Migrating to UVM : Conquering Legacy

Santosh Sarma
Wipro Technologies
Pune, India
santosh.sarma@wipro.com

Amit Sharma
Synopsys India Pvt Ltd.
Bangalore, India
amits@synopsys.com

Adiel Khan,
Synopsys, UK
Cambridge
adiel@synopsys.com

***Abstract*— Over the last few years, there have been rapid advances in verification techniques and methodology. This has enabled the creation of highly reusable, configurable and powerful verification environments which has helped to increase the overall verification productivity significantly. Though multiple verification methodologies have brought about this evolution, UVM has brought with it the much desired standardization. Before this standardization, a significant number of verification engineers still used traditional directed approaches based on HDLs (Hardware Description Languages) for their verification test benches. Also, there have been multiple projects where other verification methodologies like VMM and OVM were used. Migration of a complex verification environment and its components into a UVM-complaint test bench requires careful planning. There can be several factors that influence the migration path: Existing test bench structure and complexity, time and resources available, and the availability of the UVM-based Verification IP. In quite a few scenarios, a complete migration to UVM might not be possible. Thus, we would need to look at scenarios where legacy components would have to co-exist with UVM based components. Based on the state of different projects and their immediate priorities, we focus on the following aspects in this paper.**

- **Reuse of Legacy BFMs in a UVM based environment**
- **Gradual integration of UVM components in a Legacy environment.**
- **Extending existing applications to seamlessly work with UVM base classes**

Keywords— UVM, OVM, VMM,BFMs, System Verilog, Abstract Classes, Verification IP, Policy Classes, Parameterization

I. INTRODUCTION

Advanced verification methodologies are now being deployed by verification engineers to create highly efficient transaction-level, constrained-random verification environments using System Verilog. These methodologies provide the framework for developing re-usable verification components, sub-environments and environments. There are multiple compelling solutions available in this space with Universal Verification Methodology (UVM) being the widely accepted standard. Verification IP vendors and project groups who have already made an investment in creating a Verification IP or a verification infrastructure around their respective design blocks and clusters would necessarily want to continue to leverage the earlier investment as the projects using these move to newer methodologies. Engineers would have also developed advanced applications like performance analyzers, scoreboards, register abstraction mechanisms for their existing verification infrastructure. Hence there will be multiple cases where the legacy components would have to co-exist with the newer UVM based components. This paper focuses on migration related issues for environments based on different methdologies and low level BFMs.

Developing all such components from scratch in UVM can be a time consuming exercise. Additionally, any such components developed afresh would have to go through their own maturity cycle after which they can be considered reliable or fully functional. Hence it becomes increasingly important to plan for reuse of verification components across different methodologies. This will ensure that the effort spent and maturity undergone by the component can be leverage in new projects. It is also essential to hide all the implementation aspects of the methodology from the end user so that the verification components remain simple to use and deploy. Thus, the migration of a complex verification environment and its components into a UVM compliant testbench requires careful planning. In quite a few scenarios, a complete migration to UVM might not be possible at one go and there might also be a requirement to maintain and update the legacy components to support revisions of older projects.

Hence, the ideal scenario for the developers would be to provide such verification components and applications in newer flavors or methodologies by just spending a fraction of the time they would have otherwise taken to develop them from scratch. Also, any new development should be transparently propagated to all different flavors of the verification components and applications. Given that each methodology is different in its own way, verification engineers are seldom expected to master all of them. Hence, end user of such components should not have the need to develop knowledge of multiple methodologies for efficient adoption of the same.

This paper would bring out techniques as to how the above requirements can be met using some of the techniques available through different facets of Object Oriented Programming as well as some useful features available in UVM. This paper takes the example of some VMM applications and legacy Verilog Bus Functional Models (BFMs) and enumerates the steps that are required to be able to use them seamlessly in newer SystemVerilog verification methodologies like UVM. Using the approaches described in

this paper, the legacy infrastructure and newer methodologies can be made truly interoperable.

II. REUSE OF LEGACY BFMS IN UVM

In a SystemVerilog based verification environment, components get constructed dynamically during the simulation and interact with the Design Under Test (DUT) using virtual interface handles. However this approach does not work very well when using legacy BFMs and their task based interfaces for the signal level interaction with the DUT. This section presents an approach using *'Abstract Classes'* through which legacy BFMs written in Verilog are hooked up to higher level class based components to create a standard structure compliant to the requirements of a specific methodology and base class library. The implementation makes use of *abstract classes* to define methods that invoke the Application Programming Interfaces (APIs) of the underlying BFM.

A. Abstract BFM Classes

An '*abstract class'* is a special kind of class that cannot be instantiated directly but has to be inherited or concretized before being used. The purpose of an abstract class is to provide a generic base class definition allowing the developers to fill the implementation in the derived classes and add new flavors to it. Applying this concept to legacy verilog BFMs, an abstract class definition is created which declares prototypes for all API calls available within the verilog BFM. The prototypes are declared as "*pure virtual*" so that they can be overridden at run time with the actual BFM function calls which are tied to it in an adapter class. The adapter class is the one that derives from or concretizes this abstract class. The concrete class is then bound to the verilog instance of the BFM using the SystemVerilog "bind" concept. The concrete class handle is then used by the verification infrastructure to interact with the underlying verilog BFM. Using this approach the existing verification components can be made truly reusable by using run time binding of the verilog BFM instance to the wrapper class in any new methodology like UVM. This enables the developer to avoid using hard-coded macro names or procedural calls within the high level components in the test environment. The abstract class can also be parameterized to match similar kind of parameterization available in the Verilog BFM.

For example, the abstract class for a legacy low level driver BFM can be defined as shown below.

```
virtual class drv_bfm_api_c
#(parameter ADDR_W = 32,
parameter DATA_W = 8) extends uvm_objec t;
  int command_issued_counter;
  event signal_command_issued;
  event signal_response_complete;
  pure virtual function bfm_init();
  pure virtual function
     bit all_transactions_complete();
  pure virtual function
     int get_command_issued_queue_ size();
  pure virtual function
     int get_response_queue_size(); …..
```

Figure 1. Abstract class for a legacy driver BFM

B. Concerete BFM Adapter Classes and SV "Bind"

The abstract class for the BFM cannot be used as it is. It has to be concretized in another extended class before it can be used by the test environment components developed using UVM base classes. This is done by implementing an adapter module for the BFM. Inside the adapter module, a customized child class is defined for concretizing all the required methods from the BFM abstract class. The concrete definitions for the abstract class methods actually invoke the corresponding APIs inside the Verilog BFM. The APIs are invoked just by using the BFM module name instead of its hierarchical instance name. This allows the flexibility of binding the functions to any BFM instance at run time without having the need to use the instance name of the BFM in the function call.

The use of only the module name instead of the actual instance name is facilitated by the use of the SytemVerilog "*Bind*" concept in the top testbench module. The *bind* keyword allows auxiliary verification code/program to be bound to a target module/instance. The bind creates an instance of the auxiliary program/module inside the target scope thereby acting as an instantiation mechanism but in a non-intrusive manner. In this case the adapter module created as described above is bound to the target verilog BFM instance in the testbench scope encapusalting the compelete verification infrastructure. This automatically creates an instance of the adapter under each verilog BFM in the testbench. Thereby all procedural code inside the bound module can directly use the BFM module name for resolving the hierarchical references. The same concept can also be extended to events and variables used inside the Verilog BFM that can be hierarchically connected to events inside the UVM agent through the adapter module.

The adapter module thus provides a mechanism to acess the low level APIs of the BFM with the ability to have runtime binding. The next challenge is to somehow tie each UVM agent/component in the test environment to its corresponding target verilog BFM instance. This is easily achieved by using the string based lookup mechanism available in the form of the UVM Configuration/Resource database. Inside the Adapter module, an object of the concrete BFM class named *bfm_h* is created and a reference of this object is registered in

the UVM Configuration DB using *uvm_config_db::set*. The string name associated with this entry into the configuration space is the actual instance name of the BFM in the testbench hierarchy. The adapter module fetches the hierarchical instance name of the BFM using the "*%m*" format specification. This entry is then fetched by the UVM agent later to be connected to the corresponding erilog BFM instance. Thus the adapter module and the concrete class handle are made free of any fixed hierarchical references.

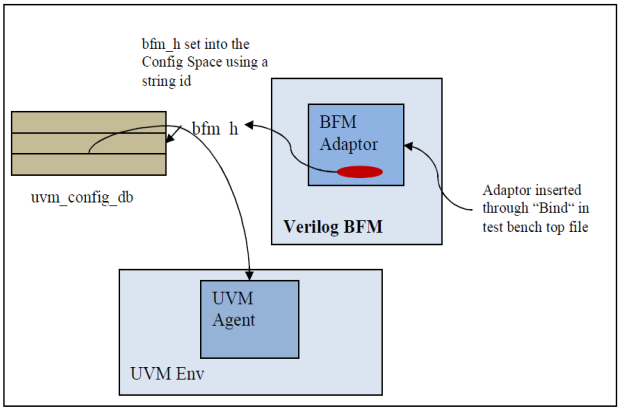

Figure 2. UVM agent connection to relevant Verilog BFM instance

```
module my_bfm_adaptor
     # (parameter ADDR_W = 32, parameter DATA_W = 32)
();
  //Concretize the Abstract Class here passing all
  //the parameters
  class drv_c extends drv_bfm_api_c #(ADDR_W,DATA_W);
    function new();
      super.new();
      //Tie the Abstract Class elements/methods to
      //the actual ones inside the BFM
      this.max_command_queue_size =
driver_bfm.max_command_queue_size;
                this.command_issued_counter =
driver_bfm.command_issued_counter;
                this.signal_command_issued =
driver_bfm.signal_command_issued;
                this.signal_response_complete =
driver_bfm.signal_response_complete;
    endfunction
    //Concretize the Abstract Class Methods here to
    //the actual BFM methods
    function init();
      driver_bfm.init();
    endfunction
    function bit all_transactions_complete();
      driver_bfm. all_transactions_complete();
    endfunction
    function int get_command_issued_queue_size();
      return driver_bfm.get_command_issued_queue_size();
    endfunction
    function int get_command_pending_queue_size();
      return
driver_bfm.get_command_pending_queue_size();
    endfunction
    function int get_read_response_queue_size();
      return driver_bfm.get_read_response_queue_size();
    endfunction
   endclass
   //Get the instance name assigned at run time
   string path = $sformat("%m");
   bfm_h = drv_c::type_id::create(path,null,path);
   //Set the Actual Hierarchical BFM name in the config
space
   uvm_config_db# (drv_c):: set ("","*",path,bfm_h);
endmodule
```

Figure 3. Concretizing Abstract Class methods

C. Binding the Verilog BFM in the top testbench scope

In the top testbench scope, the adapter module is bound to the Verilog BFM instance which is required to be used with the UVM agent. The bind here creates an instance of the adapter inside the verilog BFM which is then accessed by the top level UVM agent classes using the hierarchical name "*bfm_h*"

```
module tb_top();
….
  //Driver Verilog BFM instance named MSTR
  driver_bfm MSTR(…);

  bind driver_bfm:MSTR my_bfm_adapter
                       #(ADDR_W,DATA_W) mst();
```

Figure 4. Binding the Adapter to the BFM instance

D. Connecting the UVM Agent to the BFM Adapter

Once the hierarchical name of the BFM adapter is 'set' or registered in the configuration database, the UVM agent can fetch it during its construction and establish the required connectivity with the underlying verilog BFM. This can be done in the *build_phase()* of the UVM driver or agent wrapper as shown below.

```
class my_uvm_driver extends uvm_component;

 string BFM_NAME;
 //Local Adapter Concrete class handle
 drv_c bfm_h;

  virtual function void build_phase(uvm_phase
phase);
   super.build_phase(phase);
   //Get the handle to the Verilog BFM instance
   if (!get_config_string("BFM_NAME",BFM_NAME))
     uvm_report_fatal("NOBFMNAME","No BFM_NAME
string set");
   if (!get_config_object(BFM_NAME,obj,0))
    uvm_report_fatal("NOBFMHDL","No BFM Handle
set");

      assert($cast(bfm_h,obj));
  endfunction

  virtual task run_phase(uvm_phase phase);
    my_trans tr;
    super.run_phase(phase);
    forever begin
      seq_item_port.get_next_item(tr);
      //Access BFM clock
      @ (posegde bfm_h.clock);
      //Invoke BFM APIs
      bfm_h.init();
      bfm_h.set_command_addr(tr.addr);
      bfm_h.set_command_operation(tr.rw);
      bfm_h.set_command_data(tr.data);
      bfm_h.push_command();
      //Wait for BFM events
      @ (bfm_h.signal_response_complete);
      seq_item_port.item_done();
    end
  endtask
```

Figure 5. Binding the Adapter to the BFM instance

It can be seen above that the UVM driver can use the *bfm_h* handle just like a virtual interface handle.

In the top level UVM environment, the UVM agent is instantiated as follows:

```
class my_uvm_env extend uvm_env
  my_uvm_agent agent;

  function void build_phase(phase);
    //Bind Verilog BFM with instance name "MSTR" to the
    //UVM Driver component in the  Agent
    set_config_string("agent.driver","BFM_NAME","MSTR");
```

Thus the run time binding of the Verilog BFM to the UVM components makes the environment a lot more reusable and eliminates any need to have any hard-coded hierarchical references within the UVM VIP or environment component code.

E. Specific Migration Issues related to UVM Phasing

Run time phases were introduced from UVM-1.0 which make use of objections to co-ordinate between all the components and decide when the simulation should proceed to the next phase. Unless objections are raised in the run time phases, the *run_phase* would come to end immediately at Time 0. Hence it is recommended that each UVM component raise an objection in its run_phase and drop it when all operations are completed.

For components built using OVM, the best option to migrate to UVM Phasing is to replace the call to *global_stop_request()* with objection raise/drop calls in the *run_phase()* of the respective component or the environment. This can also be used in conjunction with sequences in UVM which have a handle of the *starting_phase* that can be used to raise and drop objections. However the *phase_ready_to_end* callback can also be very useful in properly managing the phase transitions in UVM. The use of the callback in the context of the Legacy BFMs being ported in UVM is shown below.

```
class my_uvm_driver extends uvm_component;
….
  bit ending = 1;

  virtual task delayed_end_of_phase(uvm_phase phase)
    wait (bfm_h.all_transactions_completed.triggered);
    ending = 0;
    phase.drop_objection("Finally Ending");
  endtask

  virtual function void
          phase_ready_to_end(uvm_phase phase);
    if (ending)
      phase.raise_objection("Delaying End");
    fork
      delayed_end_of_phase(phase);
    join_none
  endtask
endclass
```

Figure 6. Phasing for the Legacy BFM

The mechanism shown above ensures that any transactions queued up in the BFM get driven on to the DUT interface before the environment proceeds out of the run phase. This happens even as the user drops all *run_phase* objections from the tests/sequences without needing to know if all the transactions have been really driven out from the BFM.

III. POLICY CLASSES AND PARAMETRIZATION

In this section we introduce an approach where existing verification infrastructure developed using methodologies such as VMM can be slightly modified and made suitable for use in any new methodology like UVM.

Each component in a methodology would be mapped to a different class in the library. For example, a transaction would map to a *vmm_data* in VMM, *uvm_sequence_item* in UVM and *ovm_sequence_item* in OVM. One of the requirements for a verification component to be reusable across methodologies is the ability to work with different data types. In SystemVerilog, classes support a single-inheritance model. Hence there is no facility that permits conformance of a class to multiple functional interfaces, such as the interface feature of Java. Classes can be parameterized by type, providing the basic function of C++ templates. A parameterized class in System Verilog is like a generic class whose objects can be

instantiated to have different array sizes or data types. This avoids writing similar code for each size or type and allows a single specification to be used for objects that are fundamentally different and not interchangeable. Parameterized classes in System Verilog are very similar to Templates in C++ along with some new and exciting tweaks.

Here is a Packet Stream class that was initially tied to a specific methodology for a scoreboarding application.

```
class vmm_sb_ds_pkt_stream;
  vmm_data pkts[$];

  int n_inserted   = 0;
  int n_matched    = 0;
  int n_mismatched = 0;
  int n_dropped    = 0;
  int n_not_found  = 0;
endclass
```

To have the same class to work with different applications, it can be made parameterized as follows

```
class vmm_sb_ds_pkt_stream #(type DATA = vmm_data);
  DATA pkts[$];

  int n_inserted   = 0;
  int n_matched    = 0;
  int n_mismatched = 0;
  int n_dropped    = 0;
  int n_not_found  = 0;
endclass
```

This is the first step in ensuring that your verification component can work with different data types. The next step would be to pass the appropriate parameter when using this class to define a class reference. That would involve the following transformation.

Non-parameterized:

```
class vmm_sb_ds;
  vmm_sb_ds_pkt_stream  pkt_stream;
  …
endclass
```

Parameterized version:

```
class vmm_sb_ds #(type INP=vmm_data, type
EXP=INP) ;
  vmm_sb_ds_pkt_stream #(EXP)pkt_stream;
  …
endclass
```

In the legacy component, the following statement would have created a reference which would only work for VMM transaction types

```
vmm_sb_ds sb;
```

With the parameterized version, the above statement will still retain the same behavior but to have an application which will work with UVM, all that would be required for the user would be pass the appropriate type.

```
vmm_sb_ds #(uvm_sequence_item) sb;
```

The same core component can be made to work with other data types. However, the functionality of a verification component is captured in the methods of the different classes that comprise the component. System Verilog does not allow function overloading and hence the same function or task cannot work with different data types. To enable functions to work with different data types, parameterization can be used again. The user code can be transformed as follows:

Legacy component function:

```
function bit vmm_sb_ds_typed::match(vmm_data actual,
                    vmm_data expected);
  return this.quick_compare(actual, expected);
endfunction: match
```

Transformed parameterized function:

```
function bit vmm_sb_ds_typed::match(EXP actual,
                    EXP expected);
  return this.quick_compare(actual, expected);
endfunction: match
```

Thus, enabling parameterization in verification components goes a long way in enabling a verification component or application to be interoperable with different data types. However, parameterization alone is not sufficient to address the requirements of complex applications especially when there are signature differences between the functions of the different parameterized specializations.

Here, in the legacy component, one of the display methods of the base class is used.

```
function bit vmm_sb_ds:
:expect_with_losses(..);
…
  if(!match)
    pkt.display("…");
endfunction
```

Now, even if the function is made parameterized, with 'EXP' being the parameter, the compilation will not go through if the parameterized specialization (let's say, uvm_sequence_item) does not have the '*display*' function as one of its methods.

```
function bit vmm_sb_ds::expect_with_losses( inout EXP
    data,..);
…
  if(!match)
    EXP.display("…");       endfunction
```

These kinds of scenarios require another common mechanism used in advanced software applications to enable patterns to be adapted to the situation at hand and to the specific problems being addressed. Policy classes are used for this purpose.

The central element in policy-based design is a class template taking several type parameters as input, which are specialized with types selected by the user (called policy classes), each implementing a particular implicit method (called a policy), and encapsulating some orthogonal (or mostly orthogonal) aspect of the behavior of the instantiated host class. In this example case, the 'policies' implemented by the policy classes could be the 'compare' and 'display' routines. By supplying a host class combined with a set of different, canned implementations for each policy, the host application can support all different behavior combinations, resolved at compile time, and selected by mixing and matching the different supplied policy classes in the instantiation of the host class template. Additionally, by writing a custom implementation of a given policy, a policy-based library can be used in situations requiring behaviors unforeseen by the library implementer.

A policy class implements a single static method, with a predefined name and signature. A canned policy class provides a trivial or sensible default implementation.

Example of a canned policy class:

```
class transformation_policy#(type IN = int, OUT = IN);
  static function OUT transform(IN a);
    return a;
  endfunction
endclass
```

Example of a policy class:

```
class encapsulate_ip_in_eth;
  static function eth_frame transform(ip_frame fr);
    eth_frame eth = new();
    ...
    return eth;
  endfunction
endclass
```

The host class calls the pre-defined policy method, using a type parameter to identify which policy class to use. The default value of the policy class is the canned policy class.

Example of a host class:

```
class transmogrifier #(type IN = int, OUT = IN,
                       transform_pol = transformation_policy#( IN, OUT))
  extends uvm_component;
  ...
  task run_phase(uvm_phase phase);
    forever begin
      IN in_tr;
      OUT out_tr;
      ...
      out_tr = transform_pol::transform(in_tr);
      #100;
      ...
    end
  endtask
endclass
```

To use a different policy from the canned policy, the host class is specialized using a user-specified policy class. For example:

```
typedef transmogrifier#(ip_frame, eth_frame,
                        encapsulate_ip_in_eth) eth_transmogrifier;
eth_transmogrifier trs = new("trs", this);
```

Going back to the original problem at hand, the first step would be to create a default policy class.

```
class vmm_data_object_policy;
  static function bit compare (vmm_data actual, vmm_data expected, ref string diff);
    return actual.compare(expected,diff);
  endfunction

  static function void display (vmm_data trans, string str = "");
    trans.display(str);
  endfunction

  static function string psdisplay (vmm_data trans, string str = "");
    return trans.psdisplay(str);
  endfunction

endclass : vmm_data_object_policy
```

The next step would be to redefine the application class to use the policy classes.

```
class vmm_sb_ds_typed #(type INP = vmm_data , type EXP = INP, object_policy = vmm_data_object_policy)
```

The next step would be to create the policy class required for the new flavor or methodology for which the existing application should still work.

```
class uvm_object_policy

  static function bit compare (ubus_transfer actual, ubus_transfer expected, ref string diff);
    if(actual.read_write == READ) begin
      return (actual.compare(expected));
    end
    else begin
      return 1;
    end
  endfunction

  static function void display (ubus_transfer trans, string str = "");
```

```
    trans.print();
  endfunction

  static function string psdisplay (ubus_transfer trans, string
str = "");
    return trans.sprint();
  endfunction

  endclass : uvm_object_policy
```

Once this is done, the existing application can be specialized with the new policy class as follows:

```
  typedef vmm_sb_ds_typed #(ubus_transfer,
  ubus_transfer, uvm_object_policy)
ubus_example_scoreboard;
```

The final step would be to invoke the missing method through the new policy class:

```
  function bit vmm_sb_ds::expect_with_losses( inout EXP
      data,..);
  …
    if(!match)
      object_policy::display("…");
  endfunction
```

Thus, with a few subtle changes and additions in the base application code, an existing verification application is easily transformed into one which can work with different base classes. This can work as a very simple solution in creating methodology agnostic applications in the test environment. It also serves the purpose of migrating legacy verification environment applications like scoreboards and reference models developed using reference methodologies like VMM to newer methodologies such as UVM.

IV SUMMARY

Given that there are multiple accepted and evolving methodologies when it comes to System Verilog test benches, developers creating verification components based on a specific methodology should seamlessly provide the same functionality to users conversant with a different methodology Through the policy based design approaches mentioned in this paper, two applications created using VMM centered around Performance Analysis and Scoreboarding were easily transformed and validated for UVM base classes. With the use of abstract classes, a verification infrastructure was easily created to leverage existing verilog BFMs in a UVM based environment and enabled an average effort saving of about 75% in effort and compute resources. Developing a UVM based environment and components from scratch would have easily consumed about 2 months of efforts with an addition al 2 weeks for its validation. Moreover the abstract class based interaction provides the other advantages such as access to all the properties of the underlying verilog based BFM including clocks, resets and events, complete abstraction of the UVM wrapper layer from lower level details and meeting all the requirements for a reusable OOP based verification environment. The approaches mentioned here would be very much relevant for verification environments and components developed using existing methodologies which need to be transformed and migrated to UVM.

V REFERENCES

[1] UVM 1.1a User Guide

[2] http://en.wikipedia.org/wiki/Policy -based_design

[3] VMM User Guide

[4] Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches , Dave Rich, Mentor Graphics

[5] From the Magician's Hat: Developing a Multi -Methodology PCIe Gen2 VIP, Amit Sharma , Varun S, Anoop Kumar, Abhisek Verma , Synopsys

[6] Verification Intellectual Property (VIP) Recommended Practices (http://www.accellera.org/activities/vip/VIP_1.0.pdf).