UVM Rapid Adoption: A Practical Subset of UVM

Stuart Sutherland, Sutherland-HDL, Inc.
Tom Fitzpatrick, Mentor Graphics Corp.
The Problem...

- The UVM 1.2 Library has **357 classes, 938 functions, 99 tasks, and 374 macros**

  Why are there so many different ways to print a message?

  If it’s in the library, you have to use it!

  How do I find what I need in this huge library?

  I’m so confused!

  Which way should I use?
The Goals of this Paper

• Understand why the UVM library is so complex
• Examine UVM from three different perspectives
  – The Environment Writer
  – The Test Writer
  – The Sequence Writer
• Define a practical subset of UVM that meets the needs of nearly all verification projects
  – A subset makes UVM easier to learn, use & maintain!

You will be amazed at how small of a subset of UVM you really need!
Why the UVM Library Is Overly Large and Complex

- Why 357 classes, 1037 methods, 374 macros?
  - The history of UVM adds to UVM’s complexity
    - UVM evolved from OVM, VMM and other methodologies
    - UVM adds to and modifies previous methodologies
    - UVM contains “old ways” and “new ways” to do things
  - Object Oriented Programming adds complexity
    - OOP extends and inherits functionality from base classes
      - `uvm_driver` inherits from `uvm_component` which inherits from `uvm_object` which inherits from ...
    - Only a small number of UVM classes, methods and macros are intended to be used by end users
      - Much of the UVM library is for use within the library
Three Aspects of a UVM Testbench

1) The Test
- Connects the testbench to the DUT
- Selects the sequencers
- Configures the environment

2) The Sequencer
- Generates transactions (stimulus)

3) The Environment
- Delivers stimulus
- Verifies DUT outputs
UVM Constructs Used By The Environment Writer
The Role of the UVM Environment Writer

- The Environment Writer defines the testbench parts
  - Agents
  - Sequencers
  - Drivers
  - Monitors
  - Scoreboards
  - Coverage collectors

The environment delivers stimulus to the DUT and verifies DUT outputs.
The Environment Component

About the examples in this presentation:
- UVM-specific constructs are shown in blue text
- UVM constructs not shown in previous examples are shown in boxed text

```cpp
class my_env extends uvm_env {
  `uvm_component_utils( my_env )

  function new(string name, uvm_component parent);
  super.new(name, parent);
  endfunction: new

  ...
}
```

Extend base class from UVM lib.

Factory registration macro

Factory will call new() constructor

To save time, we are only going to count the number of UVM constructs required – refer to the paper for more details on these constructs

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Methods</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Macros</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The Environment Component (continued)

- Environments encapsulate an agent and scoreboard

``` Mundiall
    ... 
    my_agent agent;
    my_scoreboard scorebd;

    function void build_phase(uvm_phase phase);
        agent = my_agent::type_id::create("agent", this);
        scorebd = my_scoreboard::type_id::create("scorebd", this);
    endfunction: build_phase

    function void connect_phase(uvm_phase phase);
        agent.dut_inputs_port.connect(scorebd.dut_in_imp_export);
        agent.dut_outputs_port.connect(scorebd.dut_out_imp_export);
    endfunction: connect_phase
endclass: my_env
```

The “build phase” uses factory to “create” components

The “connect phase” is used to “connect” component ports
The Agent Component

• An agent encapsulates low-level components needed to drive and monitor a specific interface to the DUT

```verilog
class my_agent extends uvm_agent;  `uvm_component_utils(my_agent)

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction: new

    // handles for agent’s components
    my_sequencer sqr;
    my_driver drv;

    // handles to the monitor’s ports
    uvm_analysis_port #(my_tx) dut_inputs_port;
    uvm_analysis_port #(my_tx) dut_outputs_port;

(continued on next page)
```
The Agent Component (continued)

• The agent’s **build phase** “creates” a sequencer, driver, monitor, etc.

```plaintext
function void build_phase(uvm_phase phase);

mon = my_monitor::type_id::create("mon", this);
if (m_config.is_active == UVM_ACTIVE) begin
  sqr = my_sequencer::type_id::create("sqr", this);
drv = my_driver::type_id::create("drv", this);
end
if (m_config.enable_coverage)
cov = my_cover_collector::type_id::create("cov", this);
endfunction: build_phase
```

... (continued on next page)
The Agent Component (continued)

- The agent’s `connect_phase` connects the agent’s components together

```plaintext
... function void connect_phase(uvm_phase phase);
    // set agent's ports to point to
    dut_inputs_port = mon.dut_inputs;
    dut_outputs_port = mon.dut_outputs;

    if (is_active == UVM_ACTIVE)
        // connect driver to sequencer
        drv.seq_item_port.connect(sqr.seq_item_export);

    if (enable_coverage)
        // connect monitor to coverage collector
        mon.dut_inputs_port.connect(cov.analysis_export);
endfunction: connect_phase
endclass: my_agent
```

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Methods</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Macros</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
The Driver Component

- The driver receives transactions from a sequencer and drives values to the DUT via a virtual interface.

```verilog
class my_driver extends uvm_driver #(my_tx) ;
    `uvm_component_utils(my_driver)

function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction

virtual tb_if tb_vif;  // virtual interface pointer

function void build_phase(uvm_phase phase);
    if (!uvm_config_db #(virtual my_dut_interface)::get(this, "", "DUT_IF", tb_vif))
        `uvm_fatal("NOVIF", Failed virtual interface from db")
endfunction: build_phase
```

Extend driver’s UVM base class

A fatal error report terminates simulation

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Methods</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Macros</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
The Driver Component

- The driver receives transactions from a sequencer and drives values to the DUT via a virtual interface.

```plaintext

... task run_phase(uvm_phase phase);
  my_tx tx;
  forever begin
    @tb_vif.clk // synchronize to interface clock
    seq_item_port.get_next_item(tx);
    tb_vif.operand_a = tx.operand_a;
    tb_vif.operand_b = tx.operand_b;
    tb_vif.opcode = tx.opcode;
    seq_item_port.item_done();
  end
  endtask: run_phase
endclass: my_driver

The “run phase” is a task that can take clock cycles to execute.

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Methods</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Macros</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Port methods “block” execution flow as part of a handshake process with a sequence stimulus generator written by the Sequence Writer.
Additional Components

• A sequencer routes stimulus to
  – Specializes the `uvm_sequencer base class
  – No additional UVM constructs are needed

• A monitor observes DUT ports via a virtual interface
  – Extends the `uvm_monitor base class
  – Only additional UVM construct needed that has not already been shown is an analysis port write() method

• A scoreboard verifies DUT output value correctness
  – Extends `uvm_subscriber or `uvm_component
  – Only additional UVM constructs that might be needed are: report_phase(), `uvm_info() and `uvm_analysis_imp_decl()

• A coverage collector performs functional coverage
  – No additional UVM constructs are needed

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Methods</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Macros</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Stu Sutherland, Sutherland-HDL & Tom Fitzpatrick, Mentor Graphics
UVM Constructs Used By The Test Writer
The Role of the UVM Test Writer

- The **Test Writer** defines the specifics of a testcase
  - Connects the testbench to the DUT
  - Selects the sequences
  - Configures the environment

![Diagram of test setup](image)
The Top-Level Module

- Top-level module connects DUT and starts test

```verbatim
module test_top;
import uvm_pkg::*;
import my_test_pkg::*;

my_dut_interface my_dut_if();
my_dut_rtl my_dut(.if(my_dut_if());

initial begin
    uvm_config_db #(virtual my_dut_interface)::set(null,
                "uvm_test_top", "DUT_IF", my_dut_if);
    run_test();
end
endmodule
```

UVM Constructs

<table>
<thead>
<tr>
<th>Classes</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Methods</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>Macros</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Stu Sutherland, Sutherland-HDL & Tom Fitzpatrick, Mentor Graphics
The Base Test

- Test instantiates & configures the environment

```cpp
class my_test extends uvm_test;
    `uvm_component_utils(my_test)
    my_env m_env;
    my_env_config_obj m_env_cfg;
    ...

    function void build_phase(uvm_phase phase);
        m_env_cfg = my_env_config_obj::type_id::create("m_env_cfg");
        m_env = my_env::type_id::create("my_env", this);
        if(!uvm_config_db#(virtual my_dut_interface)::get(this, "", "DUT_IF", m_env_cfg.dut_if))
            `uvm_error("TEST", "Failed to get virtual intf in test")
        // set other aspects of m_env_cfg
        uvm_config_db#(my_env_config_obj)::set(this, "my_env", "m_env_cfg", m_env_cfg);
    endfunction
```
The Extended Test

- The extended test specializes the base test

```verilog
class my_ext_test extends my_test;

`uvm_component_utils(my_ext_test)
...

function void build_phase(uvm_phase phase);
    my_env::type_id::set_type_override(my_env2::get_type());
    my_comp::type_id::set_inst_override(my_comp2::get_type(),
                                         "top.env.c2");
    // optionally override type of my_env_cfg object
    super.build_phase(phase);
    // optionally make additional changes...
endfunction
```

UVM Constructs | First Time Seen | Running Total
---------------|----------------|---------------
Classes        | 0              | 6             |
Methods        | 4              | 16            |
Macros         | 0              | 6             |
The Extended Test

- The test starts sequences and manages objections

```verilog
class my_ext_test extends my_test;
    `uvm_component_utils(my_ext_test)
    ...

task run_phase(uvm_phase phase);
    phase.raise_objection("Starting test");
    my_seq seq = my_seq::type_id::create("seq");
    // optionally randomize sequence
    assert(seq.randomize() with {src_addr == 32'h0100_0800;
                              xfer_size == 128;});
    seq.start(m_env.m_agent.m_sequencer);
    phase.drop_objection("Ending test");
endtask
```

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Methods</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>Macros</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
UVM Constructs Used By The Sequence Writer
The Sequence Writer

• Each sequence defines stimulus and/or response functionality
• Provide list of sequence types and sequencer types to start them on
• Inheritance hierarchy and other details irrelevant to Test Writer
Designing a Sequence Item

```verilog
class my_tx extends uvm_sequence_item;
`uvm_object_utils(my_tx)
    rand bit [23:0] operand_a;
    rand bit [23:0] operand_b;
    randc opcode_t opcode;
    logic [23:0] result;

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

do_copy()
    do_compare()
    convert2string()
    do_record()
    do_pack()
    do_unpack()
endclass: my_tx
```

“Input” variables should be `rand`

“Output” variables should not be

Standard Object constructor

User calls `copy()`, `compare()`, etc.

Alternately use `uvm_field_xxx` macros (73) to auto-generate the do_ methods

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Methods</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>Macros</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
The Sequence Body Method

- The **body** method defines the **transactions** to generate

```verilog
class tx_sequence extends uvm_sequence#(my_item);
  `uvm_object_utils(tx_sequence)
  ...
  task body();
    repeat(50) begin
      tx = my_seq_item::type_id::create("tx");
      start_item(tx);
      ...
      finish_item(tx);
    end
  endtask
endclass:tx_sequence
```

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Methods</td>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td>Macros</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>
The Virtual Sequence

- The virtual sequence starts subsequences

```cpp
class my_vseq extends uvm_sequence#(uvm_sequence_item);
    ...
    bus_sequencer_t bus_sequencer;
    gpio_sequencer_t gpio_sequencer;

    virtual function void init(uvm_sequencer bus_seqr,
                                 uvm_sequencer gpio_seqr);
        bus_sequencer = bus_seqr;
        gpio_sequencer = gpio_seqr;
    endfunction

    task body();
        aseq.start( bus_sequencer , this );
        bseq.start( gpio_sequencer , this );
    endtask
endclass
```

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Methods</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>Macros</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>
UVM Constructs Used For Advanced Examples
• Delay the end of a phase when necessary

function void my_comp::phase_ready_to_end( uvm_phase phase );
    if( !is_ok_to_end() ) begin
        phase.raise_objection( this , "not ready to end phase" );
        fork begin
            wait_for_ok_end();
            phase.drop_objection( this , "ok to end phase" );
        end
        join_none
    end
endfunction : phase_ready_to_end
Pipelined Protocols

- Use the Response Handler in the sequence

```verilog
class my_pipelined_seq extends uvm_sequence #(my_seq_item);
    `uvm_object_utils(my_pipelined_seq)
    ...
    task body();
        my_seq_item req = my_seq_item::type_id::create("req");
        use_response_handler(1);
        ...
        start_item(req);
        ...
        finish_item(req);
    endtask

    function void response_handler(uvm_sequence_item_item response);
        ...
    endfunction
endclass: my_pipelined_seq
```

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Methods</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>Macros</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>
Pipelined Protocols

- Driver uses one thread per pipeline stage

```vhdl
class my_pipelined_driver extends uvm_driver #(my_seq_item);
  `uvm_component_utils(my_pipelined_driver)

  task do_pipelined_transfer;
    my_seq_item req;
    forever begin
      pipeline_lock.get();
      seq_item_port.get(req);
      ...// execute first pipeline stage
      pipeline_lock.put();
      ...// execute second pipeline stage
      seq_item_port.put(req);
    end
  endtask
endclass: my_pipelined_seq
```

<table>
<thead>
<tr>
<th>UVM Constructs</th>
<th>First Time Seen</th>
<th>Running Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Methods</td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>Macros</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>
UVM Features to Avoid

- Phase Jumping
- Callbacks
- UVM 1.2 features
The Solution…

• The UVM 1.2 Library has 357 classes, 938 functions, 99 tasks, and 374 macros

• Our recommended subset in the paper uses 10 classes, 30 tasks/functions and 6 macros

• You really only need to learn 3% of UVM to be productive!
  – 2% of classes
  – 3% of methods