From the Magician's Hat: Developing a Multi-Methodology PCIe Gen2 VIP

Amit Sharma
Synopsys
RMZ Infinity, Bangalore
0091.80.40189129,
amits@synopsys.com

Abhisek Verma
Synopsys
RMZ Infinity, Bangalore
0091.80.40188614,
abhiv@synopsys.com

Varun S
Synopsys
RMZ Infinity, Bangalore
0091.80.40188424
svarun@synopsys.com

Anoop Kumar
Qualcomm
Whitefield, Bangalore
0091.80.39841800
anoopkr@qualcomm.com

ABSTRACT

In a few weeks, the Accellera VIP TSC will release the "1.0" version of the Universal Verification Methodology (UVM). This was the next step for the committee after it had released the UVM EA release early last year This has been quite significant because, the three major verification vendors have aligned on a single SystemVerilog Base-Class Library and Methodology for the first time. There have been several compelling methodologies that abstract away and compartmentalize many of the standard components used by the verification teams. The Verification Methodology Manual (VMM) was introduced in the year 2005 and since then it has been used in creating numerous multiple robust, reusable and scalable VIPs by different organizations. As UVM gains more and more acceptance, companies will look more seriously at being able take existing VIP from one methodology and use it with a UVM based flow and vice versa. To help in this regard, Accellera's VIP-TSC had earlier released the Verification Intellectual Property (VIP) Recommended Practices (http://www.accellera.org/activities/vip/VIP_1.0.pdf). This set of guidelines has been recently enhanced to aid in the interoperability between UVM and VMM components.

By using the examples of a PCIe Gen2 Synopsys DW VIP, this paper describes beyond the usage of the interoperability guidelines to not only show how a VMM VIP can be reused easily in an UVM environment but also demonstrates different novel techniques to completely transform the VIP into UVM using the VMM/UVM interoperability kit. The underlying transformations which the VMM/UVM interoperability package brings in is made totally transparent to the end user, so that the VIP can be seamlessly integrated in a UVM environment and used in a UVM context without requiring any understanding of the VMM methodology used to implement the underlying VIP. This paper covers in detail the technical challenges presented by the differences in the two methodologies. This includes the grouping of components, the phasing methodology and the generation of exceptions, sequences generation flow, using the factory infrastructure for augmenting functionality or replacing components and mapping the different callbacks infrastructure to each other.

The approach highlighted here is scalable to most VIPs. Some additional work may be required in mapping some VIP specific features to create counterparts of the transaction and configuration classes on the UVM side. There is an emphasis on automating the 'translate' layer which helps in translating functionality and capabilities from one methodology to the other. Finally, this paper discusses how to use the same techniques to transform UVM VIPs to VMM VIPs in order to ensure that the verification engineers have the most efficient means of integrating VIPs in any methodology and use it in the specific flavor they are comfortable in.

Categories and Subject Descriptors

Methodology – UVM, VMM, re-use philosophies for interoperable verification components.

General Terms

Verification, Methodology, Interoperability

Keywords

PCIe, VIP, VMM, UVM, Layered Protocol, SystemVerilog

1. INTRODUCTION

It has become a common practice for verification engineers to adopt advanced verification methodologies to create highly efficient transaction-level, constrained-random verification environments using SystemVerilog. These methodologies provide the framework for developing re-usable verification components, sub-environments and environments. There are multiple compelling solutions in this space now with the Universal Verification Methodology (UVM) being the accepted standard. Verification IP vendors who have already made an investment in creating a VIP around a specific methodology would necessarily want to cater to a bigger clientele by being able to provide the same VIP in different flavors, i.e. by having a mechanism to make it reusable across methodologies. Developing complex VIPs from scratch can be a time consuming exercise and often the new component is neither reliable nor mature enough to ensure correctness. Hence, the ideal scenarios for these vendors would be the ability to provide these new flavors of the VIP by just spending a fraction of the time they would have otherwise taken to develop it from scratch.

Each methodology provides a different set of guidelines and Application Programming Interfaces (APIs) for the user to leverage the SystemVerilog language to target various verification requirements efficiently. Given that each methodology is different in their own ways, verification engineers are seldom expected to master all of them. Hence, while delivering an existing VIP in a new methodology, the vendor has to ensure that the user need not understand the methodology used to implement the existing VIP.
.

The VMM-UVM interoperability library includes a collection of adapters and utilities that enable easy and flexible reuse of existing IP in both UVM and VMM environments. While both UVM and VMM are self-consistent and provide guidelines and technology to ensure reusability, trying to use a VMM VIP in a UVM testbench (or vice-versa) exposes some of the different philosophies they hold. There are trade-offs in reusing an existing VIP block in a new environment that is based on a different methodology. This paper leverages and goes beyond Accellera's interoperability ideology; which is to enable the re-use of VMM VIP components in UVM (and vice versa) and shows how you can package an industry proven VIP to transform it wholly to assume a new methodology leaving no traces of the base methodology actually used to implement it.. This paper takes the example of the industry proven DesignWare PCIe Gen2 VMM VIP and enumerates the steps that are required to transform it into a UVM VIP. We also then deliberate on how the reverse process can be achieved.

2. PCIe PROTOCOL OVERVIEW

The PCI Express protocol uses transaction level packets to communicate information between components. Transactions are formed in the Transaction and Data Link Layers and helps carry the information from the transmitting component to the receiving component. As the transmitted transactions flow through the other layers, they are extended with additional information necessary to handle transactions at those layers. At the receiving side, the reverse process occurs and transactions get transformed from their Physical Layer representation to the Data Link Layer representation and finally (for Transaction Layer Packets) to the form that can be processed by the Transaction Layer of the receiving device. Figure 1 shows the conceptual flow of transaction level packet information through the layers. Though, a device design does not have to implement a layered architecture as long as the functionality required by the specification is supported, it is advisable for a VIP vendor to provide all these layers, thus enabling the user to configure the stimulus and response at all these layers.

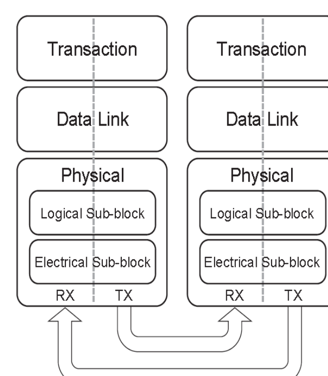

Figure 1: Multi-Layered PCIe Protocol Overview

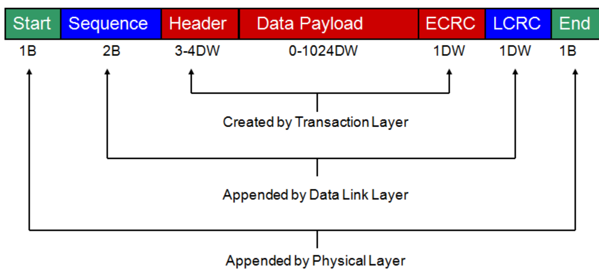

Figure 2: TLP Transaction Creation

The PCIe protocol brings in a lot of benefits from the previous generation bus architectures and introduces an advanced set of capabilities which include the following amongst others:

- Traffic Class and Virtual Channel Applications
- Flow Control Initialization /operation
- Error Checking mechanics/reporting
  Message transactions
- PCIe Enum procedure
- Quality Of Service (QoS) features for differentiated transmission performance

The above functionalities require the VIP vendor to provide the users with the capability to configure the VIP in multiple modes, enabling it to generate stimulus and exceptions to validate all the above functionalities in a device. Also, the difficulty in verifying a PCIe device arises from the inherent complexity of the protocol. Hence, it is very important to ensure that the methodologies are adopted in the most optimal manner to alleviate any bottlenecks to efficiently verify the same.

3. PCIe VMM VIP ARCHITECTURE

Synopsys' Verification IP (VIP) for PCI Express® has been used across multiple sites over many years and it provides a quick and efficient way to verify system-on-chip (SoC) designs with a PCI Express (PCIe) interface. The VIP for PCI Express enables verification of PCI Express 3.0, 2.0 and 1.1 Endpoints, Switches, and Root Complex devices at the 8b/10b, PIPE or Serial interface. It provides the capability to generate and control transactions at each of the TLP, DLLP and PHY layers.
The base Verification IP that we used in our example was a VMM based PCIE VIP that facilitates constrainable random generation of data while maintaining a flexible directed capability. It allows easy integration into a verification environment.

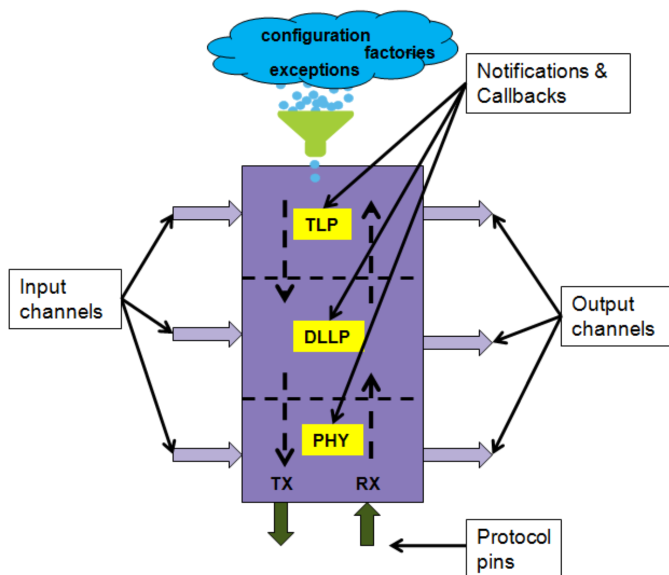

Figure 3: PCIe Gen2 VMM VIP Architecture

3.1 VIP Usage and Configurability

The following features of the VMM VIP are required to be available on the UVM side

Configuration: The VMM 1.0 based VIP has a configuration class which is shared across all components. The typical flow is for the class to be randomized at vmm_env::gen_cfg phase and then the passed down to different components.

If a user needs to change the configuration properties in specific tests, then it requires either of the two following mechanisms:

- Calling gen_cfg() and then modifying the values
- Setting constraints on a derived configuration class and overriding the configuration class in the environment

Stimulus Generation: To stay consistent with the architecture of the PCIe standard a layered approach has been adopted by the VMM VIP with transaction classes for each of these layers. These are typical VMM data descriptors with capabilities of constrained random generation which will translate to protocol specified header data. The generator classes provided with the VIP are VMM macro based atomic and scenario generator which can be used to generate random stimulus.

Transaction Level Interfaces: VMM channels are provided to the user to interface to different VIP blocks. The layering here mean, the channels at each of these layers providing the user with the flexibility of accessing the data as it flows through the layers.

Extension Points: A rich set of callbacks are provided across the different layers enabling the user to create implementations for coverage, scoreboard etc.

Data Exceptions: The transaction level interfaces and extension points can additionally be used for corrupting stimulus for negative tests on the device being verified. A number of exception data classes have been defined within the VIP library for this purpose.

Factory Overrides: The VIP provides the user with the benefit of overriding behavior that will help meet the unpredictable needs of different tests.

Event Synchronization: A number of VMM events are also provided so that the users can use for the purpose of decision making to synchronize their testbench with the transition of data or states within the VIP. Most notifications are tied to the PCIe standard but there are a few that are generic to the VMM methodology, like notifications from the data class (STARTED & ENDED).

4. THE VMM UVM INTEROPERABILITY LIBRARY

The Accellera Recommended Practices show how to integrate VIPs and testbench components independently implemented using UVM and VMM into a heterogeneous verification environment. It provides solutions for the various challenges a verification architect faces when integrating a foreign methodology VIP. It also contains a chapter containing the application programming interfaces (APIs) associated with Recommended Practices, thus specifying a library of adapter components. The VCS installations and the VMMcentral.org site has a slightly enhanced version of the same which documents how VMM UVM interoperability can be achieved with the new Transaction Level Interfaces like TLM2.0 and Implicit Phasing in VMM1.2.
In a nutshell, this library addresses the following challenges the user may experience when attempting to reuse a VIP written in a different methodology [4]:

- Instantiating and building of the component within the environment
- Coordinating different simulation phases
- Configuring components to operate properly in the desired context
- Orchestrating and coordinating stimulus and other traffic between components
- Passing data types between components
- Distributing notifications across the environment
- Issuing and controlling messages

4.1 Library Topologies

There are two reuse models of VIPs written in one methodology in an environment written in another. One is the “interconnected” model which enables taking the “foreign” VIP component and using it the environment along with other “native” VIPs. In the “interconnected” model, the user is aware that the VIP is implemented in a different methodology. The other model is the “encapsulated” model which requires wrapping the “foreign” VIP component within a “native” wrapper, so the user is not aware when he/she is using a VIP component originally implemented in a different methodology. The guidelines specified in VIP Recommended Practices supports the ‘interconnected model’

4.2 Applicability of the Interoperability Library for Delivering VIPs

VIP vendors will undoubtedly be interested in delivering their VMM-based VIPs to UVM customers. This is preferably done with all UVM features and mechanism applied as is to the VIP so the user is not required to know that the VIP is implemented using VMM. From an ease-of-use perspective, VIP providers would prefer to use the "encapsulated" model.

In DesignWare VIPs, the element of reuse are the individual transactors, not higher-level components like *vmm_subenv* or top level *vmm_env* which encapsulates different transactors.A key aspect of the "encapsulated" model is to ensure that even the adapter components from the UVM/VMM interoperability library are not exposed to users, not even the verification architects or the integrators. There were a few additional aspects which the Recommended Practices didn't address, like the use of callbacks on the UVM side. Hence, a few other additional solutions had to be incorporated on top of the existing APIs and Recommended Practices to meet these requirements.

.

5. PCIe UVM VIP ENVIRONMENT

The section describes how we used the UVM/VMM interoperability library to implement the UVM encapsulation wrapper for the VMM VIP. We ensured that the environment complies with the UVM methodology and enables the end user to leverage all the functionality with regards to configurability, introspection, generation, coordination etc that the VIP provides.

High Level Architecture:

A UVM agent encapsulates a sequencer, a driver and a monitor. We took the existing core VMM VIP transactors, namely the PCIe TxRx/ driver and PCIe monitor, and encapsulated them inside a UVM driver and monitor respectively. Thus, a UVM user is able to use sequences for stimulus generation and to have a standard mechanism to interact between the sequencer and the driver, according to the UVM methodology principles.

.

The bulk of the VIP functionality is confined to the VMM transactor/monitor and the UVM driver/monitor manage the relaying and conversion of transactions from either side using the Interoperability Kit as the bridge. Each of the layers of the PCIe UVM driver provides *uvm_sequence_item* pull ports for hooking its corresponding sequencer. In addition to these ports the driver also provides analysis ports to broadcast transactions when in 'receive' mode. Similarly the UVM monitor wrapper provides for analysis ports at each layer to broadcast transactions to be used for various purposes like coverage collection and scoreboarding. The UVM agent additionally has its set of own configurations as illustrated in the figure below.

The DUT and the driver/monitor are connected through a virtual to physical interface connection as discussed under section 5.3.4.2.

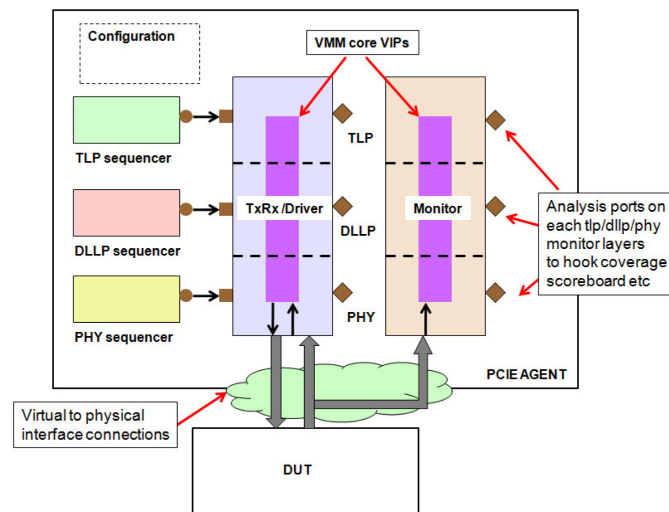

Figure 3: Multi-layered PCIe UVM Agent

5.1 The Interoperability Layer

The flow of data models in the PCIe UVM/VMM VIP using the interoperability adapters is illustrated in figure 4. The input channels of the VMM VIP on each of the TLP/DLLP/PHY layers (tx_in_chan) are mapped to sequence item pull ports (tx_in_port) in the UVM wrapper driver class using the tlm2channel adapters. Similarly, the output channels of the VMM VIP (rx_out_chan) are mapped to output analysis ports (rx_out_aport) using the channel2tlm adapters. This enables a UVM user to connect the tx_in_port on each of the layers to a type compatible sequencer, i.e. a TLP tx_in_port is connected to a TLP sequencer through a port-export connection. Similar mappings are made for the lower layers.

Thus a TLP sequence item on the requestor VIP starts from the TLP sequencer, enters the UVM VIP through the tx_in_port gets converted to a VMM transaction by the convertor class in the interoperability layer and is posted on the tx_in_chan or the input channel of the VMM VIP TLP layer. Once posted to the input channel, the VMM VIP takes over and drives the transaction appropriately at the DUT interface. Similarly, on the 'completion' side once the TLP transaction is received by the VMM VIP TLP, it is posted on the rx_out_chan or the output channel, which then gets converted to UVM sequence item by the interoperability layer and is broadcasted onto the rx_out_aport analysis port for any subscriber to feed on it.

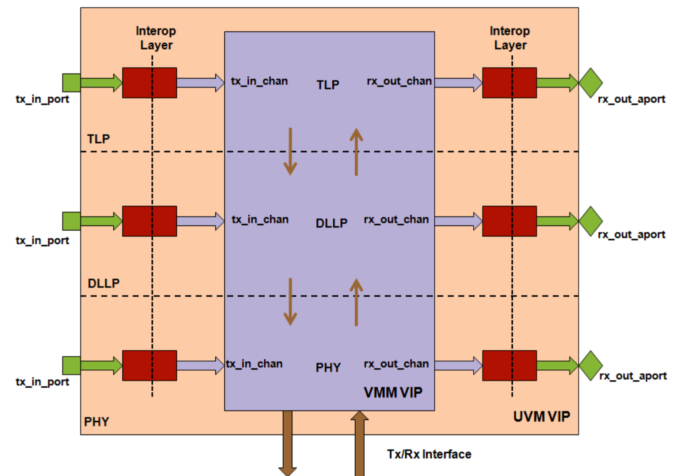

Figure 4: Data Flow Through the Layered PCIe VIP

5.2 Data Modeling

The PCIe VIP comprises the following data models -

i. Configuration classes
ii. Transaction classes for each of the three layers
iii. Exception classes, to inject errors into each layers

The interoperability layer requires these data models in both UVM and VMM and hence, the UVM transaction descriptor classes were modeled with the same members as in the VMM data descriptor. Given that the transaction methods in the original transactions in VMM had some custom logic, the same was achieved in UVM using the virtual methods: do_copy(), do_compare() etc. As required by the interoperability kit, we created the 'translation' implementation which does explicit data mapping between data fields of the transaction descriptors across the two methodologies. This requires creation of a separate converter class which defines a static "convert" function which is implicitly sourced by a number of blocks to transform transaction descriptors across the two base libraries. However, this does not allow declarative methods such as constraints and user-defined methods to be converted: functionality-equivalent declarations need to exist in the destination type.

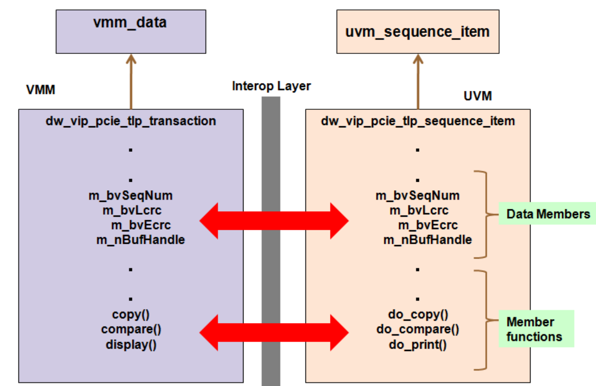

Figure 5: Transaction Mapping Between VMM and UVM

Transaction descriptors constitute the blood stream of a testbench and it becomes critical to identify each descriptor for different verification requirements. Thus we have devised a way to tie each

transaction descriptor on the UVM side to its counterpart on the VMM side as shown in Figure 6.

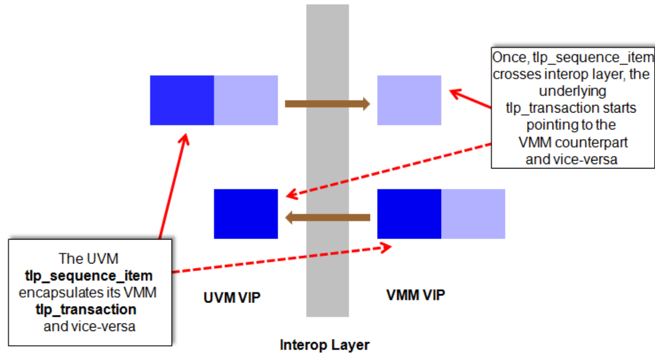

Figure 6: Transaction Refernces Between Data Models

Within every UVM transaction descriptor we maintain a handle to its VMM counterpart. This handle will be updated by the 'convert' function and this makes it possible to elevate features like notifications from the underlying VMM layer to the UVM layer as will be described later.

5.2.1 Automation

Given that there would be a plethora of data models in complex VIPs encompassing transaction, configuration and exception classes, it is important that some automation is brought in to generate the counterparts of these classes in the 'other' methodology as well as create the static convert classes. For each VMM class, this automation was achieved as shown in figure 7.

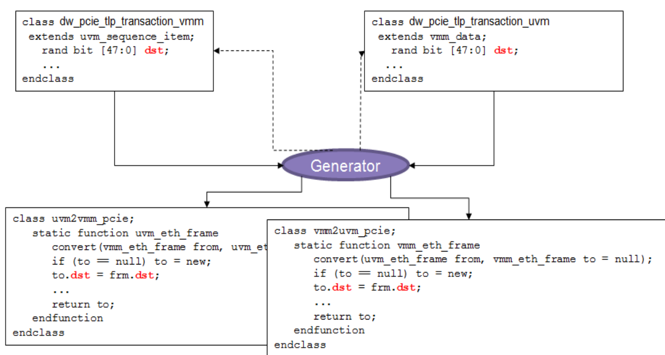

Figure 7: Automation for Generating Data Models

5.3 Delivering Capabilities in UVM

This section describes how we helped deliver all the functionality that the VMM VIP had to the UVM user.

5.3.1 VIP Phasing and Component Synchronization

The phase synchronization provided by the Accellera interoperability kit was not used here. Instead, the UVM wrappers for the driver and monitor were responsible for explicitly phasing the underlying VMM VIP as shown below in the figure. For example- the run phase of the UVM wrapper VIP would make an explicit call to the *start_xactor()* method of the underlying VMM VIP. This enabled fine control of the underlying VMM VIP from the UVM side. This also ensured a standard UVM architecture with no new interoperability classes used for the top-level environment.

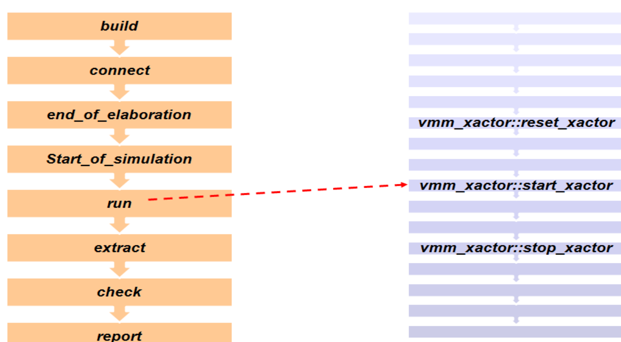

Figure 8: VIP Phasing

5.3.2 Notifications

The AVT library provided by Accellera provides a way to capture the VMM data ENDED notification onto the UVM side. This facility has been provided so that a UVM source node is aware of all completions signaled by the VMM node. The base adapter can be configured to do so and the adapter will post VMM transactions that 'END' onto a VMM response channel connected to it. This technique involves the configuration of the TLM to Channel adapter blocks to return all ENDED data descriptors through a response VMM channel. Figure 9 illustrates the same.

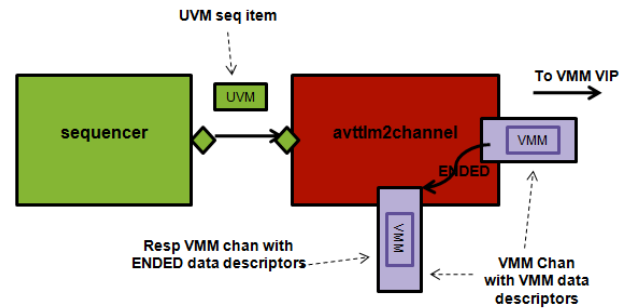

Figure 9: Solution for Relaying 'ENDED' Notifications

The technique explained above has its own limitations. Firstly, it is limited to just the VMM data ENDED notification. Secondly, the UVM testbench is notified using a VMM data descriptor which is not understood by a UVM block. To solve the latter we can connect the ENDED response channel to another convert block that will make it available at the UVM domain. However, the first limitation would still hold and there is no generic way to solve this.
The above limitations pushed us to devising better ways to map all the notifications defined within the VMM VIP that can be easily used by a UVM testbench.

We employed UVM events for this purpose mapping each notify event defined in VMM to a corresponding event in UVM. As notifications in VMM are tied to each instance of a class it necessitated associating every VMM object with its UVM counterpart. For data objects this is being taken care of by the convert layer which passes the handle of the transformed VMM data object onto the source UVM object. With this kind of a mirror association established between the VMM & UVM counterparts, the "vmm_notify_callbacks::indicate()" was used to trigger the corresponding event defined in UVM.

We had to deal with two categories of the following notifications:

1. **Data notifications**: These were pre-defined from the VMM base library namely ENDED, STARTED and EXECUTE.
2. **Transactor notifications**: These included a mix of pre-defined as well as VIP specific notifications. The pre-defined notifications such as, XACTOR_IDLE, XACTOR_STARTED, XACTOR_STOPPED etc. Besides these, there were a total of eight notification events defined within the VIP driver and three in the monitor.

For data notifications, we used the *begin_tr*() and *end_tr*() functions defined within the uvm_transaction class of the UVM base library. These functions trigger internal events defined within the transaction class namely "begin" and "end" respectively.

```
class       dw_vip_pcie_tlp_transaction_ended_notify_callbacks
extends vmm_notify_callbacks;
dw_vip_pcie_tlp_transaction_uvm uvm_tr;

  function new(dw_vip_pcie_tlp_transaction_uvm uvm_tr);
    this.uvm_tr = uvm_tr;
  endfunction

  virtual function void indicated(vmm_data status);
       this.uvm_tr.end_tr();
  endfunction
endclass
```

Figure 10: Notify Callback Triggering UVM Event

The convert layer takes care of creating a mirror like association between VMM and UVM data descriptors as illustrated in the following code snippet:

```
class dw_vip_pcie_tlp_transaction_convert_uvm2vmm;

  static function dw_vip_pcie_tlp_transaction convert(
                              dw_vip_pcie_tlp_transaction_uvm
                                                         from,
                              dw_vip_pcie_tlp_transaction
                                                       to=null);

     dw_vip_pcie_tlp_transaction_ended_notify_callbacks

                                                  tlp_ended_cb;
     ….

     from.vmm_tr = to;

     tlp_ended_cb  =  new(from);
     to.notify.append_callback(vmm_data::ENDED,
                                              tlp_ended_cb);
   endfunction
endclass
```

Figure 11: Registering Callback with the VMM Transaction

We used UVM event pools to access these events from the testbench. By using the data pools, we can get the handle of the event that we are interested in and use these event handles to wait for the event triggers.

```
    tlp_event_pool  =  cfg_wr_xact.get_event_pool();
    tlp_ended_event =  tlp_event_pool.get("end");
    tlp_ended_event.wait_trigger();
```

Figure 12: Event Usage within the UVM Testbench

For the driver and monitor events we had to define new UVM events as these were all notifications specific to the IP. However, from a usage perspective nothing changed and these were used exactly the same way the events are used in UVM.

5.3.3 Stimulus Generation

The generation of stimulus and its lifecycle for the PCIe driver in transmit mode can be illustrated by the following diagram.

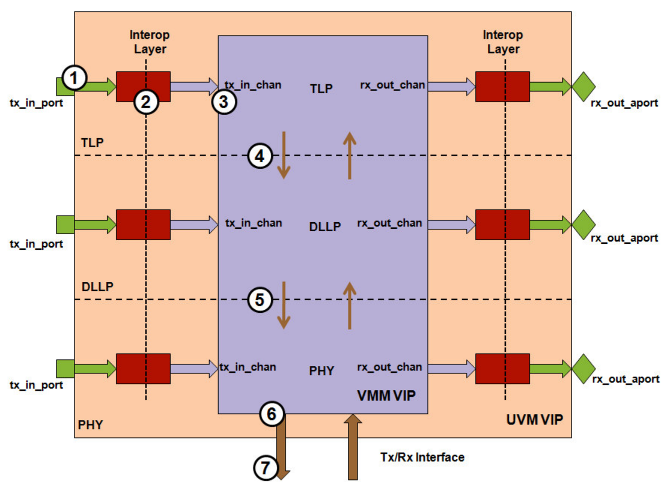

Figure 13: Stimulus Generation

Referring to the numbered bullets in the figure above,

1. The Transaction Layer (TL) sequencer generates TL packets and posts them onto the tx_in_port
2. The Interoperability layer converts the UVM sequence item into a VMM transaction and posts it to the underlying VMM channel
3. The VMM VIP gets the VMM transaction and starts processing it and transmits it to the layer below which is the Data Link Layer (DLL)
4. The TL packet flows down to the DLL
5. DLL processes the inbound TL packets and adds the DLL header fields onto the same and transmits it to the PHY layer
6. The PHY layer adds its header to the same and the packet is now ready to be transmitted on the interface pins
7. Data is transmitted on the protocol pins to be sampled by the device connected on the other side

The layered consistency was maintained with the UVM wrapper with ports provided at the DL & PHY layers enabling VIP users to stimulate the model from the lower layers as well. The same interoperability mechanism was used at the lower layers to connect the transaction interfaces across the two methodologies.

With this interoperability infrastructure in place, VIP could now be interfaced with UVM sequencers. We then created a library of sequence classes that was mapped from an existing library of VMM scenarios.

Figure 14 shows the scenario registration from one of our VMM test cases and Figure 15 shows the mapping to a UVM test sequence.

```
VMM test:
….
// Procedural task that configures the BAR registers of the DUT
pre_condition_cfg();

  mem_wr_scenario  = new();
  mem_rd_scenario   = new();
  rand_scenario        = new();

  // Registering the VMM scenarios with the generator.
  env.scenario_gen.scenario_set.push_back(mem_wr_scenario);
  env.scenario_gen.scenario_set.push_back(mem_rd_scenario);
  env.scenario_gen.scenario_set.push_back(rand_scenario);
  …
  env.run();
```

Figure 14: VMM Test Showing Scenario Registration

```
UVM test sequence:
….
  // body() - Stimulus input
  virtual task body();
    uvm_test_done.raise_objection(this);

   // precond_cfg_seq run only once for configuring the
   // BAR registers of the DUT
    `uvm_do(precond_cfg_seq)

   mem_wr_seq  = get_seq_kind("pcie_mem_wr_seq");
   mem_rd_seq   = get_seq_kind("pcie_mem_rd_seq");
   rand_seq        = get_seq_kind("pcie_rand_seq");

   for (int i=1; i<p_sequencer.count; i++) begin
         assert(randomize(seq_kind) with {seq_kind dist {
                mem_wr_seq := mem_wr_seq_wt;
                mem_rd_seq := mem_rd_seq_wt;
                rand_seq       := rand_seq_wt;

                };
         });

         do_sequence_kind(seq_kind);
   end
   uvm_test_done.drop_objection(this);
```

Figure 15: UVM Test Sequence Class

You can see how the procedural configuration sequence "pre_condition_config()" defined in VMM is translated into a UVM sequence. Also an additional constraint is set using sequence "kinds" to ensure that the configuration sequence is driven only once. All the UVM sequencer API's now become available to the user with aid of which one can develop a re-usable scenario library. The agent class provides a provision to select one of three sequencers using a configuration string. Also the objection mechanism to allow hierarchical communication of status among components also becomes available to the user.

5.3.4 Configurability and Overrides

The base VIP based on VMM 1.0 which was configurable only during construction. With this new wrapper being added we brought in the dynamic configurability using the factory infrastructure. The UVM layer also brought in the global factory infrastructure allowing component replacement using set_type_override_by_type()/ set_inst_override_by_type() methods.

5.3.4.1 UVM Configuration

The configuration infrastructure is scoped in the build() phase of the UVM VIP.As the build phases in a top-down manner, it allowed configurations to be set from within the build() phase of all its parent components . This ensured flexibility of configuring the VIP from within the UVM test. Once the configurations were made available to the UVM driver/monitor wrapper via set_config_*/get_config_* mechanism, the underlying VMM VIP received the same through its constructor. The connection to the physical interface is also achieved in a similar way with the virtual interface wrapped in a container class. All that the user has to do is to initialize the virtual interface and set the same from the test case.

5.3.4.2 Factories

The lightweight UVM wrapper brought in the UVM component replacement mechanism using the UVM global factory. This came with a limitation that declarative methods and constraints wouldn't show up on the VMM side. However this was acceptable to our users as factories were mainly being used for sequence generation for the purpose of adding test constraints. As the randomization was limited to the UVM scope our requirements were completely satisfied.

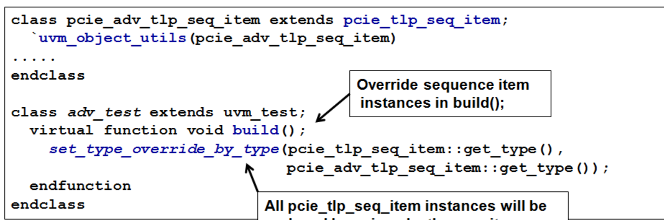

Figure 16: PCIe UVM Callbacks Implementation

5.3.5 Extension Points (Callbacks)

Figure 17 explains how callbacks infrastructure of the VMM VIP has been efficiently deployed to provide callbacks infrastructure on the UVM side without the user being aware of the VMM callbacks underneath. The UVM VIP provides a façade class which is the mapping of VIP VMM callback façade class. The user can extend the UVM façade class and implement his custom functionality and then subsequently append them to the UVM VIP. The underlying VMM VIP is always appended with the VMM callbacks. The implementation of the VMM callback tasks converts the VMM transaction handle made available to the callbacks to a UVM data model using the 'convert' function. It then calls the equivalent UVM callback tasks, which provides the hook for the UVM user to process the UVM sequence item. Finally the VMM callback task converts the UVM sequence item back into a VMM transaction for the VMM VIP. This causes any changes made on the data descriptor on the UVM callback to be propagated to the VMM transaction on the VMM VIP.

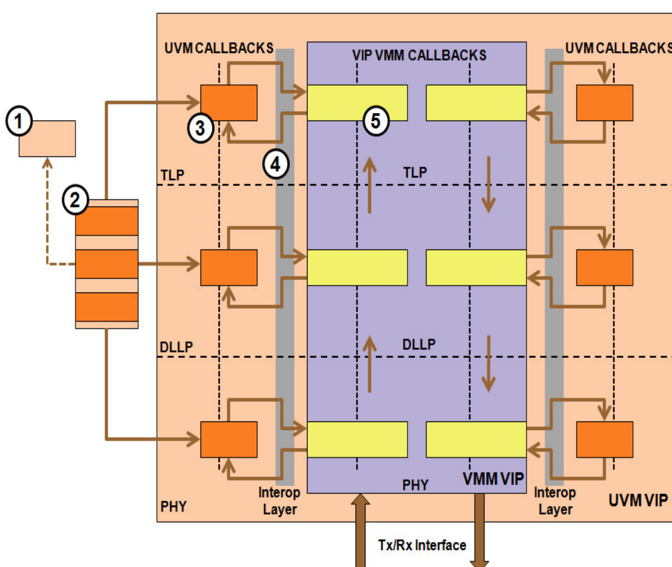

Figure 17: PCIe UVM Callbacks Implementation

From the figure above, we can note the steps involved:

1. UVM façade class mapped from the VMM VIP façade class
2. User UVM callbacks class, extended from the UVM façade class, implements the callback tasks
3. The UVM callback task, which gets triggered by the VMM callback.
4. The interoperability layer to convert VMM transaction to UVM sequence item and vice versa.
5. VMM VIP callback tasks, appended to the VMM VIP, once it is created.(always on)

The figure also shows the callbacks on various layers of PCIe VIP, namely the TLP, DLLP and PHY. It also tries to depict the flow of data through the callbacks while the VIP tries to Transmit or receive.

5.3.6 Message Service

The VMM/UVM interoperability mechanism was used to ensure all the VIP messages come though the UVM report server. For specific scenarios where more control was required with respect to the modification of messages, the UVM log catcher was used to catch the messages and then to make appropriate changes in formatting or verbosity.

5.3.7 Scoreboarding and Register Validation

We demonstrated the use of the UVM comparator with UVM VIP in a number of our legacy test cases. Previously we had employed the VMM DataStream Scoreboard which was replaced with the UVM comparator by simply connecting the exports of the comparator with the analysis ports of the newly created UVM monitors. The integration was as simple as just connecting the correct analysis export of the comparator to the analysis port of the appropriate monitor and we had a very simple in-order comparator.

We understood that using the UVM comparator was appropriate from a unified methodology perspective. At the same time we were losing out on the advanced features provided by the VMM DataStream Scoreboard like the "out of order" compare, with "expect with losses", the reporting facility etcetera. To appease that concern we even demonstrated using the VMM DataStream Scoreboard within our new UVM testbench. This was achieved by connecting the analysis exports provided within the VMM scoreboard to the analysis ports of the UVM VIP. For doing this an additional 'convert' layer was needed to convert the UVM transaction descriptors being posted by the UVM monitors to VMM data descriptors to be inserted and compared within the VMM scoreboard. This is illustrated in the following figure:

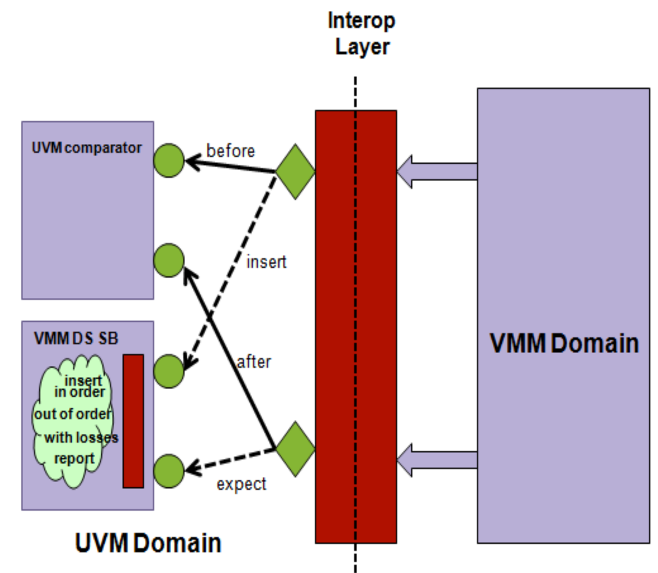

Figure 18: DS Scoreboard/UVM Comparator Integration

With RAL becoming a part of UVM with the UVM 1.0 release, similar Register Access APIs can now be used for both the flavors of VIP. Minimal changes were needed here with the old RAL translator being replaced with a UVM sequence. This allowed us to port all our VMM RAL checker test cases with relative ease.

5.3.9 Limitations

Accellera's VMM-UVM interoperability kit was appropriately used to architect our wrapper VIP, which very closely resembles a native UVM VIP. The development time was accelerated and we came very close to our objective, however, there were a few limitations that we had to inevitably deal.

As the 'convert' layer is static with the base transactions, configurations and the exceptions of the VIP, any new member added in the derived classes of the same, do not reflect on the VMM core of the VIP. In case a derived class is being used on the UVM side, it gets created, randomized but then posted as the base type to the VMM VIP core.

The VMM VIP core also provides for a mechanism to inject errors by overriding a 'factory' object through its constructor. The mechanism to inject errors or create exceptions in such a scenario from the UVM side would be to resolve all the dependencies and randomizations on the UVM side itself and then pass it as the 'base' type to the VMM core again through the 'convert' layer. However, the VMM VIP will always have the handle of the base transaction type, even if a derived class was being used on the UVM side for the transaction.

With the given architecture, an additional layer is created and the data models have to be passed to and fro between the UVM and VMM layers. This can potentially cause degradation in simulation performance compared to a native VIP. The VMM VIP once created or built always has callbacks on all the three layers appended to it during the simulation thus impacting simulation performance adversely. We have now created a mechanism to turn them on only when it is required by a runtime configuration option.

6. REVERSE ENGINEERING THE FLOW: Delivering a VMM Wrapper over a UVM VIP

Most of the techniques described above would hold true while creating a VMM flavor for an existing UVM VIP. In some cases, it will be easier. For example, unlike the specific DW VMM VIP, UVM guidelines profess the delivery of a VIP as an UVM agent which encapsulates the individual components. Thus the phase synchronization mechanism in the VMM on TOP topology can easily be used to phase a UVM VIP in a VMM 1.2 timeline as shown in Figure 19. However, for implicit phasing, there is some additional rework that needs to be done on top of the Accellera's interoperability layer and the modified library and is available for download at VMMCentral.org.

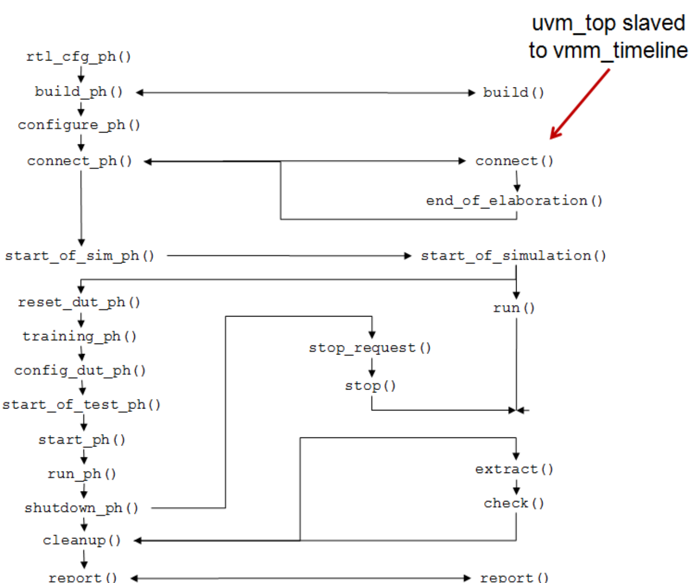

Figure 19: Phase Synchronization Between a VMM on Top Environment and a UVM VIP with Implcit Phasing

The following table summarizes the additional effort if any that would be required for creating a VMM based flavor on top of a UVM VIP.

Table 1: VMM test showing scenario registration

Feature	What needs to be done?
Sequences	Similar UVM to VMM convert mechanism to handle data modifications.
Phasing	Accellera's phase synchronization with VMM on top topology.
Factories	Consistent with our solution with the limitation of declarative methods and constraints not being added.
Notifications	Any UVM events can be mapped to notifications
Callbacks/Extension points	Our existing solution can be used here and can be mapped accordingly
Message Service	Accellera's interoperability mechanism with VMM on top topology.

7. RESULTS

It took about two man months to complete the whole activity which included, validating our new VIP's by porting our legacy test cases and examples to UVM. This was insignificant compared to the development time of a complex VIP like PCIE which would typically span over a period of four to five man years. Most importantly, we are assured of the quality as our base is an industry proven VIP that has evolved over the years.

The developed wrapper files had about 30,000 lines of code in total. A good percentage of these were generated using scripts which meant even lesser number lines of user written code. This reused the underlying infrastructure of the VMM VIP. Vendors delivering multiple VIP typically create a base infrastructure so that the use model and architecture is consistent across different IPs. Taking this base infrastructure along with the VIP specific code would typically run into hundreds of thousands of lines of code if not millions. Thus, not only did this exercise help us in meeting our requirements of delivering and integrating this VIP, it also helped us in meeting our goals much more efficiently when compared to the option of creating a VIP from scratch.

8. CONCLUSION

Given that there are multiple accepted and evolving methodologies, when it comes to SystemVerilog testbenches, vendors creating Verification IP's based on a specific methodology should have a mechanism to provide the VIPs functionality to users conversant with a different methodology. Also, vendors cannot possibly make the same big investments for multiple methodologies.

The 'Verification Intellectual Property (VIP) Recommended Practices" from Accellera and the associated base classes go a long way to make VIPs interoperable. The techniques documented in this paper are a result of our efforts to meet a requirement to provide a UVM VIP to completely match the capabilities of our proven VMM based VIP. We managed to achieve this infew weeks and also demonstrated the VIP usage in its new avatar. We leveraged the same base classes that is provided in the interoperability kit and enhanced them with additional capabilities to meet out requirements for having a proven VIP ready in UVM in quick time. While vendors as well as developers of block-level testbenches can use these techniques to make their VIPs interoperable, they can also understand the requirements for VIPs in either methodology through the various challenges and requirements demonstrated in this paper.

The guidelines will hold true for creating a new VIP from scratch. For delivering multi-methodology environments, they can create their own standard base layer that can be underpinned to make the VIP being leveraged by both VMM and UVM users. This brings an added advantage of layering the architecture of the IP making it easy to maintain. Moreover the vendor specific standard can be used across multiple VIP titles maintaining a consistent architecture.

10. REFERENCES

[1] UVM User Guide

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

[3] DesignWare PCIE VIP User Guide

[4] Mindshare Inc., Ravi Budruk, Don Anderson,Tom Shanley,PCIe PCI Express System Architecture

[5] VMM User Guide