UVM Sans UVM
An approach to automating UVM testbench writing

Rich Edelman, Mentor
Shashi Bhutada, Mentor
Goal: Better Verification ➔ Use UVM

- More stimulus with less effort
  - Constraints
  - Seeds
- Functional coverage
  - Some
  - Better
- Simpler abstraction for thinking about tests
- Simpler adoption of SystemVerilog
- Management thinks **UVM is a “good idea”**
- Resume update
UVM Vocabulary

• OOP
• Configuration database
• Resource database
• Factory
• Phases
• Components
• Objects
• Macros!
• Transactions

• Sequences
• Sequencers
• Sequence Items
• Randomization and Constraints
• Agents
• Environments
• Tests
• Virtual Interfaces
• uvm_info & VERBOSITY
Learning the UVM is like Learning to Drive

- Perilous
- Rules → DMV / CHP
- Can be Tedious
  - Classroom class
  - Behind the wheel class
  - 60 hours practice
- Try it out
  - Sunday morning
  - Safe work parking lot and business park

- Get more brave
  - Country road/Straight line driving…
  - Long drive to Yosemite
  - Glacier Point Road!
- Maps!! Guides
  - Reference material
Learning to Drive
Learning the UVM

• Pushback
  – Kids these days…. They don’t want to drive…
    • “Dad, You’ll just make me run errands”
  – Will the UVM improve my verification?
    • What’s the Return on Investment?

• Driving and the UVM
  – Independence
  – Responsibility
  – Productivity
  – Flexibility
  – Exploring new spaces

Driving Results – 10 months later
• Really good driver
• Really big smiles

UVM Results – 3-8 hours later
• Random traffic
• Simple functional coverage
Better Verification with UVM

• Training
• Learning curve

• Practice
• Becoming an expert
What to do next?
Automation

- What if we had a template and a generator?
- What if we had a framework?
- What if we had a way to come up running a UVM testbench in a day? (or less)
Who might use a generator + template?

- Block tester
- Non-UVM user
- New-UVM user
- UVM user tired of typing all the boilerplate

There are

- Many successful templates
- Many successful generators
- Google
  - “Easy UVM”, “UVM Framework”, “UVM Template”
Still need a little bit of UVM

- An **agent** manages an **interface**
- An **environment** is a simple container for **agents**
- A **test** constructs an **environment**, configures it and starts **sequences** running on the **agents**
- A **sequence** is a program which creates **transactions** and sends them to the **driver**
- A **driver** takes a **transaction** and causes pin wiggles.
- A **monitor** watches pin wiggles and creates **transactions**.
A UVM Agent

ABC Agent

XYZ Agent

Driver

Monitor

Sequencer

Coverage

Sequence

DUT

ABC Agent

XYZ Agent

Driver

Monitor

Sequencer

Coverage

Sequence

DUT

SB
What's really going on?

Sequence

creates

Sequencer

orders

Driver

executes

Transaction

start_item(t)

finish_item(t)

get_next_item(t)

item_done()

BFM & Interface

Monitor

recognizes & creates

ap.write(t)

sample(t)

Coverage

transaction

sampled
A UVM Testbench Generator

- The generator is not the goal.
  - Keep it simple
  - Copy the AGENT, changing names
  - Copy the env, test, top, etc.

```
#!/bin/csh -f
mkdir -p GENERATED
foreach agent ( ABC XYZ )
  tar cf -C uvm-templates --transform=s/MY_AGENT_/${agent}_/g agents/MY_AGENT_agent |
  tar xvf - -C GENERATED
pushd GENERATED/agents/${agent}_agent
  foreach file ( `find . -type f` )
    sed -e s/MY_AGENT_/${agent}_/g < $file > $file.NEW
    mv $file.NEW $file
  end
popd
end

```

Generator Eye Test

- Hand Edit
- Generated Code
- Templates
- My New Testbench
- tar and sed
A UVM Testbench Template

• The template is Runnable!

```
uvm-templates/
|   agents
|   └── MY_AGENT_agent
|       ├── MY_AGENT_agent.svh
|       ├── MY_AGENT_agent_pkg.sv
|       ├── MY_AGENT_config.svh
|       ├── MY_AGENT_coverage.svh
|       └── MY_AGENT_driver.svh
|       └── MY_AGENT_if.svh
|           ├── MY_AGENT_monitor.svh
|           └── MY_AGENT_seq.svh
|           └── MY_AGENT_seq_item.svh
|               └── MY_AGENT_sequencer.svh
```

```
uvm-templates/
|   env
|   └── env.svh
|       └── test_0.svh
|       └── test_pkg.sv
|   rtl
|   └── sim
|       └── makefile
|   tb
|       └── dut_wrapper.sv
|       └── test_top.sv
|       └── top.sv
```

18 files, 544 lines of code
A UVM Testbench Generated

• Name changes… MY_AGENT → ABC_/ XYZ_

28 files, 1400 lines of code
Your bus
Write an interface

- Your bus, Your pins. Collected together in a bundle

```verilog
interface ABC_if ( input CLK );
    logic RST;

    logic VALID; // Goes high when the addr/data is valid.
    logic READY; // Goes high when the DUT is ready.

    logic RW;    // READ = 1, WRITE = 0
    logic BURST; // High when a burst begins, low when it ends.

    logic [31:0] ADDR;

    logic [31:0] DATAI;
    wire [31:0] DATAO;

endinterface
```
Write a BFM interface

• BFM “helper” routines

```verilog
interface ABC_bfm ( ABC_if bus );
    task reset();
    task read( bit[31:0] addr, output bit[31:0] data);
    task write( bit[31:0] addr, input bit[31:0] data);
    task burst_read( bit[31:0] addr, output bit[31:0] data[4]);
    task burst_write( bit[31:0] addr, bit[31:0] data[4]);
    task monitor( output bit transaction_ok,
                   bit rw, bit [31:0] addr, bit [31:0] data[4], bit burst);
endinterface: ABC_bfm
```

Your BFM
Write a BFM write()

• write(addr, data)

```vhdl
task write(bit[31:0] addr, input bit[31:0] data);
    bus.RW <= 0;
    bus.BURST <= 0;
    bus.ADDR <= addr;
    bus.DATAI <= data;
    bus.VALID <= 1;
    @(posedge bus.CLK);
    while (!bus.READY)
        @(posedge bus.CLK);
    bus.VALID <= 0;
endtask
```

Call write() to cause a write transaction
```verilog
task monitor(output bit transaction_ok, 
    bit rw, bit [31:0] addr, bit [31:0] data[4], bit burst);
    @(negedge bus.CLK);
    if ((bus.READY == '1) && (bus.VALID == '1) && 
        (bus.BURST == '0)) begin
        burst = 0;
        rw = bus.RW;
        addr = bus.ADDR;
        @(posedge bus.CLK);
        if (rw == 1) // READ
            data[0] = bus.DATAO;
        else // WRITE
            data[0] = bus.DATAI;
        transaction_ok = 1;
    end
endtask
```
Write a transaction

• What is your test abstraction?
  – READ/WRITE
  – READ/WRITE Registers
  – “Direct Map”
  • signals == transaction

• What constraints are needed?

```verilog
class ABC_seq_item extends uvm_sequence_item;
  rand bit burst;
  rand bit rw;
  rand bit[31:0] addr;
  rand bit[31:0] data[4];
  constraint val_addr { addr >= 0; addr <= 136; }
endclass
```

```verilog
interface ABC_if (...);
  logic RST;
  logic VALID;
  logic READY;
  logic RW;
  logic BURST;
  logic [31:0] ADDR;
  logic [31:0] DATAI;
  wire [31:0] DATAO;
endinterface
```

Your Bus
class ABC_coverage extends uvm_subscriber #(ABC_seq_item);

ABC_seq_item t;

covergroup cg;

burst_cp: coverpoint t.burst;

rw_cp: coverpoint t.rw;

virtual function void sample(ABC_seq_item t);

this.t = t;

cg.sample();
endfunction

function void write(ABC_seq_item t);

sample(t);

`uvm_info("COVERAGE", $sformatf("Coverage=%0d%% (t=%s)", cg.get_inst_coverage(), t.convert2string()), UVM_MEDIUM)
endfunction
endclass
Write a driver

class ABC_driver extends uvm_driver#(ABC_seq_item);
  virtual ABC_bfm bfm;
  task run_phase(uvm_phase phase);
    ABC_seq_item item;
    forever begin
      seq_item_port.get_next_item(item);
      `uvm_info("DRIVER",
        $sformatf("item=%s", item.convert2string()),
        UVM_MEDIUM)
      if (item rw == 1)
        bfm.read(item.addr, item.data[0]);
      else
        bfm.write(item.addr, item.data[0]);
      seq_item_port.item_done();
    end
  endtask
endclass: ABC_driver

Convert a transaction into a BFM call
Write a monitor

class ABC_monitor extends uvm_monitor;
   virtual ABC_bfm bfm;
   uvm_analysis_port #(ABC_seq_item) ap;

   function void build_phase(uvm_phase phase);
      ap = new("ap", this);
   endfunction

   task run_phase(uvm_phase phase);
      ABC_seq_item t;
      forever begin
         t = ABC_seq_item::type_id::create("t");
         bfm.monitor(t.rw, t.addr, t.data, t.burst);
         ap.write(t);
      end
   endtask
endclass: ABC_monitor

Monitoring the bus with the BFM Monitor
Write a sequence

class ABC_seq extends uvm_sequence #(ABC_seq_item);

   int number_of_items = 100;

   rand bit[31:0] addr, addr_low, addr_high;

   constraint rand_range {addr_low>='h00; addr_high<='h82;
                           addr_low<addr_high;}

   task body();
      ABC_seq_item item;
      for(int i = 0; i < number_of_items; i++) begin
         item = ABC_seq_item::type_id::create("item");
         start_item(item);
         item.randomize() with {addr>=local::addr_low;
                               addr<=local::addr_high;};
         finish_item(item);
      end
   endtask
endclass: ABC_seq
class env extends uvm_env;
    ABC_agent ABC_1_agent_h;
    ABC_config ABC_1_config_h;
    XYZ_agent XYZ_2_agent_h;
    XYZ_config XYZ_2_config_h;

function void build_phase(uvm_phase phase);
    ABC_1_agent_h = ABC_agent::type_id::create( "ABC_1_agent_h", this);
    ABC_1_config_h = ABC_config::type_id::create( "ABC_1_config_h", this);
    uvm_config_db #(virtual ABC_bfm)::get(null,"","ABC_1_bfm", ABC_1_config_h.bfm);
    ABC_1_agent_h.config_h = ABC_1_config_h;
    ... XYZ
endfunction: build_phase
endclass: env

The "environment" – 2 connections
DUT Wrapper

• Connect pins to interface

```verilog
module dut_wrapper(
    ABC_if ABC_1_if,
    XYZ_if XYZ_2_if
);

dut_mem u_dut(
    .CLK1  (ABC_1_if.CLK),
    .RST1  (ABC_1_if.RST),
    .RW1   (ABC_1_if.RW),
    .READY1(ABC_1_if.READY),
    .VALID1(ABC_1_if.VALID),
    .ADDR1 (ABC_1_if.ADDR),
    .DATA11(ABC_1_if.DATAI),
    .DATAO1(ABC_1_if.DATO);

    .CLK2  (XYZ_2_if.CLK),
    .RST2  (XYZ_2_if.RST),
    .RW2   (XYZ_2_if.RW),
    .READY2(XYZ_2_if.READY),
    .VALID2(XYZ_2_if.VALID),
    .ADDR2 (XYZ_2_if.ADDR),
    .DATA12(XYZ_2_if.DATAI),
    .DATAO2(XYZ_2_if.DATAO)
);
endmodule: dut_wrapper
```

Connectivity
module top;

    ABC_if ABC_1_if (.CLK(clock));
    XYZ_if XYZ_2_if (.CLK(clock));

    dut_wrapper dut1 (  
        .ABC_1_if(ABC_1_if),  
        .XYZ_2_if(XYZ_2_if)  
    );

endmodule: top

Instantiate your RTL and your interfaces
module test_top;

import uvm_pkg::*;
import test_pkg::*;

ABC_bfm ABC_1_bfm (top.ABC_1_if);
XYZ_bfm XYZ_2_bfm (top.XYZ_2_if);

initial begin
  uvm_config_db#(virtual ABC_bfm)::set(null, "", "ABC_1_bfm", ABC_1_bfm);
  uvm_config_db#(virtual XYZ_bfm)::set(null, "", "XYZ_2_bfm", XYZ_2_bfm);
  run_test();
end
endmodule: test_top
Summary

- Use **automation** → a generator, framework, template

- **Build (hand edit)** your
  - Bus(interface), Transaction, Driver and Monitor
  - Constraints
  - Functional coverage

- Enjoy your new High Performance Testbench

- Questions?

And Drive Safely!

Contact **rich_edelman@mentor.com**