UVM Best Practices & Debug

Leo Fang
Synopsys, Inc.
March 2014
Agenda

• UVM Ecosystem
• UVM Best Practices
• UVM Debug
UVM Ecosystem

- Overview
Complete UVM Ecosystem

- Verification Planning
- Verification Management
- Template Generators
- Native UVM VIP
- UVM TLM/SC Adaptors
- UVM AMS Testbench
- Constraint Solver
- UVM-Aware Debug
- Protocol Debug
- Coverage & Analysis
UVM Best Practices

• Overview
• UVM Configuration Mechanism
• UVM RAL and ralgen
• UVM Factory
• UVM Callback
Overview

• Best Practices in
  – Writing Testbench

• Best Practices NOT for
  – Minimum Coding
  – Quickest development time

• Best Practices for
  – Maximal Configurable
  – Maximal Reusable
  – Maximal Extensible
UVM Best Practices

- Overview
- **UVM Configuration Mechanism**
- UVM RAL and ralgen
- UVM Factory
- UVM Callback
Use Config Mechanism to Get/Set Variables

- Every testbench has knobs / settings / parameters
- These are set at the device, agent, environment, test, and even simulation command line
  - Read throughout the testbench
- A simple configuration mechanism will not work
  - Build phase creates TB top down
  - Problem: How can a test set values for components that don’t yet exist?
  - Solution: a hierarchal database not tied to the UVM component hierarchy
## UVM Configuration DB Review

- **Mechanism for configuring component properties**

  ```
  uvm_config_db #(int)::set(this, "scb", "limit", 3)
  ```

- **For simplicity, make the field name the same as variable**

- **Any typos will cause a mismatch, but not an error**

- **Debug with DVE or Verdi’s Resource window**

- **Potential performance issues if used heavily**

### Data Type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Hierarchical Context</th>
<th>Instance Name</th>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>uvm_test_top.env.scb</td>
<td>limit</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

### Code Snippet

```haskell
DB: "uvm_test_top.env.scb.limit" -> 3
```
Example: Two Ways of Configuration

```verilog
class master extends uvm_agent;
  `uvm_component_utils_begin(master)
    `uvm_field_int(base_addr, UVM_ALL_ON)
    `uvm_field_int(max_len, UVM_ALL_ON)
    `uvm_field_int(gen_ecc_err, UVM_ALL_ON)
  `uvm_component_utils_end

function void build_phase(uvm_phase phase);
  uvm_config_db#(bit [31:0])::get(this,"","base_addr", base_addr))
  ...
endfunction
endclass

class mst_cfg extends uvm_object;
  rand bit [31:0] base_addr;
  rand bit [31:0] max_len;
  constraint c_len {max_len < 20;}
endclass
```

```verilog
class master extends uvm_agent;
  mst_cfg cfg;
  `uvm_component_utils_begin(master)
    `uvm_field_obj(cfg, UVM_DEFAULT)
  `uvm_component_utils_end

function void build_phase(...);
  uvm_config_db#(mst_cfg)::get(this,"","cfg", cfg))
  ...
endfunction
endclass
```
UVM Best Practices

- Overview
- UVM Configuration Mechanism
- UVM RAL and ralgen
- UVM Factory
- UVM Callback
Use RAL to model registers and memories

**RAL: Register Abstract Layer**

- Hierarchical model for ease of reuse
- Supports both front door and back door access
- Has Built-in functional coverage (bits, fields and address map for different scopes)
- Contains a Mirror of register data
- Pre-defined tests exercise registers and memories
- Can be reused to different concrete protocol bus
Testbench without RAL

A typical test
- check reset value
- write value
- check value

What about coverage?
self-checking?
randomization?
portability?

Address hardcoded!
Field name unknown!
Status of execution unknown!
class host_sequence extends uvm_sequence #(host_data);  ...;
  virtual task body(); uvm_status_e status; uvm_reg_data_t data;
  regmodel.HOST_ID.read(status, data, UVM_FRONTDOOR, this);
  regmodel.LOCK.write(status, '1, UVM_FRONTDOOR), this);
endtask
endclass

Access via abstracted field name
Status of execution returned

Driver Unchanged
New sequence uses register models’ abstraction to read and write DUT fields
Sequencer Unchanged

Register model with Built-in:
coverage
self-checking
randomization
portable structure

Access can be front door or back door
Use ralgen to generate RAL model

- ralgen: Synopsys Register Model Generator
  - Saves time and effort on modeling registers
  - Makes it easy to deal with changes

Diagram:
- IP-XACT
- RALF
- 3rd Party
- CoSim API
- Pure C API
- Constraints and covergroup
- HTML Documentation
- RAL Model
- RTL DUT
- Backdoor
Example: Use ralgen to Create RAL Model

// host.ralf
register HOST_ID {
    field REV_ID {...}
    field CHIP_ID {...}
}
register PORT_LOCK {
    field LOCK {...}
}
register HOST_REG {
    field USER_REG {...}
}
memory RAM {...}
block host_regmodel {...}
system dut_regmodel {...}

ralgen -uvm -t dut_regmodel host.ralf

// ral_dut_regmodel.sv
class ral_reg_HOST_ID extends uvm_reg;
    uvm_reg_field REV_ID;
    uvm_reg_field CHIP_ID;
...
endclass : ral_reg_HOST_ID
class ral_reg_PORT_LOCK extends uvm_reg;
class ral_reg_HOST_REG extends uvm_reg;
class ral_mem_RAM extends uvm_mem;
class ral_block_host_regmodel extends uvm_reg_block;
    rand ral_reg_HOST_ID HOST_ID;
    rand ral_reg_PORT_LOCK PORT_LOCK;
    rand ral_reg_HOST_REG HOST_REG[256];
    rand ral_mem_RAM RAM;
...
endclass : ral_block_host_regmodel
class ral_sys_dut_regmodel extends uvm_reg_block;
    rand ral_block_host_regmodel HOST0;
    rand ral_block_host_regmodel HOST1;
...
endclass : ral_sys_dut_regmodel
UVM Best Practices

• Overview
• UVM Configuration Mechanism
• UVM RAL and ralgen
• UVM Factory
• UVM Callback
Use factories to create and override objects

- UVM factory is used to manufacture (create) UVM objects and components.
- UVM factory provides a mechanism for overriding the default data items and objects in a testbench.
- Factories Implementation flow
  - Factory instrumentation/registration
    - `uvm_object_utils(Type)
    - `uvm_component_utils(Type)
  - Construct object using static proxy class method
    - ClassName obj = ClassName::type_id::create(…);
  - Class overrides
    - set_type_override (…);
    - set_inst_override (…);

(1) Macro creates a proxy class to represent the object/component called type_id.
(2) Registers an instance of the proxy class in uvm_factory.

Use proxy class to create object

Proxy class in uvm_factory can be overridden to create modified object.
Example: Transaction Factory - 1

- Construct object via `create()` in factory class

```verilog
class packet extends uvm_sequence_item;
    rand bit[3:0] sa, da;
    `uvm_object_utils(packet)
    ...
endclass
```

Required!
Macro defines a proxy class called `type_id`
An instance of proxy class is registered in `uvm_factory`

```verilog
class bfm extends uvm_component;
    task run_phase(uvm_phase phase);
    forever begin
        packet tr;
        tr = packet::type_id::create("tr", this);
        ...
    end
endtask
endclass
```

Parent handle in proxy `create()` method call must be a component

Use `create()` method of proxy class in run phase to construct transaction object
Example: Transaction Factory - 2

• Globally change all “packet” type

```
class newPkt extends packet;
    int serial_no;
    constraint test_constraint {...}
    `uvm_object_utils(newPkt)
endclass
```

Make modifications to existing transaction

Required! Creates proxy class

• Or change just instances found in search path

```set_inst_override("*.bfm.tr", "packet", "newPkt");```
UVM Best Practices

• Overview
• UVM Configuration Mechanism
• UVM RAL and ralgen
• UVM Factory
• UVM Callback
Use Callbacks to Change the Behavior of a Class

- The behavior of a class can be modified either using class extension or callbacks
  - Class Extension
    - Good for large scale changes or constraint modification
  - Callbacks
    - Insert additional code in predetermined places without modifying the original class
      - Insert delays, modify transaction, drop a transaction
    - More Modular
      - Delay Callbacks, Scoreboard Callbacks, Coverage callbacks
      - Multiple callbacks can be added/removed with the same class object
    - More Flexible
      - Callbacks can be added/removed dynamically
Example: UVM Callback

```verilog
class bus_drv extends uvm_component;
  `uvm_register_cb(bus_drv, bus_drv_cb)
  virtual task send(bus_tr tr);
    `uvm_do_callbacks(bus_drv,bus_drv_cb,tr_rcvd(tr, drop));
    if (drop == 1) begin
      `uvm_info("Dropped","User callback DROPPED bus_tr");
      return();
    end
  endtask
endclass

class my_busDrv_cb extends bus_drv_cb;
  virtual function bit tr_rcvd(bus_tr tr, output drop);
    if (tr.kind == MGMT)
      drop = ($urandom_range(10) == 0);
    return drop;
  endfunction
endclass

class test_cbs extends test_base;
  my_bus_drv_cb cb1;
  virtual function void build_phase(...);
    uvm_callbacks #(bus_drv, bus_drv_cb)::add(env.agt0.drvr,cb1);
    ...`
```

- `uvm_do_callback` macro executes all registered callbacks
- callback class
- facade callback class
- Callback drops 10% of packets randomly
- Register a callback
UVM Debug

- SystemVerilog Interactive Debug
- UVM-Aware Debug
Use SystemVerilog Interactive Debug

- With interactive debug, user can set breakpoints, watch variables, stack trace, etc.

Both VCS DVE and Verdi3 support interactive Debug
- VCS: 2012.09-1 or above
- Verdi: 2013.09 or above

- Verdi window is similar to DVE window, and we will focus on VCS DVE in this presentation
VCS DVE Interactive Debug

- UVM-Aware debug must work together with systemverilog interactive debug
Set Breakpoints in Interactive Simulation

- User can set breakpoints to stop the simulation
  - Denotes a disabled line break point
  - Denotes a line breakpoint was set on this line, and it is enabled

```verbatim
// read_byte
virtual protected task read_byte (output bit [7:0] data, output bit error);
  vif.rw <= 1'b0;
  @(posedge vif.sig_clock iff vif.sig_wait === 0);
  data = vif.sig_data;
endtask : read_byte

// write_byte
virtual protected task write_byte (bit[7:0] data, output bit error);
  vif.rw <= 1'b1;
  vif.sig_data_out <= data;
  @(posedge vif.sig_clock iff vif.sig_wait === 0);
  vif.rw <= 'h0;
endtask : write_byte
```
Breakpoints Window

- Create or update breakpoints by breakpoints window
Interactive simulator controls

• Click the **continue** icon \( \downarrow \) to “start/continue” simulation
• Click the **stop** icon \( \bullet \) to stop
• Click **next** icon \( \rightarrow \) to step over tasks and functions
• Click **step** icon \( \{\} \) to simulate next executable line
• Click **step in active thread** icon \( \{\} \) to stop at the next executable line in the current active thread
• Click **next in active thread** icon \( \{\} \) to advance over to the next executable line in the current thread only
• Click **step in testbench** icon \( \{\} \) to stop at the next executable line in the Native TestBench (NTB) OpenVera or SystemVerilog testbench
• Click **step out** icon \( \{\} \) to step to the next executable line outside of the current function or a task
• Click **restart** icon \( \) to reset simulation to time zero
Annotating values

- Three ways to annotate values. The annotated values are displayed in the Source view.
  - Click the Annotate Values Icon in the toolbar.
  - Select Scope → Annotate Values.
  - In the Source view, right-click and select Annotate Values.
Class Pane & Member Pane

- **Class pane**: displays all the classes defined in the design in a hierarchical view.
- **Member pane**: displays the content or methods of the selected class.
Object Pane & Member Pane

- **Object pane**: displays all the active objects defined in the design in a hierarchical view.

  ![Diagram showing Object Pane and Member Pane]
Verdi Interactive Debug

- Simulator Control
- Stack Pane
- Breakpoints
- Local Pane
UVM Debug

- SystemVerilog Interactive Debug
- UVM-Aware Debug
Use UVM-Aware Debug

• Best approach to testbench analysis and debug
• UVM-Aware debug features are designed with the methodology in mind
  – Common testbench issues can be quickly identified and fixed
  – Advanced capabilities to debug the most complex testbench problems
• Reduce time spent debugging
  – Debug and verify fixes without having to recompile or exit simulation
  – “What-if” analysis and checkpoint/rewind

Both VCS DVE and Verdi3 support UVM-Aware Debug
– VCS: 2012.09-1 or above
– Verdi: 2013.09 or above
VCS DVE UVM-Aware Debug

- A total of five view windows: resource, factory, phase, sequence and register views
VCS DVE UVM Resource Debug

UVM-aware support for resource database interaction
VCS DVE UVM Factory Debug

Easily identify when and where factory overrides occurred

Type-based Filtering
VCS DVE UVM Phase/Objection Debug

Enhanced phase management visibility

<table>
<thead>
<tr>
<th>Phase</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td>DONE</td>
</tr>
<tr>
<td>connect</td>
<td>DONE</td>
</tr>
<tr>
<td>end_of_elaboration</td>
<td>DONE</td>
</tr>
<tr>
<td>start_of_simulation</td>
<td>DONE</td>
</tr>
<tr>
<td>run</td>
<td>DONE</td>
</tr>
<tr>
<td>extract</td>
<td>DONE</td>
</tr>
<tr>
<td>check</td>
<td>DONE</td>
</tr>
<tr>
<td>report</td>
<td>DONE</td>
</tr>
<tr>
<td>final</td>
<td>DONE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase</th>
<th>Domain</th>
<th>State</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>shutdown</td>
<td>uvm</td>
<td>DONE</td>
<td>530</td>
</tr>
<tr>
<td>post_shutdo...</td>
<td>uvm</td>
<td>SCHEDULED</td>
<td>530</td>
</tr>
<tr>
<td>post_shutdo...</td>
<td>uvm</td>
<td>STRT</td>
<td>530</td>
</tr>
<tr>
<td>post_shutdo...</td>
<td>uvm</td>
<td>SKIP</td>
<td>530</td>
</tr>
<tr>
<td>run</td>
<td>common</td>
<td>DONE</td>
<td>530</td>
</tr>
<tr>
<td>extract</td>
<td>common</td>
<td>SCHEDULED</td>
<td>530</td>
</tr>
<tr>
<td>post_shutdo...</td>
<td>uvm</td>
<td>DONE</td>
<td>530</td>
</tr>
<tr>
<td>extract</td>
<td>common</td>
<td>STRT</td>
<td>530</td>
</tr>
<tr>
<td>extract</td>
<td>common</td>
<td>DONE</td>
<td>530</td>
</tr>
</tbody>
</table>

Objection debug capability

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>Count(Total)</th>
<th>Top Count</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Raise</td>
<td>1(1)</td>
<td>1</td>
<td>uvm_test_top.env</td>
</tr>
<tr>
<td>190</td>
<td>Drop</td>
<td>0(0)</td>
<td>0</td>
<td>uvm_test_top.env</td>
</tr>
<tr>
<td>190</td>
<td>Raise</td>
<td>1(1)</td>
<td>1</td>
<td>uvm_test_top.env.reg_agent0.reg_sqr.main_seq</td>
</tr>
<tr>
<td>530</td>
<td>Drop</td>
<td>0(0)</td>
<td>0</td>
<td>uvm_test_top.env.req_agent0.req_sqr.main_seq</td>
</tr>
</tbody>
</table>

© 2014 Synopsys. All rights reserved.
VCS DVE UVM Sequence Debug

Quick tracing of sequences through phases and sequencers
VCS DVE UVM Register Debug
Quick tracing the access and value of RAL

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Size</th>
<th>Value</th>
<th>Desired</th>
<th>Mirrored</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>REVISION_ID</td>
<td>0</td>
<td>25</td>
<td>h0d 6976</td>
<td>h0d 6976</td>
<td>h0d 6976</td>
<td>RO</td>
</tr>
<tr>
<td>CHIP_ID</td>
<td>7</td>
<td>8</td>
<td>h6a</td>
<td>h6a</td>
<td>h6a</td>
<td>RO</td>
</tr>
<tr>
<td>PRODUCT_ID</td>
<td>15:8</td>
<td>8</td>
<td>h176</td>
<td>h176</td>
<td>h176</td>
<td>RO</td>
</tr>
<tr>
<td>INDEX</td>
<td>32</td>
<td>8</td>
<td>h0</td>
<td>h0</td>
<td>h0</td>
<td>RO</td>
</tr>
<tr>
<td>DATA</td>
<td>7</td>
<td>32</td>
<td>h5642 6106</td>
<td>h5642 6106</td>
<td>h56d42 6106</td>
<td>RW</td>
</tr>
<tr>
<td>SRC</td>
<td>31:2</td>
<td>32</td>
<td>h5642 6106</td>
<td>h5642 6106</td>
<td>h56d42 6106</td>
<td>RO</td>
</tr>
<tr>
<td>DST</td>
<td>5117</td>
<td>64</td>
<td>h0</td>
<td>h0</td>
<td>h0</td>
<td>RO</td>
</tr>
<tr>
<td>TABLES[0]</td>
<td>5120</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h0</td>
<td>h0</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[1]</td>
<td>5121</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h7f37.4835</td>
<td>h7f37.4835</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[2]</td>
<td>5122</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[3]</td>
<td>5123</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[4]</td>
<td>5124</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[5]</td>
<td>5125</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[6]</td>
<td>5126</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[7]</td>
<td>5127</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[8]</td>
<td>5128</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[9]</td>
<td>5129</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[10]</td>
<td>5130</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
<tr>
<td>TABLES[11]</td>
<td>5131</td>
<td>32</td>
<td>uvm_reg_field</td>
<td>h661e 0c7c</td>
<td>h661e 0c7c</td>
<td>RW</td>
</tr>
</tbody>
</table>
Verdi UVM-Aware Debug

- A total of five view windows: resource, factory, phase, sequence and register views

Verdi is more powerful than DVE in Register view: RAL Access History Pane
Thank You