Fun with UVM Sequences
Coding and Debugging

Rich Edelman
Mentor, A Siemens Business
Fremont, CA
UVM Sequences

• Sequences are CODE
• “Programs that do things”
  – Issue a WRITE Transaction
UVM Sequences

- Sequences can
  - Create and start transactions
  - Create and start other sequences
UVM Sequences

• Sequences can
  – Point at other sequences
  – Point at other classes
    • Synchronization
Creating A Sequence

- A Test might create a sequence
- Four sequences running in parallel
- LIMIT is different

```cpp
class test extends uvm_test;
`uvm_component_utils(test)

my_sequence seq;
...

    task run_phase(uvm_phase phase);
        phase.raise_object(this);
        for (int i = 0; i < 4; i++) begin
            fork
                automatic int j = i;
                seq = new($sformatf("seq%0d", j));
                seq.LIMIT = 25 * (j+1);
                seq.start(sqr);
            join_none
            end
        wait fork;
        phase.drop_object(this);
    endtask
endclass
```
Running A Sequence – Sequence Items

- The sequence
- LIMIT controls the for-loop
- Each time through the loop
  - CONSTRUCT a transaction
  - “start” it
  - Set some data
  - Randomize it
  - “finish” it

```verilog
class my_sequence extends uvm_sequence#(transaction);
  `uvm_object_utils(my_sequence)
  transaction t;
  int LIMIT;
  ...
  task body();
      for (int i = 0; i < LIMIT; i++) begin
        t = new("t");
        start_item(t);
        t.data = i+1;
        if (!t.randomize())
          `uvm_fatal(get_type_name(),
            "Randomize FAILED")
        finish_item(t);
      end
  endtask
endclass
```
The Driver – Executing a Sequence Item

• The driver receives the transaction

• Forever loop
  – Get_next_item() gets the next transaction
  – Then “operate” on the transaction
  – Here: Just print and Wait.
  – Signal done with item_done()

```verilog
class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

transaction t;
...

task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(t);
    `uvm_info(get_type_name(),
      $sformatf("Got %s",
        t.convert2string()),
      UVM_MEDIUM)
    #(t.duration); // Execute...
    seq_item_port.item_done();
  end
```
Virtual Sequences

• A sequence has a body which is the program.
• This body() starts two sequences
  – sequenceA
  – sequenceB
• The sequences are started on two sequencers
  – sequencerA
  – sequencerB

```cpp
class virtual_sequence ...

  sequenceA_t sequenceA;
  sequenceB_t sequenceB;

  sequencerA_sqrA;
  sequencerB_sqrB;

task body();
  sequenceA.start(sqrA);
  sequenceB.start(sqrB);
  ...
```
**Controlling Related Sequences**

- Two sequences
  - `ping_h`
  - `pong_h`

- They share handles

- The sequences are started

```java
ping_h = new("ping_h");
ping_h.LIMIT = 25;
pong_h = new("pong_h");
pong_h.LIMIT = 40;

ping_h.pong_h = pong_h;
pong_h.ping_h = ping_h;

fork
  ping_h.start(sqr);
  pong_h.start(sqr);
join
```
Self-Checking Sequences

```plaintext
task body();
for (int i = 0; i < LIMIT; i++) begin
    t = new($sformatf("t%0d", i));
    start_item(t);
    t.rw = WRITE;
    finish_item(t);
end

for (int i = 0; i < LIMIT; i++) begin
    tr = new($sformatf("t%0d", i));
    start_item(tr);
    tr.rw = READ;
    finish_item(tr);

    // Check
    if (t.data != tr.data) begin
        ...
    end
end
endtask
```
Traffic Generator Sequences

- Generate “transactions” at some typical RATE for the kind of traffic
  - Video
  - UART
  - Lumpy calculations
  - Random

- 60 screens per second

```plaintext
class video extends my_sequence;

task body();

screendots = xpixels * ypixels;
rate = 1_000_000_000 / (60 * screendots);

forever begin
    addr = 0;
    for (x = 0; x < xpixels; x++) begin
        for (y = 0; y < ypixels; y++) begin
            t = new($sformatf("t%0d_%0d", x, y));
            start_item(t);
            t.rw = WRITE;
            t.addr = addr++;
            t.duration = rate;
            finish_item(t);
        end
    end
end
```
typedef enum bit { STOP, GO } synchro_t;

class synchronizer;
    synchro_t state;
endclass

class synchro extends my_sequence;
    synchronizer s;

    task body();
        forever begin
            addr = start_addr;
            // Is it safe?
            while (s.state == STOP) begin
                #10;
            end
            t = new($sformatf("t%0d", addr));
            start_item(t);
            finish_item(t);
        end
    endtask
endclass

fork
    forever begin
        #100;
        s.state = GO;
        #20;
        s.state = STOP;
    end
join_none

Interrupt Service Routine Sequences

- Start a sequence that does NOT complete UNTIL an interrupt is detected

```plaintext
class interrupt_transaction extends transaction;
    `uvm_object_utils(transaction)
    int VALUE;
    bit DONE;
endclass

class interrupt_sequence extends my_sequence;
    `uvm_object_utils(interrupt_sequence)
    interrupt_transaction t;

    task body();
        forever begin
            t = new("isr_transaction");
            start_item(t);
            finish_item(t);
            wait(t.DONE == 1);
        end
    endtask
endclass
```
class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

transaction t;
interrupt_transaction isr;
bit done;
int value;

  task interrupt_service_routine(
    interrupt_transaction isr_h);
  done = 0;
  isr_h.DONE = 0;
  wait(done == 1);
  isr_h.VALUE = value;
  isr_h.DONE = 1;
  endtask

  endclass

  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);

      if ($cast(isr, t)) begin
        fork
          interrupt_service_routine(isr);
        join_none
        end
      else begin
        ...
        // REGULAR driver processing
        ...
        if (AN INTERRUPT OCCURS) begin
          done = 1;
          value = mem[t.addr];
          end
        end
      end_item_port.item_done();
    end
  endtask

endclass
Sequences with Utility Libraries

- “Helper functions”
  - Read
  - Write

- Special body()
  - Never exits

```verilog
class open_door extends my_sequence;
`uvm_object_utils(open_door)

read_transaction r;
write_transaction w;

task read(input bit[31:0]addr, output bit[31:0]data);
  ...
endtask

task write(input bit[31:0]addr, input bit[31:0]data);
  ...
endtask

task body();
  wait(0);
endtask
endclass
```
Sequences with Utility Libraries

- Construct the sequence
- Start the sequence
- Call write
- Call read

```pascal
open_door open_door_h;
open_door_h = new("open_door");
fork
  open_door_h.start(sqr);
  begin
    bit [31:0] rdata;
    for (int i = 0; i < 100; i++) begin
      open_door_h.write(i, i+1);
      open_door_h.read(i, rdata);
    end
  end
join_none
```
import "DPI-C" function void c_code_add(output int z, input int a, input int b);
export "DPI-C" function sv_code;

function void sv_code(int z);
   $display("sv_code(z=%0d)", z);
endfunction

#include "stdio.h"
#include "dpiheader.h"

void
c_code_add(int *z, int a, int b)
{
   *z = a + b;
   sv_code(*z);
}
class use_c_code_sequence extends my_sequence;
    `uvm_object_utils(use_c_code_sequence)
    int z;
    c_code_transaction t;

    task body();
    forever begin
        for (int i = 0; i < 10; i++) begin
            for (int j = 0; j < 10; j++) begin
                c_code_add(z, i, j);
                t = new($sformatf("t%0d", i));
                start_item(t);
                finish_item(t);
            end
        end
    endtask
endclass
Recording Sequences as Transactions

- Transactions are “automatically” recorded by the UVM
Summary

• Sequences are fun
• Sequences are just code
• Sequences can coordinate and work with each other
• Sequences cause things to happen
• Sequences are cool

• Thank you!