The Finer Points of UVM: Tasting Tips for the Connoisseur

John Aynsley, Doulos
The Finer Points of UVM

- Sequences and sequencers
- The arbitration queue
- Virtual sequences
- Request and response
- Multiple sequencer stacks
The Big Picture

uvm_test

uvm_test

uvm_env

uvm_scoreboard

Virtual sequencer

Register Layer

uvm_agent

uvm_agent

uvm_agent

DUT

Factory

config db

uvm_agent

uvm_sequencer

uvm_monitor

uvm_driver
Sequences and Sequencers

start_item(req);
finish_item(req);

seq_item_port.get(req);
class my_seq extends uvm_sequence #(my_tx);
    `uvm_object_utils(my_seq)

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

    task body;
        repeat(4)
            begin
                req = my_tx::type_id::create("req");
                start_item(req);

                if (!req.randomize())
                    `uvm_error("", "failed to randomize")
                finish_item(req);
            end
    endtask
endclass
class top_seq extends uvm_sequence #(my_tx);
...

`uvm_declare_p_sequencer(my_seqr)
...

task body;
  repeat(3)
  begin
    my_seq seq;
    seq = my_seq::type_id::create("seq");

    if (!seq.randomize())
      `uvm_error("", "failed to randomize")

    seq.start(p_sequencer, this);
  end
endtask
...
task body;
  fork
  begin
    seq1 = my_seq::type_id::create("seq1");
    if (!seq1.randomize())
      `uvm_error("", "failed to randomize")
    seq1.start(p_sequencer, this);
  end
  begin
    seq2 = my_seq::type_id::create("seq2");
    if (!seq2.randomize())
      ...
    seq2.start(p_sequencer, this);
  end
  begin
    ...
    seq3.start(p_sequencer, this);
  end
  join
endtask
The Finer Points of UVM

• Sequences and sequencers
• The arbitration queue
• Virtual sequences
• Request and response
• Multiple sequencer stacks
The Arbitration Queue

```
begin
  seq_item_port.get(req);
  seq_item_port.get(req);
  seq_item_port.get(req);
end
```
task body;
    p_sequencer.set_arbitration(
        SEQ_ARB_STRICT_RANDOM);
fork
    begin
        seq1 = my_seq::type_id::create("seq1");
        if (!seq1.randomize())
            `uvm_error("", "failed to randomize")
        seq1.start(p_sequencer, this, 1);
    end
    begin
        ...
        seq2.start(p_sequencer, this, 2);
    end
    begin
        ...
        seq3.start(p_sequencer, this, 3);
    end
    join
endtask
### Arbitration Algorithms

<table>
<thead>
<tr>
<th>Arbitration mode</th>
<th>Order in which requests granted</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQ_ARB_FIFO</td>
<td>FIFO order (default)</td>
</tr>
<tr>
<td>SEQ_ARB_RANDOM</td>
<td>Random order</td>
</tr>
<tr>
<td>SEQ_ARB STRICT_FIFO</td>
<td>Highest priority first, then FIFO order</td>
</tr>
<tr>
<td>SEQ_ARB STRICT_RANDOM</td>
<td>Highest priority first, then random order</td>
</tr>
<tr>
<td>SEQ_ARB_WEIGHTED</td>
<td>Weighted by priority</td>
</tr>
<tr>
<td>SEQ_ARB_USER</td>
<td>User-defined</td>
</tr>
</tbody>
</table>
class my_sequencer extends uvm_sequencer #(my_tx);
...

function integer user_priority_arbitration(
    integer avail_sequences[$]);

    foreach (avail_sequences[i])
        begin
            integer            index = avail_sequences[i];
            uvm_sequence_request req = arb_sequence_q[index];
            int pri = req.item_priority;
            uvm_sequence_base seq = req.sequence_ptr;

            if (pri > max_pri)
                ...  

        end

    return max_index;
endfunction

endclass

Could access properties of the sequence object
The Finer Points of UVM

- Sequences and sequencers
- The arbitration queue
- Virtual sequences
- Request and response
- Multiple sequencer stacks
Virtual Sequences

```
vseq.start(seqr0, null, priority)
    body
    fork
        seq1.start(seqr1, this)
            body
                start_item
                ...
        seq2.start(seqr2, this, 50)
            body
                start_item
                ...
```

Can be null

No transactions

Blocks
Inherits priority

Can be null

No transactions

Blocks
Inherits priority
Sequencer Lock

No transactions

**Important!**

```
vseq.start(seqr0, null)
body
begin
  this.lock(seqr1);
  seq1.start(seqr1, this);
  body
    start_item
    finish_item
  this.unlock(seqr1);
...```

Driver

```
priority
seqx
```

```
priority
seqy
```

```
priority
seq1
```
Lock versus Grab

Virtual sequence

\[ \text{vseq1.start} \]
\[ \text{body} \]
\[ \text{begin} \]
\[ \text{lock} \]
\[ \text{seq1.start} \]
\[ \text{unlock} \]

Virtual sequence

\[ \text{vseq2.start} \]
\[ \text{body} \]
\[ \text{begin} \]
\[ \text{lock} \]
\[ \text{seq2.start} \]
\[ \text{unlock} \]

Virtual sequence

\[ \text{vseq3.start} \]
\[ \text{body} \]
\[ \text{begin} \]
\[ \text{grab} \]
\[ \text{seq3.start} \]
\[ \text{ungrab} \]

Sequencer

Head

- \[ \text{grab req} \]
- \[ \text{priority} \]
- \[ \text{vseq3} \]
- \[ \text{seqx} \]

Tail

- \[ \text{lock req} \]
- \[ \text{priority} \]
- \[ \text{vseq2} \]
- \[ \text{seq3} \]

Grabs inserted here

Locks inserted here

Driver

Head Tail
The Finer Points of UVM

• Sequences and sequencers
• The arbitration queue
• Virtual sequences
• Request and response
• Multiple sequencer stacks
The paper describes in detail how to code pipelined req/rsp
Layered Sequencers

- `seq_item_port.get(req)`
- `seq_item_port.put(rsp)`
- `seqr_upper.get(req_up)`
- `start_item(req_lo)`
- `finish_item(req_lo)`
- `get_response(rsp_lo)`
- `seqr_upper.put(rsp_up)`
- `seq_item_port.get(req)`
- `seq_item_port.put(rsp)`

Could be one:one or one:many or many:one

The paper shows more detail
The Finer Points of UVM

- Sequences and sequencers
- The arbitration queue
- Virtual sequences
- Request and response
- Multiple sequencer stacks
Multiple Agents / Sequencer Stacks

Communicate or synchronize?

Analysis ports
Callbacks
Events
Barriers

Driven by the DUT interface timing

get(req)

Must not block!
Driver calls `try_next_item`

```verilog
seq_item_port.try_next_item(req);

if (req == null)
  begin
    dut_vi.idle <= 1;
    ...
    @(posedge dut_vi.clock);
  end
else
  begin
    seq_item_port.item_done();
    dut_vi.idle <= 0;
    ...
    @(posedge dut_vi.clock);
    ...
    seq_item_port.put(rsp);
  end
```

- **Wiggle pins for idle cycle**
- **Must be called in same time step**
- **Response can be pipelined**
The Finer Points of UVM

Also in the paper

• The UVM sequence library
• Pipelined requests and responses
• The response handler
• UVM events and event pools
• The configuration database