2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

UVM-Light
A Subset of UVM for Rapid Adoption

Stuart Sutherland, Sutherland HDL, Inc., Portland, Oregon, USA (*stuart@sutherland-hdl.com*)

Tom Fitzpatrick, Mentor Graphics Corp., Wilsonville, Oregon, USA (*tom_fitzpatrick@mentor.com*)

***Abstract* — The Universal Verification Methodology (UVM) 1.2 standard defines 357 classes, 1037 methods and 374 macros. This very large library can make UVM difficult to learn, and can lead to radically different coding styles that are difficult to maintain. Just how much of the UVM library is really needed to code effective verification testbenches? This paper identifies a subset of the UVM base classes, methods and macros that will enable engineers to learn UVM more quickly and become productive with using UVM for the verification of most types and sizes of digital designs modeled in VHDL, Verilog or SystemVerilog. You might be surprised at just how small a subset of UVM is really needed in order to verify complex designs effectively with UVM.**

Keywords—SystemVerilog; UVM; verification; ASIC; FPGA

1.0 INTRODUCTION

The Universal Verification Methodology (UVM) is purposely designed to be capable of verifying all types of digital logic designs: simple or complex, large or small, processor or data or control oriented. This universal nature means there are constructs and capabilities in the methodology that might not be necessary for a specific project. UVM leveraged ideas from other methodologies, such as OVM, VMM and *e*, and has inherited overlapping features from these methodologies, further complicating UVM. The nature of Object Oriented Programming adds to this complexity. Only some of the base classes in UVM are intended to be used by those who write UVM testbenches. These end-user classes extend and inherit from other base classes. The UVM Class Reference Manual does not identify which classes are intended for end users and which classes are for internal use within the UVM.

The goal of this paper is to simplify learning and adopting UVM by suggesting preferred choices when redundancy exists, and clarifying which classes are for end-use of UVM. Nearly all verification projects can be achieved with a simple subset of UVM classes, methods and macros. This "UVM-light" subset will help make UVM easier to learn, easier to maintain, easier to re-use, and more efficient for simulation. This recommended subset is based on UVM 1.2, but only uses constructs that are compatible with UVM 1.1.

This paper examines UVM from three perspectives: the roles of the ***Test Writer***, the ***Environment Writer*** and the ***Sequence Writer***. A simple UVM testbench is used to show what UVM constructs these three types of verification engineers need to know. The paper then considers some advanced verification situations, and discusses any additional UVM constructs needed to handle those more difficult testbenches.

*This is a "**what-to-use**" paper, not a "how-to-write" paper!* This paper is not a tutorial and is not a reference manual for the definition, syntax and semantics of UVM constructs. Rather, this paper focusses on *what* UVM constructs are really needed to write effective UVM tests, environments, and stimuli for most verification projects.

Note: This paper is a revision of a paper presented at DVCon United States in March 2015 [1]. The United States version contains a more in depth description of the code that utilizes the constructs shown in this paper.

2.0 TEST WRITER GUIDELINES

The ***Test Writer*** is responsible for specifying stimuli and results checking, in order to verify a particular set of features in the Device Under Test (DUT). The ***Test Writer*** utilizes the environments and sequences written by the *Environment Writer* and *Sequence Writer*. UVM is set up so that the ***Test Writer*** can specify useful tests without knowing the underlying details of the environment, or of UVM itself. From the perspective of the ***Test Writer***, the UVM environment is mostly a black box. In this section, we will discuss the specific tasks relegated to the ***Test Writer*** and show which UVM constructs are required by the ***Test Writer***.

2.1 Connect to the DUT

The ***Test Writer*** connects the class-based UVM testbench to the module-based DUT in a top-level module. This is done using the SystemVerilog `interface` construct. The top-level module: 1) Imports the UVM package that contains the UVM class library, and the packages that contain the user-defined UVM testbench code. 2) Instantiates the DUT and the interfaces used to connect the UVM testbench to the DUT. 3) Stores the interface instance names, referred to as virtual interfaces, in a UVM data base. 4) Calls a task to start UVM running.

The only UVM constructs the ***Test Writer*** needs to know to code a top-level module are: **`uvm_config_db::set()`**, **`run_test()`**, and importing the **`uvm_pkg`** package.

Example 1 shows the code for a UVM top-level module with a single interface. **Note:** Throughout this paper, UVM-specific code is shown in **bold text**. UVM constructs that have not been used in previous examples are shown as **highlighted text**. Refer to [1] for a more detailed explanation of the code examples in this paper, as well as best-practice recommendations for writing UVM testbenches.

```
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
```

Example 1 – Top-level module

2.2 UVM tests

A UVM test has several basic responsibilities: 1) Get the virtual interface handle(s) from the configuration database. 2) Instantiate the UVM environment. 3) Use the configuration database to pass the virtual interface handle(s) and other information down to the environment. 4) Instantiate and start any sequences necessary for the test case. 5) Manage phase objections, to ensure the test successfully completes.

The UVM constructs not already listed in this paper required to write a UVM test are: uvm_test, build_phase(), run_phase(), uvm_config_db::get(), component_type>::create(), start(), raise_objection(), drop_objections(), `uvm_component_utils, `uvm_error .

Example 2 shows the code for a UVM test that builds an environment and runs a test.

```
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  function new(string name = "my_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  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 interface 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

  task run_phase(uvm_phase phase);
    ... // details of the run_phase are omitted from this example
endclass
```

Example 2 – A UVM test class template

2.3 Base tests and extended tests

The environment itself is simply a UVM component, so it is created from the factory, just as any other component would be. It is often the case that basic operations like instantiating and configuring the environment are done in a "base test", which is then extended to allow additional customization for specific testing objectives. In this case, we may extend the base test and use the factory to instantiate a different environment.

Only three additional UVM constructs not already listed in this paper are required for an extended test to override the environment specified in a base test: **`set_type_override()`**, **`get_type()`**.

```
class my_extended_test extends my_test;
  `uvm_component_utils(my_extended_test)
  ... // new() constructor and declarations are omitted from this example

  function void build_phase(uvm_phase phase);
```

```
    my_env::type_id::set_type_override(my_env2::get_type());
    // optionally override type of my_env_cfg object
    super.build_phase(phase);
    // optionally make additional changes to my_env_cfg object
  endfunction
```

Example 3 – An extended UVM test that overrides the environment type

3.0 ENVIRONMENT WRITER

The ***Environment Writer*** is responsible for getting stimulus generated by the test into the DUT, and verifying that the DUT responds correctly to that stimulus. To accomplish this, the ***Environment Writer*** defines a UVM *environment*, *agent* and *scoreboard*. The agent is further divided into subcomponents of a *sequencer*, *driver*, *monitor* and, optionally, a *coverage collector*.

3.1 UVM environments

A UVM *environment* is a component that is created and configured by a UVM test. The environment drives stimulus into the DUT, monitors the inputs and outputs of the DUT, and verifies that the DUT behaves as intended. A UVM environment instantiates the structural aspects of a UVM testbench, including:

- An agent, which handles driving DUT inputs and monitoring DUT input and output activity.
- A scoreboard, which handles verifying DUT responses to stimulus
- A configuration component, which allows the test to set up the environment for specific test requirements.
- Optionally, a coverage collector, which records transaction information for coverage analysis

A more complex UVM environment might contain multiple agents, a register model, or other UVM components. This is discussed later in this paper in Section 5.0, Advanced Examples. These more complex environments require only a few additional UVM constructs than the short set of constructs used in a basic UVM environment.

UVM constructs not already listed in this paper required to write a UVM environment are: **`uvm_env`**, **`connect_phase()`**, <port_handle>**`.connect()`**.

Example 4 illustrates the code for a simple UVM environment with a single agent and scoreboard.

```
class my_env extends uvm_env;
  `uvm_component_utils(my_env)
  ... // new() constructor and declarations are omitted from this example

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

  function void connect_phase(uvm_phase phase);
    agent.dut_inputs_port.connect(scoreboard.dut_in_imp_export);
    agent.dut_outputs_port.connect(scoreboard.dut_out_imp_export);
  endfunction: connect_phase

endclass: my_env
```

Example 4 – UVM environment

3.2 UVM agents

A UVM *agent* is a low-level building block that is associated with a specific set of DUT I/O pins and the communication protocol for those pins. For example, the SPI bus to a DUT will have an agent for that set of ports. Likewise, the APB bus will have an agent specific for that bus. An *agent* contains three required components: a *sequencer*, *driver* and *monitor*. In addition, agents may contain an optional *coverage collector* component.

UVM constructs not already listed in this paper required to write a UVM agent are: **`uvm_agent`**, **`uvm_analysis_port`**, **`` `uvm_warning``**, **`uvm_active_passive_enum`** type definition.

```
class my_agent extends uvm_agent;
  `uvm_component_utils(my_agent)  // register this class in the factory
  ... // new() constructor and declarations are omitted from this example

  uvm_active_passive_enum is_active = UVM_ACTIVE; // default of active

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

  function void build_phase(uvm_phase phase);
    if (uvm_config_db #(my_config)::get(this, "", "test_config", m_config))
      this.is_active = this.m_config.is_active;
    else `uvm_warning("LAB", Failed to access config_db -- using defaults.\n")
```

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

  function void connect_phase(uvm_phase phase);
    // set agent's analysis ports to point to the monitor's ports
    dut_inputs_port  = mon.dut_inputs_port;
    dut_outputs_port = mon.dut_outputs_port;
    if (this.is_active == UVM_ACTIVE) begin
      drv.seq_item_port.connect(sqr.seq_item_export); // connect driver/sequencer
    end
  endfunction: connect_phase
endclass: my_agent
```

Example 5 – A typical UVM agent

3.3 UVM Sequencers

A *sequencer* serves as a router of sequence_items (transactions). The sequencer can receive sequence_items from any number of sequences (stimulus generators) and route these items to the agent's driver. Sequencers are extended from the `uvm_sequencer` base class, and inherit all necessary routing and arbitration functionality from this base class. Since the `uvm_sequencer` base class functionality does not need to be extended, it is possible to use the base class directly within an agent, by simply defining the sequencer's type parameter to a specific sequence_item type.

Only one UVM construct is required to code a UVM sequencer: **`uvm_sequencer`**.

```
typedef uvm_sequencer #(my_tx) my_sequencer;
```

Example 6 – Defining a sequencer using typedef

3.4 UVM Drivers

A UVM driver requests a handle to a sequence_item object from its associated sequencer, and assigns values in the sequence_item properties to corresponding signals in a SystemVerilog interface, thus driving the DUT inputs.

The UVM constructs not already listed in this paper required to write a UVM driver are: **`uvm_driver`**, <port_handle>**`.get_next_item()`**, <port_handle>**`.item_done()`**, **`` `uvm_fatal ``**.

Example 7 illustrates the UVM driver for the simple example used in this paper.

```
class my_driver extends uvm_driver #(my_tx);
  `uvm_component_utils(my_driver)
  ... // new() constructor and declarations are omitted from this example

  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 to get virtual interface from uvm_config_db.\n")
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    my_tx tx;
    forever begin
      seq_item_port.get_next_item(tx);  // get a transaction
      tb_vif.drive(tx);  // call task in interface to drive values stored in tx
      seq_item_port.item_done();
    end
  endtask: run_phase
endclass: my_driver
```

Example 7 – A UVM driver

3.5 UVM Monitors

A UVM monitor observes the DUT inputs and outputs for a specific interface, captures the observed values into one or more sequence_items, and broadcasts handles to those sequence_items to other UVM components (such as a scoreboard and a coverage collector).

The UVM constructs not already listed in this paper required to write a UVM monitor are: **`uvm_monitor`**, **`write()`**.

Example 8 shows the code for a basic UVM monitor.

```
class my_monitor extends uvm_monitor;
  `uvm_component_utils(my_monitor)
  ... // new() constructor and declarations are omitted from this example

  uvm_analysis_port #(my_tx) dut_inputs_port;  // TLM port for DUT inputs
  uvm_analysis_port #(my_tx) dut_outputs_port; // TLM port for DUT outputs

  function void build_phase(uvm_phase phase);
    dut_inputs_port  = new("dut_inputs_port", this);  // construct the port
    dut_outputs_port = new("dut_outputs_port", this); // construct the port
    if (!uvm_config_db #(virtual my_dut_interface)::get(this,"","DUT_IF",tb_vif))
      `uvm_fatal("NOVIF", Failed to get virtual interface from uvm_config_db.\n")
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    my_tx tx_in, my_tx tx_out, tx_copy;
    fork
      forever begin
        tx_in  = my_tx::type_id::create("tx_in");
        tb_vif.monitor_inputs(tx_in); // call task in interface to read DUT pins
        dut_inputs_port.write(tx_in);
      end
      forever begin
        tx_out = my_tx::type_id::create("tx_out");
        tb_vif.monitor_inputs(tx_out); // call task in interface to read DUt pins
        dut_outputs_port.write(tx_copy);
      end
    join
  endtask: run_phase
endclass: my_monitor
```

Example 8 – A simple UVM monitor

3.6 UVM Coverage Collectors

Functional coverage is an important aspect of verifying complex designs, especially when using constrained random verification. The recommended best practice is to define a UVM component, referred to as a *coverage collector*, to contain the SystemVerilog coverage definitions for variables of interest. The UVM agent is an ideal location for instantiating a coverage collector for the DUT inputs being observed by that agent's monitor. Coverage collector components can also be instantiated in the environment, which can be useful for collecting cross coverage between multiple agents.

The UVM constructs not already listed in this paper required to write a UVM coverage collector are: **`uvm_subscriber`**, **`write()`** (called by the monitor), **`get_full_name()`**,**`report_phase()`**, **`uvm_info`**.

A typical coverage collector component is shown in Example 9.

```
class my_coverage_collector extends uvm_subscriber #(my_tx);
  my_tx tx;  // the transaction object on which value changes will be covered

  covergroup dut_inputs;
    ... // define coverpoints and bins
  endgroup

  `uvm_component_utils(my_coverage_collector)
  ... // new() constructor and declarations are omitted from this example

  function void write(my_tx t);
    tx = t;  // copy transaction handle received from the monitor
    dut_inputs.sample();
  endfunction: write

  function void report_phase(uvm_phase phase);
    `uvm_info("DEBUG", $sformatf("\n\n Coverage for instance %s = %2.2f%%\n\n",
              this.get_full_name(), this.dut_inputs.get_inst_coverage()), UVM_HIGH)
  endfunction: report_phase
endclass: my_coverage_collector
```

Example 9 – A simple coverage collector

3.7 UVM Scoreboards

A UVM *scoreboard* is used to verify that actual DUT outputs match predicted output values. How expected results are predicted and DUT outputs verified is SystemVerilog programming, and left up to the requirements of the DUT verification and the creativity of the ***Environment Writer***. The prediction and verification functionality is encapsulated into a UVM component that is instantiated at the environment level. The environment can then

connect the scoreboard component to one or more agents, which gives the scoreboard component access to the DUT input and output values observed by these agents.

The scoreboard component can have hierarchy within it, where the prediction algorithms and verification algorithms are defined in classes that are instantiated with the scoreboard. The functionality of a relatively simple scoreboard can be coded directly in the scoreboard component without using any hierarchy). A scoreboard with a flat hierarchy can encounter a limitation in the SystemVerilog language, which is that SystemVerilog does not permit a class to have two methods with the same name (i.e.: SystemVerilog does not have function overloading). With a flat scoreboard hierarchy, however, it is often necessary to have two or more `write()` methods, each implementing the functionality for a different port. UVM handles this language limitation with a macro that appends some characters to the name of a `write()` method and its corresponding port class type, making those names unique.

Only one additional UVM construct not already listed in this paper is required to write a UVM scoreboard, and that is only needed if the scoreboard has a flat hierarchy: **`uvm_anaylysis_imp_decl**.

Example 10 shows a basic scoreboard component with a flat hierarchy. The actual code is omitted for the `build_phase()`, `report_phase()` and the two `write()` methods (one with a unique name set by the macro), as this code does not require any additional UVM constructs beyond those previously shown in this paper.

```
`uvm_analysis_imp_decl(_verify_outputs)

class my_scoreboard extends uvm_subscriber #(my_tx);
  `uvm_component_utils(my_scoreboard)
  ... // new() constructor and declarations are omitted from this example

  uvm_analysis_imp_verify_outputs #(my_tx, my_scoreboard) dut_out_imp_export;

  // implement the write() method called by the monitor for observed DUT inputs;
  function void write (my_tx t);
    ... // predict what the DUT results should be for a set of input values
  endfunction: write

  // implement the write() method called by the monitor for actual DUT outputs;
  function void write_verify_outputs(my_tx t);
    ... // compare the DUT outputs to the predicted results
  endfunction: write_verify_outputs

  function void report_phase(uvm_phase phase);
    ... // print a summary report of the scoreboard results
  endfunction: report_phase
endclass: my_scoreboard
```

Example 10 – A simple UVM scoreboard component (flat hierarchy)

4.0 SEQUENCE WRITER

The ***Sequence Writer*** is responsible for supplying the *Test Writer* with a set of sequences that define stimulus and response functionality to be used in a particular test case. In keeping with the modularity goals of UVM, all the ***Sequence Writer*** needs to provide to the *Test Writer* is a list of sequence types and enough information so that the *Test Writer* knows on which sequencer(s) to start them. The *Test Writer* need not know the inheritance hierarchy, or even the details of what is in the sequence.

4.1 Defining a sequence_item

The basic unit of communication in a UVM testbench is the `uvm_sequence_item`, often referred to as a *transaction*. A sequence_item encapsulates the fields and methods necessary to pass information between the UVM components, such as from a monitor to the scoreboard. The UVM infrastructure depends on sequence_items implementing a standard set of methods, such as `copy()` and `compare()`. These methods can be implemented by writing the code for a set of "`do_`" counterpart methods (e.g.: `do_copy()`, `do_compare()`, etc.), or through the use of "field macros".

The UVM constructs not already listed in this paper needed to write a UVM sequence_item are: **`uvm_sequence_item`**, **`do_copy()`**, **`do_compare()`**, **`do_record()`**, **`do_pack()`**, **`do_unpack()`**, **`convert2string()`**, **`uvm_object_utils`**. (Sequence_items also have `print()` and `sprint()` methods, but the authors recommend using convert2string() to print the values of sequence_item contents.)

Example 11 illustrates a simple sequence_item definition. The implementation of the `do_` methods is not shown in this example. Information on implementing the `do_` methods can be found in [4], [7] and [8].

```
class my_tx extends uvm_sequence_item;
  rand   bit   [23:0] operand_a;
  rand   bit   [23:0] operand_b;
```

```
  randc  opcode_t     opcode;
         logic [23:0] result;

  `uvm_object_utils(my_tx)

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

  virtual function void do_copy(uvm_object rhs);
    ... // implementation not shown
  endfunction

  virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    ... // implementation not shown
  endfunction

  ... // implementation of remaining do_ methods
endclass: my_tx
```

Example 11 – A UVM sequence_item (aka "transaction")

An alternative to writing the `do_` methods is to use a set of `` `uvm_field_ `` macros, but the resultant code is inefficient, hard to debug, and can be prone to error. The authors of this paper differ in opinion on whether it is preferred to use field macros or manually write the implementation of these methods. We agree, however, that manually coded `do_` methods will improve simulation performance. Sutherland feels the simplicity of the field macros is worth the performance trade-off, and to only manually write a `do_` method if performance profiling shows a specific field macro code is causing a performance problem. Fitzpatrick feels the impact on performance can be significant, and it is, therefore, worth investing the time up front to manually write the `do_` methods.

4.2 Sequences

A UVM *sequence* generates one or more sequence_items. The handles to these sequence_items are routed to a driver through a sequencer. The heart of a UVM sequence is the `body()` method, which is a user-defined method that defines the behavior of the sequence.

The UVM constructs required to write a typical sequence that are not already listed in this paper are: **`uvm_sequence`**, **`body()`**, **`start_item()`**, **`finish_item()`**.

A simple UVM sequence is shown in Example 12. Section 5.0 discusses more complex sequences and the small number of additional UVM constructs that might be needed.

```
class tx_sequence extends uvm_sequence #(my_item);
  `uvm_object_utils(tx_sequence)
  ... // new() constructor and declarations are omitted from this example

  task body();
    repeat(50) begin
      tx = my_seq_item::type_id::create("tx");
      start_item(tx);
      ... // randomize or set the values of the sequence_item variables
      finish_item(tx);
    end
  endtask
endclass: tx_sequence
```

Example 12 – A basic UVM sequence

4.3 Virtual sequence

A *virtual sequence* is used to coordinate the execution of other sequences on specific sequencers. The virtual sequence includes sequencer handles that will be set to point to the specific sequencers in the environment. How the virtual sequence handles are initialized is SystemVerilog programming and does not require any UVM constructs. An example of a virtual sequence is not shown in this paper, as no additional UVM constructs beyond those already listed in this paper are required. Refer to [9] for more information on virtual sequences.

5.0 ADVANCED EXAMPLES

The preceding sections have outlined the core set of UVM constructs needed to verify the vast majority of designs. There are some situations that may arise which may require the use of a few – very few – more UVM constructs. The most common additional constructs are: 1) **`phase_ready_to_end()`** to give a component, such as the scoreboard, an opportunity to raise an objection again, in order to prevent the phase from ending until that component drops its objection, and 2) a sequence/sequencer/driver handshaking protocol using **`get()`**, **`put()`**, **`response_handler()`** and **`use_response_handler()`** to represent a pipelined bus protocol[5] that allows several transfers to be in-progress at the same time, with each transfer occupying one stage of the pipeline.

6.0 CONCLUSION

The UVM Committee is to be congratulated for producing such an extensive and useful implementation of the UVM class library. UVM delivers on its goals of enabling the creation of modular reusable verification components, environments and tests. However, it is important to realize that when using UVM, most of the class library is dedicated to infrastructure and support of the relatively small set of features that ***Test Writers***, ***Environment Writers*** and ***Sequence Writers*** actually use. Indeed, ***the set of features identified in this paper consists of 11 classes, 38 methods and 7 macros that end users need to be familiar with***, in order to use UVM (in addition to proper SystemVerilog coding and general use-model approaches). When compared to the 357 classes, 1037 unique methods (938 functions and 99 tasks) and 374 macros that comprise UVM 1.2, ***UVM users really only need to learn 3% of UVM to be productive***!

Note: The intent of this paper is to show that effective UVM testbenches can be written with a very small subset of the classes, methods and macros defined in the UVM Class Reference Manual. Some engineers might prefer using a few additional UVM constructs. Even if this subset is extended, what is important, and has been shown in this paper, is that end-users of UVM only need to learn a small subset of UVM.

7.0 REFERENCES

[1] Sutherland and Fitzpatrick, "UVM Rapid Adoption: A Practical Subset of UVM", DVCon United States 2015, http://proceedings.dvcon.org/proceedings/content/getDownload.aspx?filename=12_1_finalpaper.pdf

[2] van der Schoot, Saha, Garg and Suresh, "Off To The Races With Your Accelerated SystemVerilog Testbench", http://events.dvcon.org/2011/proceedings/papers/05_3.pdf

[3] Arunachalam, "Controlling On-the-Fly-Resets in a UVM-Based AXI Testbench", http://www.mentor.com/products/fv/verificationhorizons/volume10/issue3/on-the-fly-resets

[4] Mentor Graphics, Online UVM Cookbook, "Transactions/Methods", https://verificationacademy.com/cookbook/transaction/methods

[5] Mentor Graphics, Online UVM Cookbook, "Driver/Pipelined", https://verificationacademy.com/cookbook/driver/pipelined

[6] Fitzpatrick, T. and Rich, D., "Of Camels and Committees: Standards Should Enable Innovation, Not Strangle It", DVCon 2014 Proceedings, http://events.dvcon.org/2014/proceedings/papers/08_1.pdf

[7] Erickson, "Are OVM & UVM macros Evil? A Cost-Benefit Analysis", DVCon 2011 proceedings. Alternate url: http://www.mentor.com/products/fv/verificationhorizons/horizons-jun-11

[8] Cummings, "UVM Transactions: Definitions, Methods, and Usage", SNUG Silicon Valley 2014 proceedings. Alternate url: http://www.sunburst-design.com/papers/CummingsSNUG2014SV_UVM_Transactions.pdf

[9] Mentor Graphics, Online UVM Cookbook, "Sequences/Virtual", https://verificationacademy.com/cookbook/sequences/virtual

8.0 APPENDIX

This appendix summarizes the practical subset of UVM recommended in this paper. These are the relatively small number of UVM constructs that end users of UVM need to know, in order to be effective with UVM.

UVM Classes (11 constructs): `uvm_analysis_port`, `uvm_analysis_imp_export`, `uvm_agent`, `uvm_driver`, `uvm_env`, `uvm_monitor`, `uvm_sequence`, `uvm_sequence_item`, `uvm_sequencer`, `uvm_subscriber`, `uvm_test`

UVM Method Definitions (15 constructs): `build_phase()`, `connect_phase()`, `run_phase()`, `report_phase()`, `body()`, `convert2string()`, `do_compare()`, `do_copy()`, `do_pack()`, `do_record()`, `do_unpack()`, `phase_ready_to_end()`, `response_handler()`, `write ()`, `new()` (for uvm_objects and uvm_components)

UVM Method Calls (23 constructs): `run_test()`, `super.build_phase()`, `raise_objection()`, `drop_objection()`, `get_type()`, `get_full_name()`, `start_item()`, `finish_item()`, set_type_override(), set_inst_override(), use_response_handler(), uvm_config_db::set(), uvm_config_db::get(), <component_type>`::type_id::create()`, <object_type>`::type_id::create()`, <port_h>`.connect()`, <port_h>`.get_next_item()`, <port_h>`.item_done()`, <port_h>`.get()`, <port_h>`.put()`, <port_h>`.write()`, <sequence_handle>`.start()`, <virtual_sequencer_handle>`.start()`

UVM Macros (7 constructs): `` `uvm_analysis_imp_decl ``, `` `uvm_component_utils ``, `` `uvm_object_utils ``, `` `uvm_error ``, `` `uvm_fatal ``, `` `uvm_info ``, `uvm_warning`

Miscellaneous (2 constructs): `uvm_pkg` package import, `uvm_active_passive_enum` type definition