Experiences in Migrating a Chip-Level Verification Environment from UVM EA to UVM 1.1

LSI Technologies
RSD HW Verification
Ashish Kumar Ashish.kumar@lsi.com
Dave Stang Dave.Stang@lsi.com
Dudyala Sasidhar Dudyala.Sasidhar@lsi.com
S, Manikandan Manikandan.S@lsi.com
Thirumalai Srishan Srishan.Thirumalai@lsi.com

Abstract— **This paper outlines the challenges we faced in migrating a UVM EA based multi-million gate SOC verification environment to UVM 1.1, and the approaches we followed to address these challenges. The SOC has a variety of blocks like DDR, SAS, PCIe, and many general purpose peripherals like USB, Ethernet and Flash.**

Since UVM EA is very similar from OVM 2.1.x, this paper would be relevant from the perspective of porting OVM 2.1.x environments to UVM 1.1. Following are the topics we plan to address:

1. Backward compatibility of components developed in UVM EA.
2. Migrating UVM EA configuration to resource manager.
3. Synchronization between UVM EA and UVM 1.0 phases
4. Migrating to UVM 1.0 sequences and sequence-libraries.
5. Taking advantage of the command line processor & resource/config db.
6. Taking advantage of the sequence library.
This paper also discusses the challenges and the different approaches used to solve the problems at the SOC.

Keywords: OVM,UVMEA, UVM1.0, Migration, Verification Methodology.

I. INTRODUCTION

The decision to move all the verification environments from various methodologies both home grown ones and standard methodologies like OVM, VMM to UVM was based on the following considerations.

1. Developed and maintained by Accellera and is a standard verification methodology
2. The methodology should be tool independent and open source.
3. Should be able to address verification challenges across generations of chips with no major changes
4. Explore the possiblity of using testbench from a core to a chip(handled across teams).
5. Availablity of 3rdParty IP's and support.
6. And the obvious advantage of reduced ramp up time of new engineers.

Existing verification methodologies like VMM, OVM were explored. Although at the stage when the company was deciding which methodology to be followed, VMM and OVM methodology were not completely tool independent(Some of the constructs were not supported by the eda tools), one was not sure of the future of these methodologies given the fact that UVM was picking up. Unlike other methodologies, UVM is a collective development effort standardized by Accellera. Developed from inputs from both EDA and non-EDA companies, UVM was released only after it has been tested with all the three eda-vendors, unlike OVM or VMM.

II. MOTIVE FOR MOVING FROM UVM-EA TO UVM 1.1

The primary motive for moving UVM-EA to UVM 1.1 is to adopting new features such as phases, sequence library, objections, config db and command line processor, which help both in reducing the effort and maintainability.

At the time when the team decided to move to UVM, development had just started on UVM1.0 and UVM-EA was pretty much OVM2.1.1. Taking above factors into account the decision was made to start the development in UVM-EA at the earliest and subsequently address any changes which would be required from moving UVM-EA to UVM1.1

III. OVERVIEW OF THE DESIGN

It is a complex design with 20m plus gate count. It has on-chip buses like PLB and has SAS/SATA/DDR/PCIe interfaces. There are roughly around 100 directed tests and over 800 complex random threads targeting multiple cores at chip. The random threads use constrained random and coverage driven approaches.

IV. MOVING FROM UVM EA TO UVM1.X

This paper classifies the changes required from moving UVM-EA to UVM-1.X into two categories.

1. Automated changes(or structural changes).
2. Manual or non automated changes.

A. *Automated changes:*

As the team started listing down the changes required to move to newer version of the UVM, it soon became obvious that most of the changes are repetitive and could be automated.

There were deprecated features like `uvm_sequence_utils had to be replaced with other macros. Then there were some changes which were optional, for example, one could still use build() and not switch to build_phase(). However it was deemed better to move in such cases thinking they may get deprecated over -time.
Following are the list of changes which have been automated. Actual code is never deleted, instead the script identifies the piece of code, comments it and adds a new line of code. This helped us to easily identify the code which has been modified. The script, albeit easy to write, helped in automating more than 90 % of the required changes.

1) Configuration uvm_config_db#:

UVM configuration provides a simpler mechanism of sharing resources compared to UVM-EA. For example, uvm_config_db is a parameterized class and can accept any data type including virtual interfaces. Also, the type specific methods get_config_<type> are replaced with a common set/get method. In this section we list the changes which are required for configuration management.

UVM-EA	UVM-1.X
`get_config_int`	`uvm_config_db#(int)::get`
`get_config_string`	`uvm_config_db#(string)::get`
`get_config_object`	`uvm_config_db#(uvm_object)::get`
`set_config_int`	`uvm_config_db#(int)::set`
`set_config_string`	`uvm_config_db#(string)::set`
`set_config_object`	`uvm_config_db#(uvm_object)::get`

Additional manual intervention was required in couple of cases.

a) get_config_int:

In UVM-EA, irrespective of the data type being int/bit, get_config_int works well. In UVM 1.X the parameter used with uvm_config_db should match the defined type. Hence, care needed to be taken if the data type was bit or bit vector. Since the script was used in converting get/set_config_int to uvm_config_db, we had to re-visit the code and change bit or bit_vector to uvm_bitstream_t. Same is true for set_config_int.
Below is an Example code, that explains the above issue

UVM-EA	UVM-1.X

`bit  enable;`
 	
`get_config_int(“my_comp”, enable);`
	`bit enable;`
	
	`uvm_config_db#(`**`uvm_bitstream_t`**`)(“my_comp”, enable);`

b) get_config_object:

In UVM–EA the return type of get_config_object was uvm_object, so one has to type cast it back to the required class type.
In UVM 1.X using config_db get mechanism, objects of the required type are directly set and get, so typecasting is no longer needed. This was not taken care of by the script, and hence required our manual intervention. The table below shows an example of this.

UVM-EA	UVM-1.X

`cfg_class cfg0;`
 	
`function build();`
`   uvm_object tmp0;`
`   cfg0 =cfg_class::type_id::create(“cfg0”);`
 	
`if(`**`get_config_object(get_name(), “cfg0”, tmp0))`**` begin`
`  $cast(cfg0, tmp0);`
`end`
`endfunction`
	`OUTPUT FROM SCRIPT:`
	
	`cfg_class cfg0;`
	
	`function build_phase(uvm_phase phase);`
	`   uvm_object tmp0;`
	`   cfg0 =cfg_class::type_id::create(“cfg0”);`
	
	`if(uvm_config_db#(uvm_object)(get_name(), “cfg0”, tmp0)) begin`
	`  $cast(cfg0, tmp0);`
	`end`
	`endfunction`
	———
	`AFTER MANUAL CHANGES:`
	`function build_phase(uvm_phase phase);`
	`   cfg_class cfg0;`
	
	`if(uvm_config_db#(cfg_class)(get_name(), “cfg0”, cfg0))`` begin`
	`end`
	`endfunction`

c) set_config_string

As a rule, the script changes set_config_string to uvm_config_db#(string). The usage of “default_sequence”, however is an exception. In UVM-1.X uvm_sequence have changed from name based to type based. For this case, the script changes set_config_string(“default_sequence”…) to uvm_config_db#(uvm_object).
Example code, explaining the above change:

UVM-EA	UVM-1.X
`set_config_string(“plb_sequencer”, “default_sequence”, “plb_random_seq”);`	`uvm_config_db#(uvm_object_wrapper)::set(this, “plb_sequencer.main_phase”, “default_sequence”, plb_random_seq::type_id::get());`

2) Phases

UVM-1.X provides a standard way of synchronization between components during the run phase. In order to facilitate run time synchronization between phases, a new class uvm_phase has been introduced in UVM 1.X. Due to this all the standard methods have been redefined with _phase and has an extra argument _phase.

Following are the list of changes which had to be done to comply with the new phasing mechanism. Note: The table below lists the difference between the build method implementation in UVM-EA and UVM-1.X but is also applicable to other standard phases such as connect, start_of_simulation, end_of_elaboration, run, extract, check and report.

UVM-EA	UVM-1.X
`function void build()`	`function void build_phase(uvm_phase phase)`
`function void class::build()`	`class::build_phase(uvm_phase phase)`
`endfunction: build`	`endfunction: build_phase`
`super.build()`	`super.build_phase(phase)`
`task run()`	`task main_phase(uvm_phase phase); phase.raise_objection (this,"main_phase));`

Note:

1. build is not part of deprecated code.(ie. not part of `ifndef UVM_NO_DEPRECATED).
2. The script changes run() to run_phase() everywhere except in testcases. In testcases, the run() method has been changed to main_phase(uvm_phase phase). This is explained under Section:*B Non Automated*.

3) Sequence Utils:

UVM–EA sequence library has been deprecated and new sequence library is introduced which extends from sequence. Hence UVM-EA sequence utils macros are deprecated.
The table below lists the changes done by the script.

UVM-EA	UVM -1.X
`` `uvm_sequence_utils(seq, sequencer) ``	`` `uvm_declare_p_sequencer(sequencer); `uvm_object_utils(seq) ``
`` `uvm_sequence_utils_begin(seq, sequencer) ``	`` `uvm_declare_p_sequencer(sequencer) `uvm_object_utils_begin(seq) ``
`` `uvm_sequence_utils_end ``	`` `uvm_object_utils_end ``
`` `uvm_sequence_param_utils ``	`` `uvm_object_param_utils(seq ) `uvm_declare_p_sequencer(sequencer); ``

4) Sequencer Utils:

The UVM EA sequence library is deprecated. The new UVM sequence library is type based instead of name based, so it does not need any of sequence or sequencer macros.

UVM-EA	UVM-1.X
`` `uvm_sequencer_utils ``	`` `uvm_component_utils ``
`` `uvm_sequencer_utils_(begin|end) ``	`` `uvm_component_utils_(begin|end) ``

5) Deprecated Code:

Script was automated to identify the deprecated code and comment. The table below lists the changes done by the script.

UVM-EA	UVM-1.X
`` `uvm_update_sequence_lib_and_item ``	`//ea_to_10 Commented`
`` `uvm_update_sequence_lib ``	`// ea_to_10 Commented`
`global_stop_request`	`// ea_to_10 Commented`
`set_global_timeout`	`// ea_to_10 Commented`
`set_global_stop_timeout`	`// ea_to_10 Commented`

6) Delays:

In UVM EA there was no standard way of waiting till all the assignments are over in Non Blocking Region on event queue. Typically #0 was used in the testbenches to postpone the event execution. In UVM1.x there is the standard method uvm_wait_for_nba_region() which can be used to achieve this.

UVM-EA	UVM1.0
`#0`	`uvm_wait_for_nba_region();`

7) Miscellaneous:

In addition to above mentioned changes, a few other constructs were changed, either for better performance or for consistency.

Display/Reporter:

UVM-EA	UVM-1.X
`uvm_report_info/error/fatal/warning`	`` `uvm_info/error/fatal/warning ``
`$psprintf`	`` `sformatf ``
`$display`	`` `uvm_info ``
`p_sequencer.uvm_report_info`	`` `uvm_info ``

B. Non-Automated

1) Virtual Interface container:

To pass virtual interfaces, in UVM-EA, we used the commonly used method of wrapping the virtual interface with a container class and pushing this object into the factory using the set config method. In UVM 1-X we used the uvm_config_db, which supports parameter of any type, to pass virtual interfaces.

UVM-EA	UVM-1.X
`container class:` `class vif_wrapper #(type virt_interface_type=int) extends uvm_object;` `virt_interface_type virt_interface_inst;`	

function new(string name="",virt_interface_type intf);
virt_interface_inst = intf;
endfunction
 	
endclass	

Setting virtual interface in config space:
vif_wrapper #(virtual system_if) vif_wrapper_system_if = new("vif_wrapper_system_if" ,system_if0);
 	
set_config_object("*","vif_wrapper_system_if",vif_wrapper_system_if,0);
	Setting virtual interface in config space:
	uvm_config_db #(virtual system_if)::set(null , "*" , "system_if0" , system_if0);

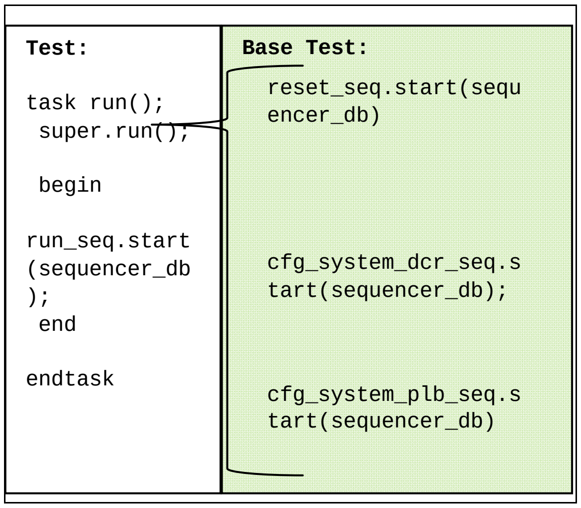
Getting virtual interface from config space :
 	
if(!uvm_top.get_config_object("vif_wrapper_system_if", temp_getcfg_obj,0))
 	
uvm_report_error(get_name() ,"get_config_object on vif_wrapper_system_if failed");
else
 	
assert($cast(vif_wrapper_system_if_h,temp_getcfg_obj)) ;
system_if= vif_wrapper_system_if_h. virt_interface_inst;
	Getting virtual interface from config space :
	
	if(!uvm_config_db#(virtual system_if)::get(this, "", “system_if0” ,system_if0))

2) Tests:

A separate switch was added to script ea_to_10.pl to identify the file which needed modification as a test case.

a) Test Flow in UVMEA:

Before we get into the changes, it would be appropriate to explain the flow of our testcases in existing UVM-EA environment.



```
Test:

task run();
 super.run();

 begin

run_seq.start
(sequencer_db
);
 end

endtask
```

```
Base Test:
  reset_seq.start(sequencer_db)

  cfg_system_dcr_seq.start(sequencer_db);

  cfg_system_plb_seq.start(sequencer_db)
```

Tests run method calls super.run and this in turn executes top reset sequence and top configuration sequences on their corresponding sequences sequentially. Top level configuration sequence in-turn spawns different core configuration sequences either in random cyclic manner or all the threads are forked out.

b) Changes to testcases and base test

There were two options of changing the test cases explored.

1. Change run to run_phase and synchronize the end of main_phase with run phase and the test sequence would remain a part of run_phase.
2. Change the run() to main_phase() and move the reset and configuration sequences under reset_phase() and config_phase() respectively. This would not require any further synchronization between phases. As there are builtin UVM-1.X phases for reset and configuration so Option 2 made sense since there was no need of further user defined synchronization and is taken care by the UVM methodology.

c) Changes:

1. Add a new reset_phase in base test. Move the reset sequences to reset_phase
2. Add two new sequence library (cfg_dcr_seq_lib & cfg_plb_seq_lib) which extends from uvm_sequence_library. All the core initilization sequences have been added to above sequence library.
3. Instead of starting the cfg_dcr_seq, cfg_dcr_seq_lib was started.

UVM-EA	UVM-1.X
Base Test:	**Base Test:**
task run();	task
reset_seq.start	reset_phase(uvm_phase
(sequencer_db)	phase);
	phase.raise_objection
cfg_system_dcr_seq.start	(this,“reset_phase“);
(sequencer_db);	
	reset_seq_lib.start(s
cfg_system_plb_seq.start	equencer_db)
(sequencer_db)	phase.drop_objection(
	this,“reset_phase“);
	endtask
endtask	
	task
	config_phase(uvm_phas
	e phase);
	phase.raise_objection
	(this,“config_phase“)
	;
	cfg_system_dcr_seq_li
	b.start(sequencer_db)
	phase.drop_objection(
	this,“config_phase“);
	endtask

UVM-EA	UVM-1.X
TestCase:	**TestCase:**
task run();	task
super.run();	main_phase(uvm_phase
begin	phase)
	phase.raise_objection(
run_seq.start(sequencer_db	this,“main_phase“);
);	
end	run_seq.start(sequence
endtask	r_db);
	phase.drop_objectectio
	n(this,“main_phase“);
	endtask

d) Drivers:

Apart from automated changes the initial reset values being driven during the reset phase was pushed into reset_phase of the driver. One assumption is that there is no need to jump phases and the other assumption is that there would not be any transaction queued until reset is de-asserted. Example:

UVM-EA	UVM-1.X
Run Task:	**Changes:**
task run();	task reset_phase(uvm_phase
begin	phase);
drv_reset_signal();	
wait_for_reset();	phase.raise_objection(this,
forever begin	„“reset_phase“);
get_and_drive();	drv_reset_signal();
end	wait_for_reset();
end	phase.drop_objection(this,
	„“reset_phase“);
	endtask
	task run_phase(uvm_phase
	phase);
	forever begin
	get_and_drive();
	end
	endtask

3) UVM Objections:

In UVM EA the test was terminated using global_stop_request. In UVM 1.x since there is more than one phase which can consume time, the new mechanism of phasing control and test termination is added.
Objections should be raised and dropped from every component in its phases so that they are not terminated prematurely. Typical usage is to raise an objection before starting the sequence and drop it after the sequence is completed. The raising and dropping of objections is hierarchical in nature and it traverses up to top with help of counters.

UVM-EA	UVM-1.X
task run();	task
	run_phase(uvm_phase
my_seq.start(sequencer_d	phase);
b);	phase.raise_objection(t
global_stop_request();	his);
	my_seq.start(sequencer_
endtask	db);
	phase.drop_objection(th
	is);
	endtask

In some components like scoreboards which don’t want to raise and drop objections for very transaction there may be a need to delay the phase termination after all the objections to that phase are dropped then those components should raise an objection in phase_ready_to_end method

Caution:
It is necessary to raise and drop objection. Avoiding to do so would result in any thread started in the phase being killed and there is no error/warning message printed.

V. Recommendation:

1. For users moving from OVM2.1.X to UVM: It is recommended that they convert OVM to UVM-EA first with scripts available in public domain and then move to UVM1.X
2. For consistency, avoid wild cards in the scope argument of uvm_config_db.
3. One must raise objection in each time-consuming phase a component uses. Otherwise, its threads would be killed without any error message when all other components have dropped their objections for this phase.
4. Though most of the features are backward compatible, run simulation with +define+UVM_NO_DEPRECATED

to identify all the deprecated features and any compatibility issues.

5. It is recommended to print the UVM version in the log file to avoid any confusion, using builtin macro `UVM_VERSION_STRING
6. For performance reason, do not raise and drop objection for every transaction received in monitor, driver or scoreboard.
7. UVM provides an automated way of setting the variable from command line. This is useful but needs to be used with care. Please see VI.4 below.
8. To set parameters of a component from the command line, use uvm_config_db. It's better to avoid embedding field automation macros which would add huge amount of extra code to copy, clone, print etc.
9. Always pass unique string while raising and dropping objections. This will aid in debug.

VI. Enhancement Requests to Accellera

1. User Guide: Need an example showing what care needs to be taken to achieve block to sub-system to full-chip reuse.
2. User Guide: Need guidelines on collection and reuse of functional coverage UVM environments.
3. Common base library: In tests, set_config are used to apply a variety of configurations. In case one or more are not applied due to some errors, no UVM errors are reported. Unsuccessful set_configs done by tests should report UVM errors.
4. Common base library: UVM needs to flash an error message if command line argument is set, but is not used or the said hierarchy does not exist.
5. Sequence library should be enhanced to support parallel thread execution.

VII. ConClusion

The effort required to develop the automation script was minimal and helped us convert more than 90% of the code to UVM1.1 with ease. The move was justified by the many new features of UVM1.1 which were beneficial in improving the quality of our testbench. During the process of migration we filed a few incremental enhancement requests on UVM.

In our experience, the eco-space around UVM has taken off very well based on the availability of VIPs and EDA tool support. With this, plus the simulator independence which came with UVM-EA, and the new features of UVM1.x, we believe we have a long-term stable baseline to develop our testbenches and utilities going forward.

VIII. Acknowledgements:

The authors are indebted to the continuous guidance and feedback they received from Ashish Kumar of LSI Technologies. The authors also thank Amit Sharma from Synopsys for his valuable inputs.

IX. References:

[1] "IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2009, 2009
[2] OVM User Manual, ovmworld.org.
[3] OVM 2.1.1 Reference, ovmworld.org
[4] Accellera Verfication IP Technical SubCommittee (UVM Development Website); http://www.accellera.org/
[5] UVM Class Reference, http://www.accellera.org/activities/vip
[6] Verification Intellectual Property (VIP) Recommended Practices (http://www.accellera.org/activities/vip/VIP_1.0.pdf).
[7] Universal Verification Methodology (UVM) draft 1.0 User's Guide
[8] On-line resources from http://www.uvmworld.org
[9] On-line resources from http://www.verificationacademy.com/