Transitioning to UVM from VMM

Chen Jingguo[1], *Chen Yu*[2], *Zhang* Lianbo[1*]

[1]Department of Electronic Information Engineering，BITZH ,ZhuHai, 519000, China
[2]Zhuhai Yizhi Electronic Technology Co., Ltd, ZhuHai, 519000, China

Abstract. This paper discusses the process of transitioning to a UVM design verification environment for current VMM users. Differences and parallels between the two verification methodologies are presented to show that updating to UVM is mostly a matter of learning a new DV syntax. Topics include UVM phases, agents, TLM ports, configuration, sequences, and register models. Best practices and reference resources are highlighted to make the transition from VMM to UVM as painless as possible.

1 Introduction

With the continuous improvement of technology and the increasing complexity of chip design, verification has become a major challenge in chip design, and its proportion in the entire project is also increasing[1][2]. Therefore, improving the verification efficiency of chips has become crucial, and quickly building a powerful, efficient, flexible, and scalable verification platform is the key to successful chip design[3].

The Verification Methodology Manual (VMM) is a validation methodology based on SystemVerilog, launched by Synopsys in 2006. By utilizing the hierarchical and random constraints of VMM, existing validation methods can be effectively improved, and a validation environment with target modules can be quickly built[4].

The Universal Verification Methodology (UVM) was introduced by the Accellera Verification IP Technical Subcommittee in 2011 as the first industry standardized verification methodology. it adopts a validation platform development framework mainly based on the SystemVerilog class library, inheriting the advantages of VMM and OVM[3][5]. The validation environment built with reusable components has standardized and hierarchical characteristics[6][7], and has got full support from the world's three major EDA vendor (Cadence, Synopsys, Mentor Graphics)[8]. Due to its ability to provide validation consistency across projects, teams, companies, and simulators, it has become the most widely used validation methodology in the industry. Verification engineers within the Synopsys Users Group have been using the Verification Methodology Manual (VMM) since its release in 2005. Over the years these engineers have become VMM experts and may be hesitant to switch to a new methodology.

This paper discusses the differences between the two methodologies for design verification(DV) engineers who are considering switching from VMM to UVM. UVM verification components are presented as either parallel equivalents to VMM components or as new features. The main take-away from this paper should be that most of the differences between VMM and UVM are a matter of syntax. If a verification engineer already understands the concepts behind VMM, then it should be fairly straightforward to create a UVM testbench.

2 Parallels between VMM and UVM

Both VMM and UVM are based on the basic validation methodology principles of the SystemVerilog self-test platform, and are validation architectures driven by test coverage [4][8]. The architecture of VMM and UVM testbench is largely the same, but there are several important differences that need to be understood before migrating to UVM. This section discusses these differences and focuses on providing parallel comparisons between the two methodologies. Table 1 below summarizes how many of the fundamental verification concepts are implemented in VMM and shows their equivalent UVM class implementations[9-11].

Table 1. Verification concepts in VMM and UVM

Verification Concept	VMM Base Class	UVM Base Class
Environment	vmm_env	uvm_env
Data Object	vmm_data	uvm_sequence_item
Data Connectors	vmm_channel, mailbox, vmm_xactor_callback	uvm_port_base
Generator	vmm_xactor	uvm_sequencer
Driver	vmm_xactor	uvm_driver
Monitor	vmm_xactor	uvm_monitor
Scenarios	vmm_ms_scenario	uvm_sequence
Tests	vmm_test	uvm_test
Register Model	RAL	uvm_reg_block
Scoreboard	vmm_sb	uvm_scoreboard
Report	vmm_log	uvm_report_object

* Corresponding author:1602710469@qq.com

2.1 Environment and Phases

In UVM, the testbench execution flow is controlled through simulation phases. These phases are very similar to the simulation phases in VMM1.2. However, VMM1.1 uses the environment methods (`gen_cfg`, `build`, `reset_dut`, etc.) to control simulation flow, so users migrating from that version will notice a significant conceptual difference.

Table 2 shows the UVM simulation phases[10]. These phases are run in order and are common to any object extended from `uvm_component`. This means that the simulation will wait for all `build_phase` functions to complete in all testbench components before starting the `connect_phase`.

The `run_phase` task is the only phase that is allowed to consume simulation time, so most of the action will happen there. It is not necessary to define all of the phases for each component; the most frequently used phases are `build_phase`, `connect_phase`, and `run_phase`.

The phase sequence is kicked off by calling the global `run_test` task from an initial statement in the top-level testbench module or program block.

Table 2. UVM Common Simulation Phases

Phase Name	Execution Steps
build_phase	Create and configure testbench structure
connect_phase	Establish cross-component connections
end_of_elaboration_phase	Fine-tune the testbench
start_of_simulation_phase	Get ready for DUT to be simulated
run_phase (task)	Simulate the DUT
extract_phase	Extract data from different points in the environment
check_phase	Check for any unexpected conditions in the environment
report_phase	Report results of the test
final_phase	Tie up loose ends

2.2 Data object

Data object classes in VMM and UVM are fairly similar. Like VMM, UVM has its own object utility macros, which allow for automatic creation of the following data methods for each data member: copy, compare, pack, unpack, record, print, and sprint. UVM utility macros are a great time-saver and can be customized using the macro flags (see the reference [15] for syntax details). If necessary, the default data methods can be overridden with custom methods by just extending the appropriate function. Example code for a simple UVM object is shown below.

```
class packet extends uvm_sequence_item;
  rand bit [7:0] addr;
  rand bit [7:0] data[8];
  function new(string name="packet");
    super.new(name);
  endfunction
  `uvm_object_utils_begin(packet)
  `uvm_field_int(addr, UVM_ALL_ON)
  `uvm_field_array_int(data, UVM_ALL_ON)
  `uvm_object_utils_end
endclass : packet
```

2.3 Connectors and TLM Ports

In VMM, transactions can be passed between testbench components in a variety of ways. The most common connections occur through channels, mailboxes, and callbacks. UVM replaces all of these connector types with transaction-level modeling (TLM) ports. TLM ports allow for better re-usability because a producer component sending data does not need to know how that data is used by the consumer component. In this way, TLM ports are very similar to VMM callbacks. There are several UVM classes for TLM communication including ports, exports, and implementations. The syntax for each of these classes is detailed in reference[15] and example usage is shown in reference[16].

However, for basic UVM testbenches, it is not really necessary to understand all of the details of the TLM classes, because UVM provides the `uvm_agent` class which automatically handles most of the transaction communication. UVM agents are discussed in detail in Section 3.1. One TLM port concept that is important to understand is analysis ports.

The `uvm_analysis_port` class provides a way to make transaction data available to passive components in other parts of the testbench. Analysis ports are commonly used to send transactions from a monitor to a scoreboard, similar to a VMM callback. Furthermore, it is possible to connect any number of consumer components to a single `uvm_analysis_port`.

2.4 Generator and Scenario

VMM generators and scenarios are replaced by UVM sequencers and sequences, respectively. For atomic test sequences with only one driver these mappings are very straightforward and just a matter of coding in the new syntax.

The mapping continues to be uncomplicated for multiple scenarios or scenario nesting; in UVM it is easy to create multiple nested scenarios. Controlling scenarios for multiple interfaces is somewhat more complex.

In UVM, each interface has its own agent with its own independent sequencer. At the system-level, these sequences need to be coordinated across all interfaces in order to create meaningful tests. This requires using a UVM virtual sequencer, which is a top-level sequencer with handles to each agent's subsequencer. An explanation of how to use a virtual sequencer can be found in reference [16].

2.5 Register Model

The UVM Register Layer was derived from VMM RAL, so the two register models are very similar. The API access methods are identical, but the syntax of the register definition file is different. Synopsys' ralgen tool is able to auto-generate UVM register descriptions by adding the –uvm switch[12]:

```
ralgen -l sv -t top -uvm ral_file.ralf
```

The UVM register model can be built in the testbench uvm_env class and then connected to other

testbench components via direct assignments or by passing the handle through the configuration database (see Section 3.3)Scoreboard.

A final difference between the VMM and UVM register models is how the register API methods get converted to bus transactions. In VMM, this is done by overloading the `vmm_rw_xactor::execute_single` task. In UVM, a register adapter class must be extended from `uvm_reg_adapter`, and the `reg2bus` and `bus2reg` functions must be overloaded. Since the register model sends bus transactions through the UVM sequencer, it is not necessary to force any additional transaction timing (VMM `put/get/notify`).

Once the register adapter class is defined, it needs to be linked to a specific sequencer. This is usually done during the `connect_phase` of the testbench environment, as shown below.

```
virtual function void connect();
if (regmodel.get_parent() == null) begin
    reg2apb_adapter reg2apb = reg2apb_adapter
      ::type_id::create(reg2apb, ,get_full_name
      ());
    regmodel.APB.set_sequencer(apb.sequencer,
      reg2apb);
    regmodel.set_auto_predict(1);
end
```

2.6 Scoreboarding

One area where VMM is clearly ahead of UVM is scoreboarding. VMM provides a full-featured scoreboard class with multiple streams, ordering, and losses[14]. In contrast, the `uvm_scoreboard` class is an empty placeholder extension of `uvm_component`. There are some UVM comparator classes that directly compare two transactions of the same data type, but lack any other features.

There are several options for an improved scoreboard in UVM. First, Synopsys has adapted the VMM Data Stream Scoreboard to work with UVM by allowing the scoreboard to accept any type of class (VMM, UVM, or OVM).

An explanation of the modified VMM scoreboard is posted on VMMCentral[17] and a full example can be downloaded from SolvNet[18]. A second scoreboard alternative is an open-source UVM scoreboard from Paradigm Works [19]. This scoreboard has many useful features and has the advantage of being written in UVM which makes it easy to use in a UVM testbench.

2.7 Reporting

Logging testbench messages is very similar in VMM and UVM. The recommended practice for both methodologies is to use the built-in reporting macros. These macros are listed in Table 3 below.

Table 3. VMM and UVM Reporting Macros

VMM Macros	UVM Macros
`vmm_fatal (log, “msg”)	`uvm_fatal (“id”, “msg”)
`vmm_error (log, “msg”)	`uvm_error (“id”, “msg”)
`vmm_warning (log, “msg”)	`uvm_warning (“id”, “msg”)
`vmm_note (log, “msg”)	`uvm_info (“id”, “msg”, verbosity)
`vmm_trace (log, “msg”)	verbosity inside {UVM_NONE, UVM_LOW, UVM_MEDIUM, UVM_HIGH, UVM_FULL, UVM_DEBUG}
`vmm_debug (log, “msg”)	
`vmm_verbose (log, “msg”)	

There are several syntax differences to note between the two macro styles.

First, in UVM a log class instance is not specified when the macro is called because all UVM messages are sent to a global report server.

Second, UVM macros have a required “id” string field that is printed as a tag for each message in the log file. A best practice for this id field is to use the built-in `uvm_component get_type_name()` method to print out the component’s class name.

Third, the `uvm_info` macro has a built-in verbosity field which can be used to filter which info messages are printed at runtime using the `+UVM_VERBOSITY` option, similar to VMM’s `+vmm_log_default`[13] runtime option.

Finally, when printing variables it is recommended to use the `$sformat` system task inside of the macro instead of `$psprintf, $display, or $write`.

3 Additional UVM Features

3.1 Agents

UVM agents are a convenient way to package together a sequencer, driver, and monitor for a particular interface.

The agent pre-defines the TLM connections between these components and the user can just parameterize the component classes with the desired data object type. This simplifies the process of transaction synchronization between the sequencer and the driver, and enables horizontal re-use across projects. VMM1.1 does not have an equivalent to UVM agents, but VMM1.2 implements this feature using groups. In VMM1.2, the testbench top-level is a group of groups, whereas UVM has an environment composed of agents. Another feature of the UVM agent class is that it can be easily configured to be “active” or “passive.” In passive mode the sequencer and driver are disabled so that only the monitor is turned on. This allows for easy vertical re-use of verification components from the block-level to the chip- and system-levels.

3.2 Factory

UVM provides a built-in factory that allows for run-time configuration of the class type of each testbench component. This makes it possible to have tests modify the actual testbench environment without having to change the testbench code, which enables more flexibility and reuse.

Two additional pieces of code are required in order to take advantage of the UVM factory. First, instead of using the new constructor, data objects must be statically allocated with the factory using the create function:

```
type_name::type_id::create(string name,
uvm_component parent);
```

Second, all components types that might be modified by the factory need to be registered using the UVM utility macro `uvm_component_utils. The uvm_factory itself does not need to be created as it is already instantiated within the UVM package. Once the components are configured to be used with the factory, it is possible to replace one class type with another. A particularly useful application of this is replacing a data object class with an extended class to add test-specific constraints.

3.3 Configuration

Testbench configuration in UVM is handled through a resource database that is globally accessible by any component. Configuration variables (resources) are stored in the database by name and type. The resources can be stored with an optional hierarchical path context to set any variable in any component within the testbench. This eliminates the need to pass a configuration class handle around the testbench, as in a VMM 1.1 testbench.

Configuration resources are stored using the set method:

```
uvm_config_db#(type)::set(context, inst_name,
field_name, value);
```

The inst_name and field_name strings can contain regular expressions. For example:

```
uvm_config_db#(int)::set(this, "*.my_env",
"num_masters", 3);
```

If the field being set was defined using a UVM field utility macro or before `super.build_phase()` then the value will be automatically propagated to the field. If not, it is necessary to call the corresponding get method before the field is used:

```
uvm_config_db#(type)::get(context, inst_name,
field_name, value);
```

Typical examples of using the configuration database include setting the number of masters in an environment or setting the number of iterations in a sequence. Less obvious uses include connecting a virtual interface from the top-level simulation module to the agent:

sim_top:
```
uvm_config_db#(virtual
dut_interface)::set(null,
"uvm_test_top.tb0.env0.agent.*", "vif",
dut_if);
```
driver:
```
uvm_config_db#(virtual
dut_interface)::get(this, "", "vif", vif);
```

or passing a handle to the register model from the testbench environment to a sequence:

tb env:
```
uvm_config_db #(dut_reg)::set(this,
"env0", "cfg_reg_model", reg_model);
```
sequence:
```
uvm_config_db #(dut_reg)::get(null,
"uvm_test_top.tb0.env0", "cfg_reg_model",
model);
```

or configuring a UVM agent to be active or passive:

```
uvm_config_db #(int)::set(this,
"tb0.env0.agent_a2b", "cfg_is_active",
UVM_ACTIVE);
uvm_config_db #(int)::set(this,
"tb0.env0.agent_axi", "cfg_is_active",
UVM_PASSIVE);
```

4 Conclusions

This paper has presented the differences between UVM and VMM to facilitate transitioning verification methodologies. Most of the effort involved in moving to a UVM testbench is learning the new syntax for creating and using the same basic verification components. The Accellera UVM Class Reference [15] and User's Guide [16] are invaluable tools during this process. Other useful UVM references are the UVM World Contributions and Forums. By using these resources and the information provided in this paper, migrating to UVM should be a manageable and straightforward process.

References

1. Su xue, Pan ming, Zhai jiangtao, ALU Verification based on VMM, Modern Electronics Technique, 38(07):144-147, (2015)
2. Li lei, Luo shengqing, SOC integration verification based on VMM, Electronic Measurement Technology, 34(01):128-131, (2011)
3. Lu shuna, Dai xianying, Wang ximing. Research and Engineering Practice on Physical Layer Verification of Communication Systems Based on UVM, Applications of IC, 39(03):114-115, (2022)
4. Duan chengchao, Xu jinfu, Building a Reusable Verification Platform Based on VMM. Modern Electronic Technology, 34(08):127-129, (2011)
5. Ma peng, Liu pei, Zhang wei, Universal Verification Platform for AMBA Bus Interface Based on UVM, Computer Systems Applications, 30(7):57-69, (2021)
6. Du yue, Zheng jieliang, Wu yirang, Design of SoC System Level Peripheral Verification Platform Based on UVM, China Integrated Circuit, 39(03):114-115, (2022)
7. Sun xiaodong, Wang zhiqiang, Typical Structure Design of Integrated Circuit UVM Verification Environment, Journal of Test and Measurement Technology, 36(02):135-140, (2022)
8. Li shuxuan, Bu gang, Han yuxin, Design of a Software-Hardware Co-Verification Platform Based on UVM, Computer Technology and Development, 32(08):76-81, (2022)
9. Chris Spear, SystemVerilog for Verification, Springer, (2006).
10. Liu bing, A Walking Guide to SoC Verification, Electronic Industry Press, (2018)
11. Janick Bergeron, Eduard Cerny, Alan Hunter, Verification Methodology Manual For SystemVerilog, Springer, (2005)
12. Synopsys, Inc., VMM Register Abstraction Layer User Guide, (2011)
13. Synopsys, Inc., UVM Register Abstraction Layer Generator User Guide, (2019)
14. Synopsys, Inc. VMM User Guider[S]. (2011)

15. Accellera Systems Initiative， Universal Verification Methodology 1.2 Class Reference, (2014)
16. Accellera Systems Initiative. Universal Verification Methodology 1.2 User's Guide, (2015)
17. Sharma, Amit, Using the VMM Datastream Scoreboard in a UVM environment, VMM Central, Feb 2, (2012)
18. Using the VMM Datastream Scoreboard in a UVM Environment Using Policy Classes, Synopsys SolvNet. Feb 15, (2012)
19. Paradigm Works, PW UVM Scoreboard Version 1.0, UVM World