Melting Verification Pot: Integrating RVM/VMM and UVM, a Practical Guide and Lessons Learned.

Mark A. Azadpour azadpour@yahoo.com

***Abstract*— In this paper, we share our experience with migration of a RVM testbench to SystemVerilog with a mixture of UVM and VMM methodologies. Various practical approaches and novel usage of UVM capabilities are presented as well as initialization and phasing in such a mixed environment. The purpose of this paper is to share our experience integrating various VIP's that use different languages and methodologies in a unified environment. This paper shares our experiences with a complex System-on-Chip (SoC) ASIC verification process which integrates a number of externally and internally generated VIP's using UVM methodology with VCS. We detail our conversion experience of an SoC level test bench and environment and provide guidelines on what needs to be considered and how the migration should be planned to achieve a successful VMM to UVM conversion of top level environment as well as all the tests. Our goal was to achieve the conversion at the least amount of time to minimize schedule impact. These could be used by the user community to plan the migration to UVM.**

Keywords-Verification, UVM, VMM,RVM, SoC Verification.

I. INTRODUCTION

This paper details our experience of converting a Vera/RVM based testbench into a SystemVerilog/UVM based top level testbench with VMM sub-environments as well as Vera based sub components that were inherited from our legacy testbench. The conversion effort utilizes the Synopsys interoperability library for UVM1.0 release and we used VCS2011.03-SP1-2 as the simulation engine.

A novel concept used from UVM 1.0, namely the Universal database (UDB), was utilized to register components and provide a mechanism to provide a new facility to synchronize various components in the three methodologies of RVM, VMM and UVM. The goal of this project was to demonstrate the ability to convert testbenches in layers from Vera to SystemVerilog in general and utilize UVM in particular. The ultimate goal is to eventually convert all IP VIP's as well as SoC level VIP to the UVM methodology.

In this project, we proved that the full conversion can be accomplished in layers and not the whole testbench needs to be converted to UVM in one step. This provides us with a migration path which gives our organization the flexibility to perform the conversion as resource and schedule permits without a need to jeopardize either constraint for the sake of accomplishing the conversion,

II. THE TESTBENCH

A. General Structure

The verification environment was that of a Constraint Randomized Testbench (CRT) that is used to verify controller chips for a storage application. The ASIC was used in an embedded environment and required substantial amount of firmware in terms of programming CSR to allow the inter-operation of various blocks as well as . This CSR programming and interrupt handling was one of the reasons we needed to re-use portions of testbench from the IP testbenches, Verification IP (VIP's), and required further constraint definition at the SoC level to allow various components to work with each other. In general, the VIP's are meant to stress the IP and cover a lot more functionality than is feasible both in terms of time and resources. Therefore, the goal of VIP reuse is to use the IP testbench with minimal structural modification and use the constraint setting to custom tailor the behavior of the VIP at the SoC level.

The Device Under Test (DUT) consists of a number of internal and external Intellectual Property (IP) that was developed in both VHDL and Verilog. The supporting VIP's were developed in both Vera and SystemVerilog languages as well. The challenge of creating SoC testbench was further complicated by the fact that various methodologies were utilized to develop VIP for various IP testbenches including RVM, VMM1.0, VMM1.1, VMM1.2 and UVM. Since our goal was to reuse as much as of the VIP's as possible, we had to accommodate all the above methodologies in our testbench as the IP deliveries were on a tight schedule and we could not afford to re-engineer any of the VIP's into a new methodology.

B. Various methodologies (UVM vs. VMM)

The fundamental issue we had to deal with when migrating from VMM top level to UVM was the object hierarchy in UVM versus VMM. In this discussion, VMM included all various versions as well as RVM as we had already developed testbenches with VMM as the top level. The environment object is the top level object in VMM where as test entities are the top level objects in UVM. Furthermore, in the RVM and VMM code that was supplied by various IP teams, the function "new" was used to instantiate children objects and the formal arguments to the new function was used to pass higher level object handles such as environment objects and firmware pointers to the various objects as opposed to breaking the instantiation and connection of various objects in various phases as advocated by the UVM methodology.

Other parts of the testbench utilized VMM1.2 which uses implicit phasing and the construction and connection of sub-

components are delegated to various phases. These portions of the VMM1.2 testbench were used verbatim as the interoperability library automatically took care of interaction between VMM and UVM as detailed in the following sections.

C. *Internal vs. External developed Intellectual Property.*

In our testbench, we integrated a number of internally developed VIP's that are fairly coherent and follow the same methodology for the most part as well as externally developed testbenches and models that we have less control over and have to make wrappers around various parts of the testbench to integrate them in the SoC testbench.

III. MIGRATION PATH TO UVM

Before starting the migration process, we needed to decide what methodology should be used as the top level. This is the fundamental step needed in order to be able to utilize interoperability library which allows VMM and UVM to interact seamlessly for the most part as detailed next.

A. *Synopsys Interoperability Library*

The Synopsys interoperability library provides two scenarios which the user must decide upfront before starting to integrate a mixed environment of UVM and VMM verification components.

In one scenario, the top level of the test bench is written in UVM, UVM 1.1 in our case, with subordinate components implemented in VMM methodology.

In the other scenario, the top level of the testbench is implemented in VMM and the sub-groups are in UVM. This designation is communicated to the interoperability library via compile time switches and it determines the general flow of the overall SoC testbench.

We wanted to evaluate the capabilities of interoperability library as well as how much work was involved in terms of glue code for each of the above scenarios. First, we set out to test a top level VMM1.2 testbench with UVM1.1 sub-envrironment to demonstrate the interoperability. We were able to verify functionality in that configuration and compare to the other scenario, it seemed for the test case, it required less coding on the part of the user of the interoperability library for our configuration.

Since our long term goal is to migrate to UVM exclusively, we decided to choose a UVM top level to set the stage for future migration of new components to UVM.

There are a number of benefits provided by the interoperability library. One major facility is the seamless messaging interoperability between VMM and UVM. For our case, with UVM top level, all the `vmm_notes macros were automatically taken care of to behave as if they were `uvm_info and the messaging hierarchy was managed by the library and it was seamless to us. This feature saved us a lot of manual labor or scripting effort to manage this on our own.

The other area of interest to us was the implicit phasing and once we created a general mapping between the two methodologies as depicted in figures 2a, and 2b, the phasing methods were called in the appropriate methodology seamlessly. That saved us a lot of work during the creation of the SoC testbench.

The following packages and include files were added at the top-level module:

```
import uvm_pkg::*;
import vmm_std_lib::*;
import uvi_interop_pkg::*;
  `include "vmm_ral.sv"
```

The above packages are followed by the OpenVera packages. The order is important as the interoperability package modifies some of the macros in the open Vera package

```
import OpenVera::*;

import OpenVera::register ;
```

In the NTB mode, all the Vera code gets put in a package of SystemVerilog. That gets imported by import Open Vera::*.

In our testbench, we had both the VMM Register Access Language (RAL) and the UVM implementation residing in parallel in one environment. In order to accomplish this, we divided the memory map so that the new blocks were done in UVM register space and the legacy code used the VMM RAL from the previous projects. Since they are both object oriented implementations, we were able to call the VMM RAL from UVM sequences with no problems.

The following compilation options were included in our VCS compile options to pull in appropriate libraries and to make the interoperability work properly. In addition, the NTB options were required for the Vera and SystemVerilog mixed environment.

```
${lib_loc}/uvm_dpi.cc
CFLAGS  -DVCS
+sverilog
+cpp g++
+debug_all
+cc gcc
+acc
+sverilog
+debug_all
+acc
+vpi
+define+VMM_UVM_INTEROP
+define+VMM_ON_TOP
+incdir+/home/mazadpou/interop/packages/uvm_vmm_intero
p/src
+define+END_NTB_OPTS__VCS
+define+START_TB__SV
+incdir+$RPATH.$L
```

Figure1- List of switches needed to compile with the interoperability and UVM library.

With the last +incdir+ providing a mechanism to include files that are customized per compilation to provide flexibility. This allows us to customize each build for various targets and testbenches.

Compilation of UVM/VMM mixture is hard to debug as the error messages were not as descriptive as we would like them to be. Therefore, it is wise to create two separate standalone environment and do most of the work separately and bring the files in two methodologies together at the end to save a lot of debugging headaches as teams become savvy with the compiler error messages and what they really mean

Following are a few of examples of error messages with the cause of the error:

```
Error-[SV-ICA] Illegal class assignment
  Expression 'obj' on rhs is not a class or a compatible class and hence cannot be assigned to a class handle on lhs.
 Error-[SV-ICA] Illegal class assignment.
```

The above error message is due to mixing uvm_test and vmm_test instances in the same testbench. with UVM_TOP defined as a compilation switch, the interoperability expects uvm_test objects in the top level and not the vmm_test type objects.

Here is another example of failure due to the library compilation that may not be very obvious at the first glance:

```
undefined reference to `uvm_hdl_read'
collect2: ld returned 1 exit status
make: *** [product_timestamp] Error 1
Make exited with status 2
```

This is due to dpi.cc not being compiled in the compile options.

Next we will look at some of the RVM and VMM inclusion points in our mixed testbench.

B. RVM and VMM

NTB mode of VCS allows RVM objects to be called from VMM objects by importing NTB libraries. In this mode of operation, a VMM class can be extended from a RVM class and overloading of member functions is allowed. In this manner, we were able to instantiate RVM objects in VMM or UVM objects and extend functionality.

However, once the transition is made to RVM components, the RVM objects may not call VMM or UVM based objects. As a result, care must be taken that no transaction goes back and forth and all the intermediate objects are implemented in SystemVerilog (UVM or VMM). A sample code is shown in Figure 3 detailing how an RVM method is called from a VMM class.

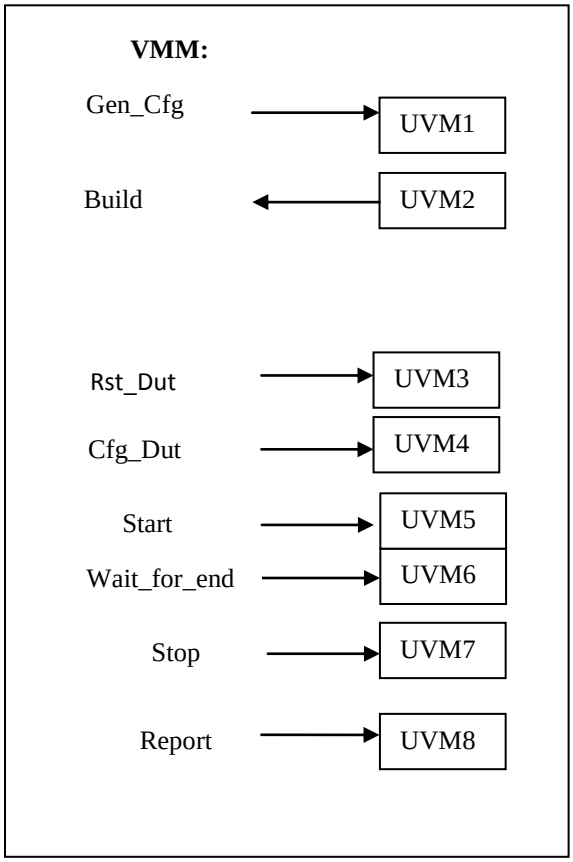

Figure 2-a Phasing between VMM and UVM.

The VMM phases refer to the UVM phases the passing of execution order is designated with a UVM number in a box. The receiving end of those boxes are shown in Fig 1-b

The actual UVM code that is part of the test that was run is shown in figure 4 where the function call is implemented to the UVM sequence _item. In this case, a write takes place.

Figure 5 is the UVM intermediary function that is called. This function in turn calls the VMM function. It should be noted that this function is extended from the OpenVera implementation of the micro class and the transfer occurs at this level to the Vera implementation which is depicted in Figure 6.

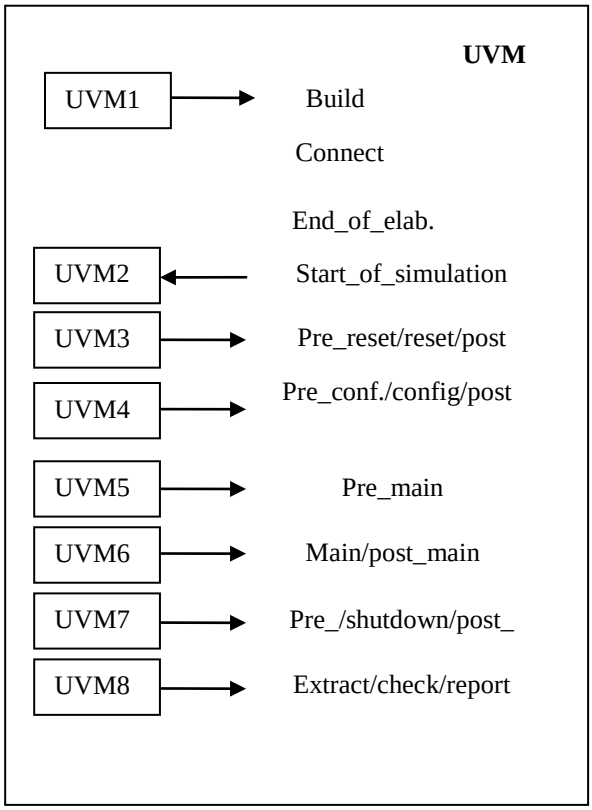

Figure 2-b Phasing between VMM and UVM.

The UVM phases refer to the phases that the execution order is passed to. The UVM boxes with numbers in a box are the entry points to the phase with arrow leaving the box and the opposite is when the execution order is passed back to the VMM side.. The receiving end of those boxes are shown in Figure 2a.

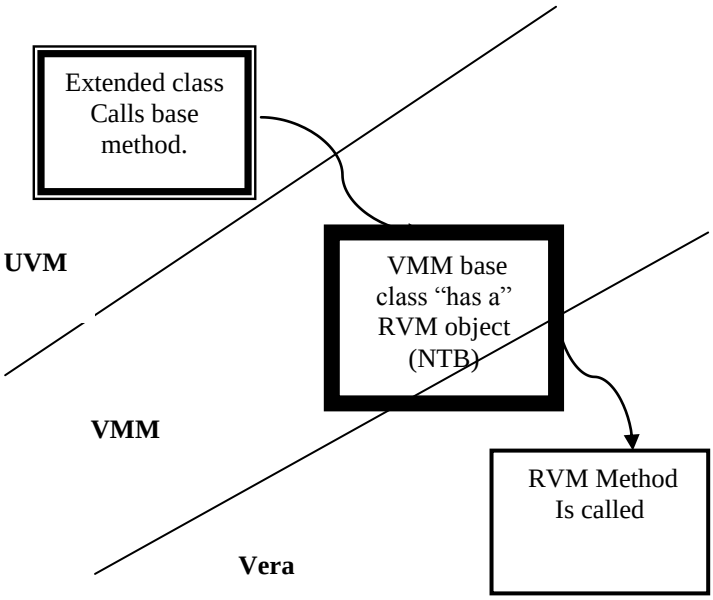

Figure 3- Flow of RVM method call from a VMM object

```
`ifndef FOO_TEST
`define FOO_TEST

import uvm_pkg::*;

import OpenVera::*;
`include "sregs.vri"
`include "regs.vri"

typedef class env;

class foo_test extends base_class;
  `uvm_component_utils(test_class_name);

function new(string name = "foo_test",
uvm_component parent = null);
      super.new(name, parent);
      `uvm_info(get_name(), $psprintf("Newing class
%s.\n",name), UVM_NONE);
  endfunction

 env mEnv;

virtual function void build_phase(uvm_phase phase);

   super.build_phase(phase);
   this.mEnv = super.mEnv;
   tb_env=mEnv;

 endfunction // build_ph

virtual function void
start_of_simulation_phase(uvm_phase phase);
   super.start_of_simulation_phase(phase);

……

Endfunction

virtual task main_phase(uvm_phase phase);
   register foo_reg;
   integer status;
   super.main_phase(phase);

  phase.raise_objection(this);

  mEnv.micro.write(rst_reg, 32'h1212,, 1);
endfunction
```

Figure4 – The UVM code that is calling UVM sequence item the in the main phase.

```
import AXI_MODEL::*;

class Ccmd extends uvm_sequence_item;
   process process_id;
   integer transID;
   SCmd cmd=new;
endclass

class Micro extends OpenVera::micro;
  SAxi_cmd responses[$];

extern virtual task automatic write( register _reg,
                        logic [31:0]  value,
                        integer enable=-1,
                        integer stall=0,
                        axi_cmd::priority_t
_priority=axi_cmd::priority_high,
                        bit status=1);

  extern virtual task automatic read_task( register
_reg,
                          axi_cmd::priority_t
_priority=axi_cmd::priority_high,
                          bit status=1,
                          ref integer read_data);
…

endclass
```

Figure 5- Extension from the OpenVera to a UVM

The transactor in Figure 6, calls the appropriate transactors that are actually run the BFM to implement the detailed transaction and return the results back.

```
#include <rvm_std_lib.vrh>
#include "features.h"

class micro extends rvm_xactor {

  axi_cmd_channel out_chan;
  axi_cmd_channel in_chan;

  virtual task write(register _reg, bit [31:0]  value,
integer enable=-1, integer stall=0,
axi_cmd::priority_t priority=axi_cmd::priority_high,
bit status=1) {
….
}

  virtual task read_task(register _reg,
                    axi_cmd::priority_t
priority=axi_cmd::priority_high,
                    bit status=1,
                    var integer read_data)
  {

     axi_cmd cmd;
…
}

virtual  task main_t() {
    axi_cmd cmd;
    super.main_t();
    fork
      monitor_results();
    join none

    while (1) {
      wait_if_stopped_or_empty_t(in_chan);
      cmd=in_chan.peek_t();
      if (cmd.stream_id!=this.stream_id) {
        cmd=in_chan.get_t();
        rvm_debug(log,psprintf("found
id(stream=%3d)!=ourid(stream=%3d), removed...",
                        cmd.stream_id,this.stream_id));
      } else {
        void=in_chan.notify.wait_for_t(in_chan.GOT);
      }
    }
  }



}
```

Figure6 – The underlying VMM implementation that is called through the wrapper UVM extended class.

C. *Using UDB as a method to keep track of instantiations in VMM.*

Due to the mixture of languages and methodologies used in our bench, we needed a way to keep track of various components to be constructed and to ensure that a component is constructed prior to being used. In the VMM code, that was originally ported from RVM, the construction of sub-components as well as connection among those components were done in the "new()" method; therefore, we had to ensure that the consumer of those component had a handle that was pointing to the actual object prior to consumption.

In order to remove the order dependency among VMM and UVM code, we utilized a registry mechanism that allowed us to produce objects and register them to signal the existence of the component to all consumers. And then at the consumer end, we checked for the existence of the object before consuming the information.

Universal Database (UDB) is a static Singleton which is a perfect candidate for this operation. The static nature of this object is perfect and made it available from the start of simulation to both UVM and VMM as well as RVM objects by the nature of inheritance that is available in the NTB.

Therefore, the consumer method would check with the registry to ensure that an object of interest is registered with UDB. If it is not, it would instantiate it and then goes through setting constraints and finally doing randomization to ensure proper constraints were applied.

As an example in the code snippet shown in Figure 7, in the *start simulation* phase of the environment, if the configuration object of a block called clkreset is not instantiated, then the simulation is halted. This ensures that we never encounter NULL pointers and that someone else would take care of instantiation as well as registration of this object with the config_db. This decouples the dependencies between VMM and UVM and in fact, we create this object in the VMM side and then register it with the config_db which is a UVM entity.

A side-effect to watch out for that was experienced was that in one of the phase implementations, we had forgotten to call the super function. In that phase, since super.xxxx is not called, the reference to the config_db is not set and one must call uvm_config_db.get() to set the reference manually. This is a pitfall in the set and get routines to retrieve things from the UDB database that might be hard to pin-point as the error messages are not descriptive.

```
function void
env::start_of_simulation_phase(uvm_phase phase);
  super.start_of_simulation_phase(phase);
  `uvm_info("TRACE", $sformatf("%m"),
UVM_HIGH);

   uvm_report_cb::add(null,mUEC);

   // bind the configurations
   if ( !(uvm_config_db
#(clkrst_cfg)::get(uvm_top,"","CLKRST_CFG",firm.m
ClkRstFw.mCfg)))
     `uvm_fatal("DUT_ENV","failed to get the
ClkRst_cfg handle\n");

uvm_config_db
#(uvm_object_wrapper)::set(mCtrlAxiMaster,
                           "sequencer.main_phase",
                           "default_sequence",
AXI_MODEL::AxiCmdSequence::get_type());

  uvm_config_db
#(uvm_object_wrapper)::set(mSrvoAxiMaster,
                           "sequencer.main_phase",
                           "default_sequence",
AXI_MODEL::AxiCmdSequence::get_type());
……
endfunction
```

Figure7- A code snippet showing how the config_db can be used as a mechanism to ensire objects are created and registered prior to using them.

UVM Components

There were a number of UVM components and environment that were instantiated in the testbench. The major components included a set of base-testcase classes that all the testcases were inherited from. This allowed for information hiding and extending those base classes that housed the common code among a set of tests is supported of coping common code.

The top environment classes as well as top firmware classes were implemented a UVM component to ease interaction as well as one of the register programming blocks.

TLM1.0 facilities provided by UVM were used in the testbench. The TLM export and import ports as well as TLM FIFO's were used in such manner that allowed for another method of data production and consumption between VMM1.2 and UVM1.1 entities and provided for seamless integration. A code snippet in Figure 9 represents the uvm_tlm_fifo and the put and get port s that were used to throttle the information being sent to the virtual sequencer.

```
class clkrst_agent extends uvm_agent;
  `uvm_component_utils(clkrst_agent)

  typedef uvm_sequencer#()
clkrst_firmware_sequencer;

   clkrst_firmware_sequencer clkrst_seqr;


    virtual function void build_phase(uvm_phase
phase);
     super.build_phase(phase);

      clkrst_seqr =
clkrst_firmware_sequencer::type_id::create("clkrst_s
eqr", this);

     // set the execution phase of the sequence to
config_phase since that is when firmware is invoked.
     uvm_config_db #(clkrst_cfg)::set(uvm_top,
"clkrst_seqr.config_phase", "default_sequence",
clkrst_firmware::get_type());
   endfunction
endclass
```

Figure 8- Code snippet showing the creation and registration of the entity with the config_db.

```
Class cThrottle extends OpenVera::micro;
  S_cmd responses[$];
static   integer transID=0;
  AxiMasterAgent mAxiAgent;
   uvm_tlm_fifo #(SAxi_cmd) micro_cmd_queue;
   uvm_put_port #(SAxi_cmd,SAxi_cmd) p;
   uvm_get_port #(SAxi_cmd,SAxi_cmd) g;
   logic process_cmd_queue_flag =0;

   event read_complete;

  function new(string instance="Micro", integer stream_id=-
1, AxiMasterAgent axi_agent);
    super.new(instance, stream_id);
    mAxiAgent = axi_agent;

micro_cmd_queue=new($sformatf("%s.micro_cmd_queue",
instance),uvm_top,0);
    p = new($sformatf("%s.p",instance), null);
    g = new($sformatf("%s.g",instance), null);
    p.connect(micro_cmd_queue.put_export);
    g.connect(micro_cmd_queue.get_export);

  endfunction

  task automatic ::check( register _reg,
                integer value,
                integer enable=-1,
                integer stall=0,
                axi_cmd::priority_t
_priority=axi_cmd::priority_high);

   SAxi_cmd Scmd;
   int num_of_entry;
   AxiCmdSequence seq;

continued on the next  page …
```

Figure9- Code snippet showing the extension of an OpenVera class in a UVM sequence-item.

```
Contin….

  forever begin
    this.g.get(Scmd);
    num_of_entry = micro_cmd_queue.used();
    `uvm_info("TRACE", $sformatf("%m : fifo depth %d
\n",num_of_entry),UVM_NONE);
    seq = new;
    if ((Scmd.process_id.status !== process::KILLED)||
(Scmd.process_id.status !== process::FINISHED)) begin
      seq.axi_cmd = Scmd.cmd;
//          $display(" process start\n");
      seq.start(mAxiAgent.mSeqr);
//          $display("Process ended\n");

          if (Scmd.cmd.mKind == AxiCmd::READ)
begin
            responses.push_back(Scmd);
            -> read_complete;
             #1;
          end
    end else begin
             `uvm_info("TRACE", $sformatf("%m ,a
killed process's command was removed from queue "),
UVM_NONE);
     end
  end
endtask

task Trottle::pull_data;
 int        process_index[$], process_id;
   Scmd.cmd = cmd;
   Scmd.process_id = process::self();
   p.put(Scmd);
   `uvm_info("TRACE", $sformatf("%m : fifo depth %d
\n",micro_cmd_queue.used()),UVM_NONE);
   do begin
    @read_complete;
    process_index = responses.find_index(x) with (
x.transID == Scmd.transID );  //responses.first_index()
with i: (responses[i].process_id == Scmd.process_id);

    if (process_index.size() > 0) begin
     process_id = process_index.pop_front();

      if ( process_id >= 0) begin
       read_data = responses[process_id].cmd.mData[0]
>> (responses[process_id].cmd.mAddr[1:0] * 8);
            responses.delete(process_id);
      end
    end
   end
   while ( responses.size() > 0);

 end else begin
   super.read_task(_reg, _priority, status, read_data);
 end
endtask
```

I. CONCLUSION

This paper demonstrated that migration to UVM can be accomplished in stages to ease the transition pain. In addition, the legacy code could be used while the new code is developed or ported to the new methodology in a multi-tier IP development and SoC organization. This migration allows for organizations to port various portions of the testbench as time and resources are available rather than waiting for a one-shot conversion that inherently will incur downtime as well as recovery time and the learning curve involved for testbench users after the conversion. In addition, this layered approach allows organizations to introduce a new methodology gradually and independent from the IP deliveries to allow a smooth transition. In our case, we had legacy Vera code and we used Synopsys's NTB and interoperability library to allow the transition from the top level with possibility of moving various sub-environments at a later date.

In this paper we detailed the migration of a RVM/VMM to a top level UVM compliant testbench with lower levels to be converted at a later time. Various practical and novel usage of UVM capabilities were presented to coordinate the initialization and phasing in various methodologies. Integrating various VIP's that use different languages and methodologies in a unified environment was the goal of this project and we were successful in achieving this goal.. We shared our conversion experience of an SoC level test bench and environment with the goal of providing guidelines on what needs to be considered and how the migration should be planned to achieve a successful VMM to UVM conversion of top level environment as well as all the tests. Our conversion goal was to achieve the fastest conversion time with maximum depth of conversion with minimum schedule impact. Hopefully, our experience could be used by the user community to plan their migration to UVM.

REFERENCES

[1] Chris Spear, SystemVerilog for Verification,, 2nd edition Springer

[2] Training Department at Synopsys, SystemVerilog UVM 1.0 Workshop, Student Guide, 2011.03.

[3] Wlillamette HDL staff, Introduction to the Unviersal Verification Methodlogy, part number 071988

[4] Doulos, UVM Golden Reference Guide, Doulos, May 2011.

[5] VCS 2011.03 reference manual, Synopsys Corporation..

[6] UVM/OVM Methodology Cookbook. Mentor Graphics uvm.mentor.com or http://verificationacademy.com/uvm-ovm.

[7] UVM refernce guide at http://www.accellera.org