Seven Separate Sequence Styles Speed Stimulus Scenarios

Sequence design patterns for your UVM stimulus coding toolkit

The Shaped Sequence

Intent: To control a sequence's generation behaviour programatically

Motivation: Enabling sequence reuse.

Applicability: To any sequence, adding fields to control generation so generalising the sequence.

Implementation: Add random members for control fields, non-random members for status fields. The rand field values are used to constrain the sequence item(s) generated by 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];

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

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
```

The Hierarchical Sequence

Intent: To create abstraction layers in stimulus generation.

Motivation: Improve productivity by defining complex stimulus at a high level.

Applicability: To any stimulus generation activity that can be disassembled into a series of sub-tasks.

Implementation: Create a library of sequences that implement the task hierarchy with a bottom layer of 'worker' sequences.

'Worker' Layer ➡

```
class top extends base_seq;
task body;
  next_level.start(m_sequencer);
endtask: body
endclass: top
```

```
class next_level extends base_seq;
task body;
  fork
    dma.start(m_sequencer);
    tfr.start(m_sequencer);
  join
endtask: body
endclass: next_level
```

```
class dma extends base_seq;
task body;
  setup.start(m_sequencer);
  dma_read.start(m_sequencer);
endtask: body
endclass: dma
```

```
class tfr extends base_seq;
task body;
  btm_1.start(m_sequencer);
  btm_2.start(m_sequencer);
endtask: body
endclass: tfr
```

`class setup ….` `class dma_read` `class btm_1 ….` `class btm_2 ….`

The Configurable Sequence

Intent: To control a complex generalised sequence.

Motivation: Ease shaping difficulties with many variables.

Applicability: To any sequence, particularly related sequences which share shaping variables.

Implementation: Create a configuration that is paired with the sequence. The sequence takes it shaping guards from the configuration object that is randomized outside the sequence.

```
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 prot_e prot[] = '{CODE_SECURE, DATA_TRUST_ZONE,
DATA_USER};
endclass seq_config

class config_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};})
   `uvm_error("body", "Constraint error with ex_seq_item")
 finish_item(item);
endtask: body
endclass: config_sequence
```

The Library Sequence

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

Motivation: To support stimulus generation scenarios where random choices of sequences are required

Applicability: Useful when one of several sequences would be valid.

Implementation: Create a library sequence that contains an array of sequences extended from a common base type. Select and execute sequences from the array.

```
class library_sequence extends base_seq;

common_sequence_base_t seq_a[string];
common_sequence_base_t sel_seq
```

index	Common_sequence_base_t
"A"	Sequence_A_h
"B"	Sequence_B_h
"C"	Sequence_C_h
"D"	Sequence_D_h

```
function void set_seq(string key, common_sequence_base_t seq_h);
  seq_a[key] = seq_h
endfunction: set_seq

function void select_seq(string key);
  sel_seq = seq_a[key];
endfunction: select_seq

task body;
  sel_seq.start(m_sequencer);
endtask: body

endclass: library_sequence
```

The Resourced Sequence

Intent: To access testbench resources.

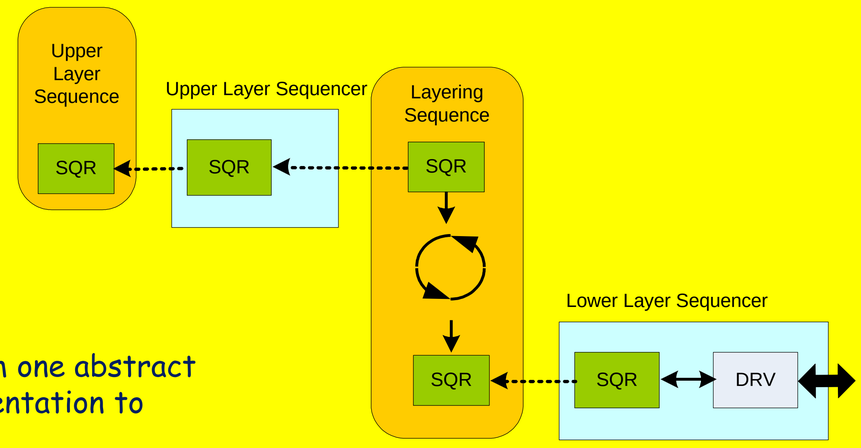
Motivation: To generate stimulus based on information present in the UVM testbench hierarchy

Applicability: To register based sequences and sequences making generation decisions based on design or testbench state.

Implementation: A base class assigns to resource handles using the m_sequencer to access the uvm_config_db.

```
class resourced_seq_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;

// 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_seq_base

class resourced_seq extends resourced_seq_base;
`uvm_object_utils(resource_seq)

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_seq
```

The Layering Sequence



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

Motivation: It is often more convenient to generate stimulus in one format and then transform it to another.

Applicability: Useful any time one type of sequence_item needs to be changed to another. E.g. getting video data to stream on an USB transport layer.

Implementation: Insert a sequencer that has a layering sequence running on it. The layering sequence uses the sequencers seq_item_export methods to get the upper layer items, then transforms them before sending to the lower layer sequencer.
Multiple layerings can be combined or chained, as illustrated in the example code.

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

// 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: audio_layering_seq

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

// Upper-layer sequencer handle:
uvm_sequencer #(video_item) video_sequencer;

task body;
  // Get video items, transform to USB items
  // Send to usb sequencer
endtask body;
endclass: layering_sequence


// From the env:
// Create the layering sequences
// Set upstream sequencers to the audio & video sequencers
// Start the layering sequences on the usb sequencer
task run_phase(uvm_phase phase);
  audio_layering_seq a_2_u =
              audio_layering_seq::type_id::create("a_2_u");
  a2u.voice_sequencer = audio_sequencer;

  video_layering_seq v_2_u =
              video_layering_seq::type_id::create("v_2_u");
  v2u.video_sequencer = video_sequencer;

  fork
    a2u.start(usb.m_sequencer);
    v2u.start(usb.m_sequencer);
  join
endtask: run_phase
```

The Virtual Sequence

Intent: To generate stimulus on several DUT interfaces.

Motivation: Most testbenches need to generate and co-ordinate stimulus on two or more interfaces.

Applicability: Used to control the stimulus generation in most UVM test cases.

Implementation: An extension of the resourced sequence pattern with handles for the target sequencers. An init_vseq() method in the test assigns the sequencer handles to the virtual sequence.

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

// Sequencer handles
target1_sequencer_t t1;
target2_sequencer_t t2;
// Register model
asic_reg_model rm;

endclass: vseq_base

class sys_vseq extends vseq_base;
`uvm_object_utils(sys_vseq)

task body;
  // Sub sequences:
  t1_setup setup = t1_setup::type_id::create("setup");
  t2_slave slave = t2_slave::type_id::create("slave");
  t1_t2_tfr t1_t2 = t1_t2_tfr::type_id::create("t1_t2");
  setup.rm = rm;
  slave.rm = rm;

  setup.start(t1);
  fork
    slave.start(t2);
    t1_t2.start(t1);
  join_any;

endtask: body
endclass: sys_vseq

// Handles assigned 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;
Endfunction: init_vseq

// Started 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
```

DVCon 2013 Design & Verification Conference & Exhibition

mark_peryer@mentor.com

Mentor Graphics®