM, sequences are not part of the testbench component hierarchy and cannot directly access resources which may be useful for stimulus generation.

Applicability: The resourced sequence pattern ensures that a sequence can access data and methods available in other parts of the testbench, often giving visibility of system state or device configuration. Examples include register model handles and methods that give information on clock and reset states.

Implementation: The resourced sequence pattern is usually implemented using a base class from which other sequences are derived. The base class contains handles to the required resources and is responsible for getting the handle from the UVM configuration database.

```
class resourced_sequence_base extends
                                  uvm_sequence #(ex_seq_item);
`uvm_object_utils(resource_sequence_base)
// Configuration object containing resource handles
// and methods
env_config cfg;
// Handle to register model
asic_reg_model rm;

// Handles to common register variables
uvm_status_e status;
uvm_reg_data_t data;

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

// Responsible for:
// Getting handle to configuration object
// Assigning register model handle
task body;
  if(!uvm_config_db #(env_config)::get(m_sequencer, "",
"env_config", cfg)) begin
    `uvm_error("body", "Unable to find env_config in
uvm_config_db")
  end
  rm = cfg.rm;
endtask: body
endclass: resourced_sequence_base
```

Figure 6 - Resourced Sequence base class

Fig.6 shows an example implementation of such a base class. In its body() method, a handle for a configuration object is retrieved from the uvm_config_db using the sequencer handle (m_sequencer). The configuration object contains a number of methods and a handle for a UVM register model that inheriting sequences can use.

```
class resourced_sequence extends resourced_sequence_base;
`uvm_object_utils(resource_sequence)

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

task body;
  super.body(); // Assigns resource handles
  cfg.wait_for_reset(); // Config method
  rm.lte.dsp.fltr_cfg.write(status, 32'hdeadbeef,
                            .parent(this));
  // ...
endtask: body
endclass: resourced_sequence
```

Figure 7 - Resourced Sequence Design Pattern

An example of an inheriting resourced sequence is shown in Fig.7, this calls the body() method of the base sequence to assign handles to the testbench resources before calling a wait_for_reset() method available in the configuration object and then making a series of register accesses via the register model. The same sequence might also modify its behavior based on the device configuration information contained in the register model.

All of the sequence patterns described so far concern the generation of stimulus on a single UVM agent, which in turn drives a single interface. In order to generate and control stimulus over several interfaces a variant of the resourced sequence pattern is required and this is known as the virtual sequence.

Note that the virtual sequence design pattern described here is an alternative to the virtual sequencer/virtual sequence pattern described elsewhere. Its advantages are that it is more flexible and it avoids the use of a virtual sequencer component that quickly becomes redundant with vertical reuse.

D. The Virtual Sequence

Intent: To control the execution of stimulus on multiple interfaces.

Motivation: Almost all practical testbenches use sequences to generate stimulus streams for more than one signal-level interface, this means that the separate sequence streams have to be controlled and coordinated by a central control thread.

Applicability: The virtual sequence pattern is used to control the overall sequence stimulus generation process in most UVM test cases. An example of the flow of a virtual sequence running in a block level testbench would be to initialize and configure a block using sequences on a peripheral bus interface, and then to transfer data in and out of the block using sequences controlling other interfaces before checking the overall result via sequences running on the peripheral bus interface.

Implementation: The virtual sequence contains handles for the sequencers for each of the target interfaces, it then runs sub-sequences on the target sequencers as required to implement the test case functionality. The virtual sequence is an extension of the resourced sequence pattern and is usually implemented with a base class from which specific virtual sequences are derived. Such a virtual sequence base class is shown in Fig.8, it contains handles for three target sequencers and a register model.

```
class vseq_base extends uvm_sequence #(ex_seq_item);
`uvm_object_utils(vseq_base)

// Sequencers
target1_sequencer t1;
target2_sequencer t2;
target3_sequencer t3;

// Register model
asic_reg_model rm;

function new(name = "vseq_base");
  super.new(name);
endfunction

endclass: vseq_base
```

Figure 8 - Virtual Sequence base class

An example virtual sequence is shown in Fig.9, this extends the base class and its body method contains a set of sequences each of which are written in the context of a particular target sequencer. These sequences are run in turn on the target sequencers to implement the test case.

```
class sys_vseq extends vseq_base;
`uvm_object_utils(sys_vseq)

function new(name = "sys_vseq");
  super.new(name);
endfunction

task body;
  // Sub sequences:
  t1_setup setup_1 = t1_setup::type_id::create("setup_1");
  t3_setup setup_3 = t3_setup::type_id::create("setup_3");
  t2_slave slave_2 = t2_slave::type_id::create("slave_2");
  t1_t2_transfer t1_t2 =
                 t1_t2_transfer::type_id::create("t1_t2");
  t2_t3_transfer t2_t3 =
                 t2_t3_transfer::type_id::create("t2_t3");

  setup_1.rm = rm;
  setup_2.rm = rm;
  slave_2.rm = rm;
  t1_t2.rm = rm;
  t2_t3.rm = rm;

  fork
    t1_setup.start(t1);
    t3_setup.start(t3);
  join

  fork
    slave_2.start(t2);
    begin
      t1_t2.start(t1);
      t2_t3.start(t3);
    end
  join_any;

endtask: body

endclass: sys_vseq
```

Figure 9 - Virtual Sequence Design Pattern Example

The virtual sequence is usually started by a UVM test class. In order to assign the target sequencer handles in the virtual sequence an initialization method is added to the test base class. The run_phase() method of the derived test case class calls this initialization method before starting the virtual sequence. This is illustrated in the code fragments in Fig.10.

```
// From the test base class:
function void test_base::init_vseq(vseq_base vseq);
  vseq.rm = asic_rm;
  vseq.t1 = env.ch_1.bus_agent.m_sequencer;
  vseq.t2 = env.ch_3.bus_agent.m_sequencer;
  vseq.t3 = env.ch_3.bus_agent.m_sequencer;
endfunction

// From the test class:
task system_test::run_phase(uvm_phase phase);
  sys_vseq vseq = sys_vseq::type_id::create("vseq");

  phase.raise_objection(this);
  init_vseq(vseq);
  vseq.start(null);
  phase.drop_objection(this);
endtask: run_phase
```

Figure 10 - Test Class methods for initializing and starting a virtual sequence

Virtual sequences can be used to combine several sequences running on different sequencers to define the stimulus generation for a whole test case. Another simple way to improve sequence creation productivity is to use abstraction layers built from hierarchical sequences.

E. The Hierarchical Sequence

Intent: To create abstraction layers for stimulus generation.

Motivation: Introducing layers of abstraction makes stimulus easier to write and increases productivity by using proven lower level sequences.

Applicability: The Hierarchical Sequence pattern can be used to build up layers of abstraction, starting with a layer of atomic sequences which are used by the next layer of the hierarchy to implement common functions and then by subsequent layers of hierarchy to implement increasingly more abstract functions. An example from a disk driver verification environment would be atomic sequences to handle tasks like finding an index, writing a header, writing data and verifying data. The next level of abstraction hierarchy would be sequence to write a file to disk based on the atomic sequences, and then the top level would be a sequence to write a file and then read it back.

```
class write_burst_16 extends bus_base;
`uvm_object_utils(write_burst_16)

task body;
  ex_seq_item item = ex_seq_item::type_id::create("item");

  start_item(item);
  if(!item.randomize() with {addr == local::addr;
                             length == 4'hf;
                             rnw == 0;}) begin
    `uvm_error("body", "Constraint error in randomization")
  end
  item.wdata = data;
  finish_item(item);
endtask: body
endclass: write_burst_16
typedef class read_burst16;
typedef class single_read;;
typedef class single_write;
```

Figure 11 - Atomic Level Sequence

Implementation: The Hierarchical Sequence pattern is implemented by creating a library of sequences which can be called in successive layers of abstraction. The lowest layer sequences have a specific function, but are generalized. As the layers become more abstract, the sequences become less general.

Fig.11 shows an example atomic level sequence which implements a burst write. Included in the code excerpt are typedefs for other atomic level sequences.

```
class setup_ch1 extends bus_base;
`uvm_object_utils(setup_ch1)

task body;
  single_write write =
single_write::type_id::create("write");
  write_burst_16 write_16 =
write_burst_16::type_id::create("write_16");

  write.addr = `BUFFER1_START;
  write.data[0] = addr;
  write.start(m_sequencer);
  // ...
  // Set up area in memory
  write_16.addr = addr;
  write_16.start(m_sequencer);
  // ...
endtask: body
endclass: setup_ch1
```

Figure 12 – Mid-level of hierarchical sequence

The next level of hierarchy is shown in Fig.12 where a setup sequence is defined in terms of atomic sequences, again typedefs are included for other sequences at this abstraction layer. A further level of sequence hierarchy is shown in Fig. 13 this combines sequences from the middle abstraction layer to implement a complex function.

```
class ch1_ch2_transfer extends bus_base;
`uvm_object_utils(ch1_ch2_transfer)

task body;
  setup_ch1 setup_1 = setup_ch1::type_id::create("setup_1");
  setup_ch2 setup_2 = setup_ch2::type_id::create("setup_2");
  dma_transfer transfer =
                dma_transfer::type_id::create("transfer");

  setup_1.addr = `DMA_READ_ADDR;
  setup_2.addr = `DMA_WRITE_ADDR;

  setup_1.start(m_sequencer);
  setup_2.start(m_sequencer);
  transfer.start(m_sequencer);

endtask: body
endclass: ch1_ch2_transfer
```

Figure 13 - Top level of hierarchical sequence

Stimulus generated from a set of hierarchical sequences is well ordered, with each layer calling the next and so on. However, there may be situations where the exact stimulus that gets generated or the order in which it gets generated is not important. For instance the programming of registers in a DUT may be specified to be order independent, so it is important to test this by configuring the device in a random order. The sequence library pattern facilitates the generation of this type of stimulus.

F. The Sequence Library

Intent: To be able to select and execute one of several sequences at will.

Motivation: Some stimulus generation scenarios require that a random choice be made from a set of available sequences in order to flush out subtle interactions in behavior.

Applicability: The sequence library pattern is used when one of several stimulus options would be valid but it does not matter which one is selected and executed. The choice of which sequence gets executed is usually randomized. Examples include generating background irritant traffic where the selected sequence has the potential to cause an interaction with foreground traffic or randomizing the order in which the registers of a configurable device are programmed.

```
package bus_sequence_lib_pkg;

import uvm_pkg::*;
`include "uvm_macros.svh"

// Base class for the library
`include "bus_seq_base.svh"
// Library sequences extending from base class:
`include "read_burst_seq.svh"
`include "write_burst_seq.svh"
`include "read_modify_write_seq.svh"
`include "write_check_read_seq.svh"
`include "read_block_seq.svh"
`include "write_block_seq.svh"
// Library sequence:
`include "bus_seq_library.svh"
```

```
// Irritant sequence:
`include "bus_irritant_seq.svh"

endpackage: bus_sequence_lib_pkg

class bus_seq_library extends bus_seq_base;
`uvm_object_utils(bus_seq_library)

// Associative array of sequences, indexed by string
bus_seq_base lib[string];
bus_seq_base sel_seq;
string sel;

task body
  lib["read_burst"] =
   read_burst_seq::type_id::create("read_burst");
  lib["write_burst"] =
   write_burst_seq::type_id::create("write_burst");
  lib["read_mod_write"] =
   read_modify_write_seq::type_id::create("read_mod_write");
  lib["write_chk_read"] =
   write_check_read_seq::type_id::create("write_chk_read");
  lib["read_block"] =
   read_block_seq::type_id::create("read_block");
  lib["write_block"] =
   write_block_seq::type_id::create("write_block");

  // Choose a sequence at random by shuffling the array:
  lib.shuffle();
  // Take the sequence at the top of the pile and start
  sel_seq = lib.first(sel);
  sel_seq.start(m_sequencer);
endtask: body

endclass: bus_seq_library
```

Figure 14 - Library Sequence Design Pattern Example

Implementation: The sequence library pattern can be implemented by exploiting polymorphism. This is illustrated by the code in Fig.14 which contains a sequence library class that contains an associative array of base sequence handles. In the body method of the library sequence, each element of the array is assigned a handle corresponding to one of the sequences declared in the sequence library package. The default behavior of the sequence library is then to shuffle the order of the array randomly and then start the first sequence in the array.

```
class bus_irritant_seq extends bus_seq_library;
`uvm_object_utils(bus_irritant_seq)

rand int iterations = 1;

constraint limit {
  iterations inside {[1:20]};
}


task body;
  super.body();
  repeat(iterations) begin
    lib.shuffle();
    sel_seq = lib.first(sel);
    sel_seq.start(m_sequencer);
  end
endtask: body

endclass: bus_irritant_seq
```

Figure 15 - Bus irritant sequence extended from the bus_seq_library class

An example extension to the bus_seq_library class is shown in Fig.15, this is an irritant sequence which will run up to 20 sub-sequences randomly selected from the library.

All of the design patterns examined so far have concerned the generation of stimulus in terms of sequence_items that are sent directly to a specific UVM driver attached to a target interface. In some cases, it is more convenient to generate stimulus in one form of sequence_item and convert it to another before it is applied to a target interface, this is where the layering sequence pattern comes into play.

G. The Layering Sequence

Intent: To transform one abstract representation of stimulus data to another.

Motivation: Often it is convenient to generate data using one virtual or abstract representation that needs to go through a transformation process before it can be applied to a concrete interface. In other stimulus generation scenarios it may be necessary to map or combine several virtual data streams into a single data stream.

Applicability: The layering sequence pattern is applicable to any situation where sequences are available that use one sequence_item but must transformed to another sequence_item to be executed on a target sequencer. An example of this would be converting a uvm_tlm_generic_payload item to a bus specific sequence_item.

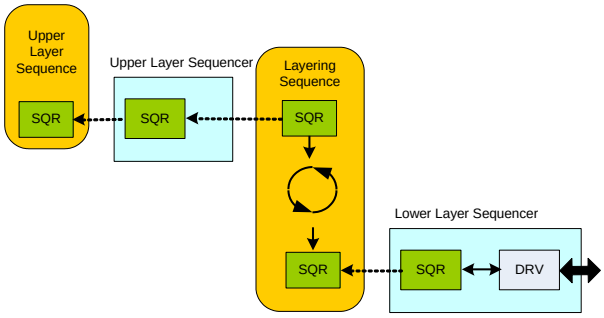

Figure 16 - Layering Sequence Block Diagram

Other applications would be modeling a layered protocol or where several types of data streams need to be mapped onto a common transport layer - for example, mixing streams of video data, voice data and pure data traffic onto an Ethernet or USB transport layer.

The layering sequence pattern can also be stacked, or chained, to implement multiple conversion layers.

```
class layering_sequence extends uvm_sequence #(usb_item);
`uvm_object_utils(layering_sequence)

// Upper-layer sequencer handle:
uvm_sequencer #(audio_item) voice_sequencer;

// Get a audio_item, start a usb_item
// Translate the audio_item to the usb_item
// finish the usb_item
// Call item_done() on the audio_item
task body;
  audio_item voc;
  usb_item usb = usb_item::type_id::create("usb");

  forever begin
    voice_sequencer.get_next_item(voc);
    start_item(usb);
    convert_voc_2_usb(usb, voc);
    finish_item(usb);
    voice_sequencer.item_done();
  end
endtask body;
endclass: layering_sequence
```

```
// From the env:
// Create the layering sequence
// Set its upstream sequencer to the audio sequencer
// Start the sequence on the usb sequencer


task run_phase(uvm_phase phase);
  layering_sequence audio_2_usb =
layering_sequence::type_id::create("audio_2_usb");
  audio_2_usb.voice_sequencer = audio.m_sequencer;
  audio_2_usb.start(usb._m_sequencer);
endtask: run_phase
```

Figure 17 - Layering Sequence Design Pattern Example

Implementation: The layering sequence is implemented by declaring the handle for the upper-layer sequencer. In the body method of the layering sequence, handles to sequence_items from the upper-layer sequencer are assigned via the sequencers get_next_item() method. (Normally accessed from the driver via the sequencers seq_item_export) The upper-layer sequence_items are then transformed into lower-level sequence_item(s) and then sent to the lower-layer sequencer using its start_item() and finish_item() methods. When the upper-layer sequence_item has been converted, the upper-layer sequencers item_done() method is called to complete the transfer.

This process flow is illustrated by the code example in Fig. 17. The example shows a layering sequence that converts audio format sequence_items to USB format sequence_items. The voice_sequencer is the upper-layer sequencer. In the body method, there is a loop where the audio sequence_items are got from the voice_sequencer, converted to USB sequence_items and then sent to the USB sequencer using the start/finish() item methods. When the finish_item() method completes, the audio_sequencers item_done() method is called before the loop starts over. The layering sequence is started in the run_phase() method of the env after the handle to the voice_sequencer has been assigned to the upper-layer sequencer in the layering sequence. The sequence stimulus stream for the audio data will run on the voice_sequencer with no modification but appear on the USB interface rather than an audio interface.

This example shows a one to one layering, but it is possible to have a many to one layering e.g. Video, Audio, and data links over a physical transport layer such as USB or Ethernet or multiple levels of layering or any combination thereof.

IV. CONCLUSION

The seven sequence design patterns described in this paper can be used separately or combined to solve stimulus generation challenges. UVM sequences provide a powerful and flexible way of writing stimulus. Knowing about the design patterns described here is a useful addition to any verification engineer's toolkit.

REFERENCES

[1] Accellera, Universal Verification Methodology (UVM) 1.1 Users Guide, 2011

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

[3] Verification Academy, UVM Cookbook, https://verificationacademy.com/cookbook

[4] Verification Academy, UVM Cookbook Sequences/Overview, https://verificationacademy.com/cookbook/Sequences/Overview.