Wrapping Verilog Bus Functional Model (BFM) and RTL as Drivers in Customized UVM VIP Using Abstract Classes

Roman Wang
+8613482890029, roman.wang@amd.com
AMD Shanghai, China

Thomas Bodmer
AMD Sunnyvale, USA

***Abstract*- With the increasing complexity of design and verification requirements, more and more verification teams adopted the UVM methodology to build their testbench. UVM could bring a highly reusable, scalable, extensible and configurable framework to engineers to shorten the verification schedule and ensure quality. UVM verification IP (VIP) usually stands for a specific standard protocol interface UVM verification component (iUVC), and could be easily reused to different level of UVM verification testbenches (IP or SoC).**

In general, the verification team integrates in-house or 3rd part UVM VIP to verify the standard bus protocol. However, there may be no available UVM VIP for a particular bus protocol, and an engineer has to develop the customized UVM VIP as specific requirements.

In special cases, the master IP is connected to different slave IP through a shared bus, where timing is particular and a little bit complex. This master IP with bus interface had been proved in past projects at the SoC level. When the IP team wants to verify a new IP with this type of shared particular bus interface that will be connected to the same master IP at the SoC level, the customized UVM VIP is necessary to help verify the new IP at the UVM stand-alone level at an early stage. With such requirements, the master IP designer could decouple the RTL into a clean Verilog BFM model with two bus interfaces: one is to drive the new slave IP and easy, and another is simply to get drive from others. The efficient way to create a customized UVM VIP is to reuse and wrap the Verilog BFM model inside and use abstract classes to define functions that easily communicate BFM with the VIP driver. The SystemVerilog (SV) interfaces are bind between BFM and UVM VIP to make sure that BFM is invisible from the outside. The user could treat this UVM VIP as the general UVM VIP. In this way, when the master IP designer updates the bus timing, the UVM VIP could sync up the decoupled BFM without any change.

This paper will introduce a proposal to deploy this kind of customized UVM VIP, and discuss the reuse considerations. Users could extend this approach to implement other similar scenarios.

Keywords - Customized UVM VIP, Verilog BFM, SV Interface, Bind Methodology, Abstract Class

I. INTRODUCTION

With the increasing scale and complexity of design, the requirements of VIP are becoming critical for verification team to reduce the verification cycle and improve verification quality. In the traditional HDL based verification testbench, verification engineers create the Verilog BFM VIP to mimic a specific bus protocol (such as SRAM and DDR models) and integrate BFM in the legacy verification environment. The Verilog BFM has several common functions and tasks which could be directly called by verilog test code. These modular application program interfaces (API) are usually used for BFM initial, driving bus, capturing bus, etc. With the great UVM methodology adoption in recent years, more and more verification environments are built by UVM. UVM defines the reusable framework, versatile APIs and coding guidelines for users to create and integrate UVM UVCs. The interface UVC is a reusable UVM interface VIP, and acts in the same role as the legacy Verilog BFM. It's usually hard to create the iUVC from scratch to meet the tough verification schedule, so it's really important to reuse the legacy and leverage the effort and stability into the iUVC VIP. To make the iUVC VIP easy to use and straight forward, it's also important to hide all of the implementations from the end users point of view. As the engineer's experience [1], the legacy Verilog BFM could be connected to the UVM driver by calling the built-in APIs of BFM. In this way, it really makes the proved Verilog BFM model able to be reused from the legacy verification environment to the UVM VIP. The end user could use this kind of VIP as the normal UVM iUVC.

Unfortunately, we don't have this kind of existing Verilog BFM model in our case. This paper discusses a proposal for a customized UVM iUVC VIP which wraps Verilog BFM (which is directly decoupled from the proved RTL design). For a better hidden integration of BFM into UVM iUVC VIP, we use the abstract class concept, adapter, internal SV interfaces, and SV binding methodology. Let's start with our verification requirements and challenges, gradually introduce our proposed solution with example code, and finally discuss the consideration of the use model extending to an SoC.

II. VERIFICATION REQUIREMENTS AND CHALLENGES

As we see in the Figure 1, the master IP is connected to different Slave IPs through a shared bus where timing is particular and a little bit complex. The bus functional behavior of this master IP has been proved by Slave VIP in past projects at the SoC level and other slave legacy IPs could work well with this master IP as well. The master IP and Slave IPs are developed and maintained by other groups.

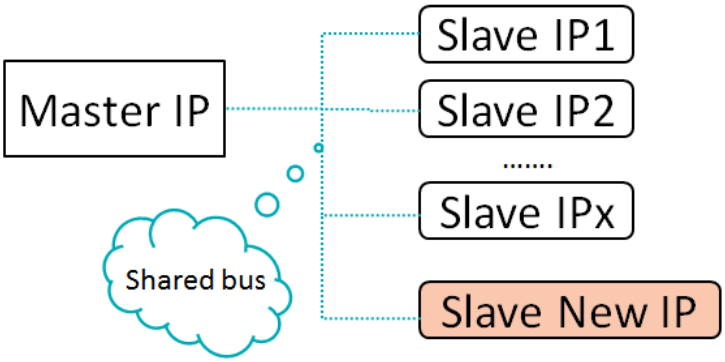

Fig. 1. Master & slave IP layout in an SoC integration

When our group develops the new slave IP with the same particular bus interface which will be integrated in the SoC, we encounter the following challenges in UVM stand-alone verification.

- There is no available UVM VIP including in-house and 3rd party.
- There is no Verilog BFM which has modular functions and tasks API.
- IP must be fully verified in UVM stand-alone, the SoC level verification is too late.
- If master IP changes the bus timing, the slave IP UVM verification environment must sync up without any changes.

To address the above challenges, as we can see in Fig. 2, the master IP designer could decouple the RTL into a clean Verilog BFM model with two bus interfaces: the one called the **lower bus** is there to communicate with the slave IP and the other called the **upper bus** is to get drive from high layers. The designer also makes the upper bus timing simple and easy to drive. The IP verification engineer could get the Verilog BFM (which always sync up with the master IP RTL design changes) to create the customized UVM VIP.

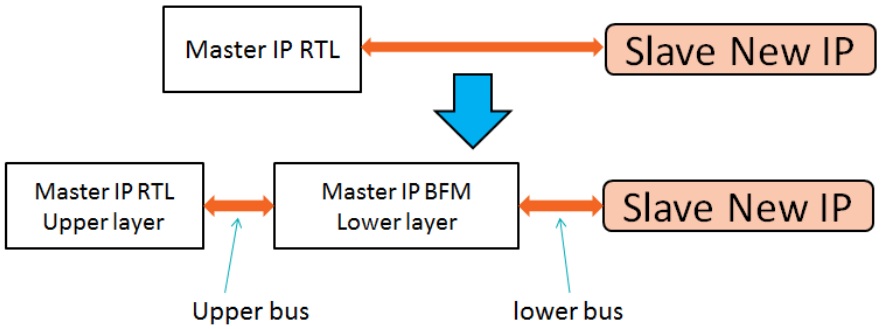

Fig. 2. Decouple the RTL design into Verilog BFM

In the simple way, we can create an interface UVC (iUVC) and adopt the SystemVerilog (SV) bind methodology to hookup the Verilog BFM to a slave new IP and hookup iUVC's virtual interface to the Verilog BFM model. Verification engineers can write several sequences and run them on the iUVC to drive

the upper bus to BFM, and then the BFM will communicate with the slave new IP as master IP behavior. The disadvantage of this method is it is not reusable at the higher level. At the SoC level, the BFM upper layer is driven by the Maser IP RTL upper layer design. We don't like to hack and bind the passive UVM iUVC to the master IP RTL design.

Fig. 3. BFM's location between UVM iUVC and DUV

III. PROPOSED SOLUTION

Fig. 4 depicts our general IP UVM stand-alone testbench architecture as below. It's a configurable layering structure, UVM interface UVCs (including the slave VIPs to mini the shared bus type) are bind to design under verification (DUV) and integrated into the module UVC (mUVC) component. There are other UVM components inside the mUVC to achieve the prediction, checking, and coverage staffs, such as module monitor, module prediction, module scoreboard, etc. So we expect to create a pure UVM iUVC VIP and directly communicate with DUT.

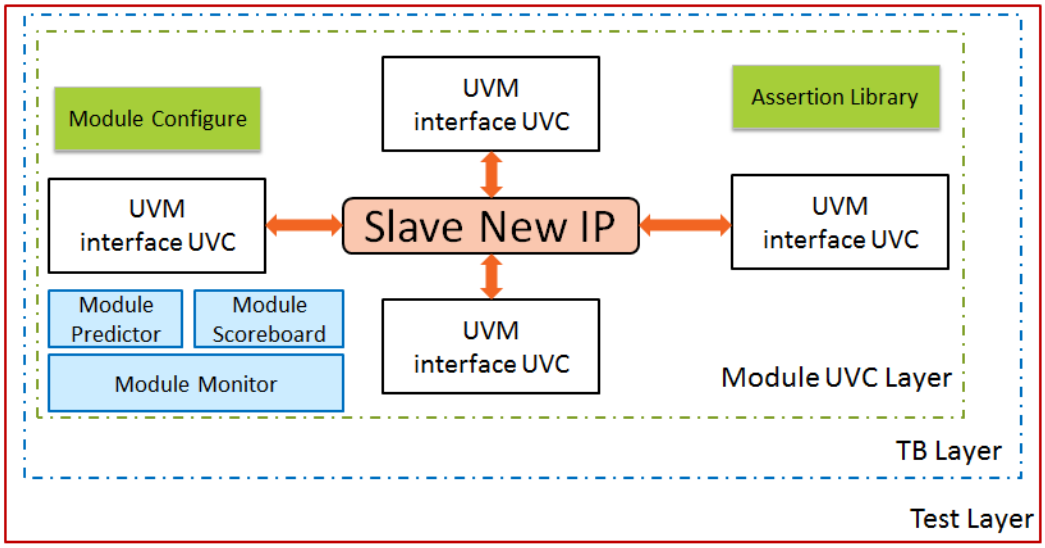

Fig. 4. Our general IP UVM stand-alone testbench architecture

In Fig. 5, we propose a customized UVM iUVC VIP which has following key features:

1. It reuses and wraps the Verilog BFM model which is invisible for users.
2. We create one internal upper bus interface and one VIP interface (which is for the user to hookup to DUV). Actually, the internal lower interface is a copy of the VIP interface. The two internal interfaces are bind to Verilog BFM and invisible for users. These interfaces are parameterized and scalable allowing requirements to change.
3. The abstract classes are created in SV interfaces. They assist to create the parameterized UVM component instances and provide APIs such as the tasks/functions/events defined in legacy BFM.
4. The UVM driver is a UVM component and drives the sequence item to Verilog BFM through the upper bus interface.
5. The adapter is a UVM component, and it acts like a bridge and passes down the bus data between the VIP interface from DUV and the lower bus interface to the Verilog BFM model at UVM run_phase. The implementation can ensure no re-timing or pipelining. We also support few types of error injection in the adapter.
6. The agent monitor spies the upper bus interface and sends back the bus data to sequence by uvm_event if necessary, e.g. sequence needs to get the return data to calculate the CRC to send the next sequence item.

7. The iUVC's monitor spies the VIP bus interface (Lower bus) and broadcasts the transaction item to other subscribers, e.g. predictor or scoreboard. There are few transaction level coverage inside.
8. There is a bus protocol assertion library and written in assertion interface which will hookup to DUV by SV binding methodology and verify the timing of VIP interface.
9. It has the built-in sequence library to verify the basic handshake between VIP and DUV.
10. The built-in functional coverage will cover the VIP's protocol and be vertical usable.
11. Only the UVM driver and adapter are parameterized, the other UVM components are non-parameterized. That will be easy for VIP integration.
12. To avoid Verilog BFM compile conflict with the same module name when reused to SoC level, we define the macro to exclude it from SoC compile.
13. It provides built-in debug logging verbosity control facilities.

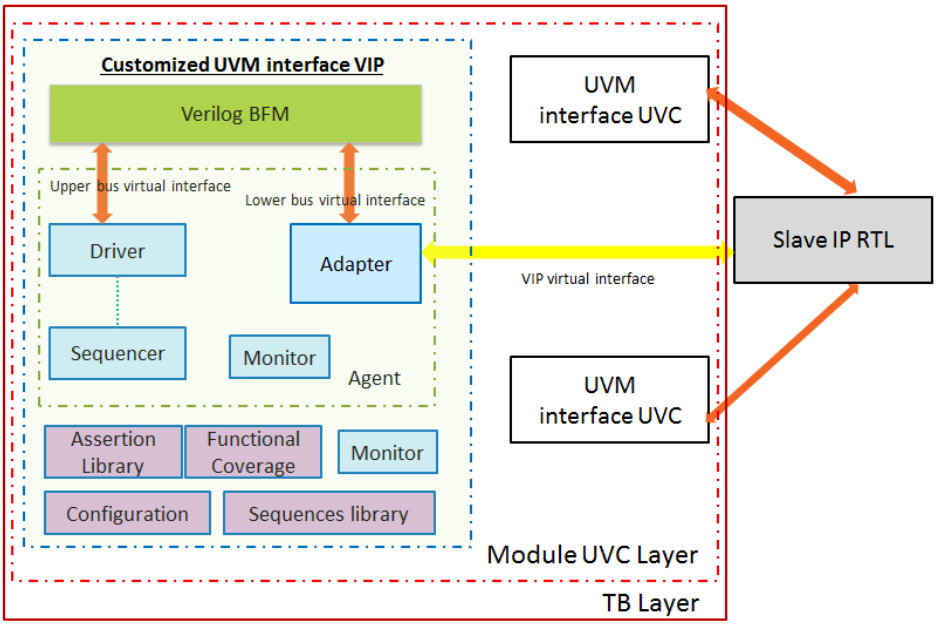

Fig. 5. Our proposed customized UVM VIP and integration

```
virtual class slave_vip_driver_api_base extends uvm_object;
  function new (string name = "salve_vip_drv_api_base");
     super.new(name);
  endfunction
  pure virtual function slave_vip_driver_base create_slave_driver (string m_name, uvm_component m_parent);
  .....
endclass
`define SLAVE_VIP_UPPER_IF_PARAM_DECL \
  LDATA1_WIDTH=`VIP_DEF_WIDTH1, \
  LDATA2_WIDTH=`VIP_DEF_WIDTH2, \
  ...
`define SLAVE_VIP_UPPER_IF_PARAM_MAP .LDATA1_WIDTH(LDATA1_WIDTH),.LDATA2_WIDTH(LDATA2_WIDTH),...
interface m_slave_upper_bus_if #(`SLAVE_VIP_UPPER_IF_PARAM_DECL) (
  input clk,
  input rst_n,
  inout [LDATA1_WIDTH-1:0] data_SLV2BFM;
  inout [LDATA2_WIDTH-1:0] data_BFM2SLV;
  .....
  );
  .....
  import uvm_pkg::*;
  `include "uvm_macros.svh"
  import slave_vip_uvc_pkg::*;
  class vip_driver_api extends slave_vip_driver_api_base;
    function new(string name = "vip_driver_api");
      super.name(name);
      uvm_config_db#(slave_vip_driver_api_base)::set(uvm_top, name, "vip_driver_api", this);
    endfunction
    function slave_vip_driver_base create_slave_driver (string m_name, uvm_component m_parent);
      slv_drv = vip_slave_driver#(`SLAVE_VIP_UPPER_IF_PARAM_MAP)::type_id::create(m_name,m_parent)
    endfunction
  ...
  endclass
  vip_driver_api driver_api = new($psprintf("%m"));
endinterface
```

Fig. 6. Parameterized interface and abstract classes for vip_driver

In Fig. 6, we create the parameterized upper bus interface and the abstract class 'slave_vip_driver_api_base'. In the m_slave_upper_bus interface, we implement the pure virtual functions in the vip_driver_api class which extends the base abstract class. In the vip_driver_api class's new function, we set the handle of this class instance to uvm_configure database. When we create the class instance by calling new function at the bottom, it will pass down the instance hierarchy as the name by "%m" and the hierarchy information could be used in uvm_config_db API.

In Fig. 7, the concrete slave_vip_adapter class implements similarly as the vip_driver_api class does. In addition, it defines several pure virtual functions (e.g. get/set_data1) and tasks (Ex. toggle_data1). They provide the abstract APIs to communicate the interface.

```
39 virtual class slave_vip_adapter_api_base extends uvm_object;
40   function new (string name = "salve_vip_adptr_api_base");
41      super.new(name);
42   endfunction
43   pure virtual function slave_vip_adapter_base create_slave_adaper (string m_name, uvm_component m_parent);
44   pure virtual function bit [LDATA1_WIDTH-1:0] get_data1();
45   pure virtual function void set_data1( bit [LDATA1_WIDTH-1:0] dat);
46   pure virtual task toggle_data1();
47   .....
48 endfunction
49
50 `define SLAVE_VIP_LOWER_IF_PARAM_DECL \
51   LDATA1_WIDTH=`VIP_DEF_WIDTH1, \
52   LDATA2_WIDTH=`VIP_DEF_WIDTH2, \
53   ...
54 `define SLAVE_VIP_LOWER_IF_PARAM_MAP .LDATA1_WIDTH(LDATA1_WIDTH),.LDATA2_WIDTH(LDATA2_WIDTH),...
55 interface m_slave_adpter_bus_if #(`SLAVE_VIP_LOWER_IF_PARAM_DECL) (
56   input clk,
57   input rst_n,
58   inout [LDATA1_WIDTH-1:0] data_SLV2ADPT;
59   inout [LDATA2_WIDTH-1:0] data_ADPT2SLV;
60   .....
61   );
62   .....
63   import uvm_pkg::*;
64   `include "uvm_macros.svh"
65   import slave_vip_uvc_pkg::*;
66   class vip_adpater_api extends slave_vip_adapter_api_base;
67     function new(string name = "vip_adapter_api");
68       super.name(name);
69       uvm_config_db#(slave_vip_adapter_api_base)::set(uvm_top, name, "vip_adapter_api", this);
70     endfunction
71     function  slave_vip_adapter_base create_slave_adaper (string m_name, uvm_component m_parent);
72       slv_adpter = vip_slave_adpater#(`SLAVE_VIP_LOWER_IF_PARAM_MAP)::type_id::create(m_name,m_parent)
73     endfunction
74
75     function bit [LDATA1_WIDTH-1:0] get_data1();
76       return data_SLV2ADPT;
77     endfunction
78     function void set_data1( bit [LDATA1_WIDTH-1:0] dat);
79       data_SLV2ADPT = dat;
80     endfunction
81     task toggle_data1();
82      @(data_SLV2ADPT);
83     endtask
84     .....
85   endclass
86   vip_adapter_api adapter_api = new($psprintf("%m"));
87 endinterface
```

Fig. 7. Parameterized interface and abstract classes for vip_adapter

In Fig. 8, we implement the parameterized slave VIP interface for end users, it's similar to the slave_adapter_bus interface. The user could hookup the slave VIP interface instance to the DUV module using SV bind methodology and set the handle of the interface instance to the uvm_configure database so as to get it in the hierarchical UVM components.

```
`define SLAVE_VIP_LOWER_IF_PARAM_LIST LDATA1_WIDTH,LDATA2_WIDTH,...

interface m_slave_VIP_bus_if #(`SLAVE_VIP_LOWER_IF_PARAM_DECL) (
  input clk,
  input rst_n,
  inout [LDATA1_WIDTH-1:0] data_SLV2DUV;
  inout [LDATA2_WIDTH-1:0] data_DUV2SLV;
  ...
endinterface

bind `SLAVE_DUV_MODNAME m_slave_VIP_bus_if#(SLAVE_VIP_LOWER_IF_PARAM_LIST) if_slv_vip_inst( ....);
uvm_config_db #(virtual m_slave_VIP_bus_if#(SLAVE_VIP_LOWER_IF_PARAM_LIST))::set(null,"uvm_test_top*","vip_vif" , if_slv_vip_inst);
```

Fig. 8. Parameterized interface and the binding for slave_VIP

In Fig. 9, we use the uvm_config_db API to get the adapter_api and driver_api handle in the slave VIP agent and call the create_slave_adapter/driver API to get an instance of parameterized adapter/driver. The slave_vip_agt is not parameterized, so it removes the integration overhead from uvm_agent to uvm_env caused by the parameterized.

```
class slave_vip_agt extends uvm_agent;
   vip_driver_api     driver_api;
   vip_adpater_api    adapter_api;

   slave_vip_adapter_api_base  adapter;
   slave_vip_driver_base       driver;

   function void build_phase(uvm_phase phase);
     ...
     if(!uvm_config_db#(slave_vip_adapter_api_base)::get(uvm_top, cfg.lower_vif_path,"vip_adapter_api",adapter_api))
       `uvm_fatal(get_type_name(),"Could not get adapter_api")
     if(!uvm_config_db#(slave_vip_driver_api_base)::get(uvm_top, cfg.upper_vif_path,"vip_driver_api",driver_api))
       `uvm_fatal(get_type_name(),"Could not get driver_api")
     adapter = adapter_api.create_slave_adaper("adapter",this);
     driver  = driver_api.create_slave_driver("driver",this);
    ...
endclass
```

Fig. 9. Use concrete classes APIs to create the parameterized UVM component

Fig. 10 depicts the flow to bind interfaces and use the uvm_config_db facility to pass down the virtual interfaces through Verilog BFM/DUV and UVM components.

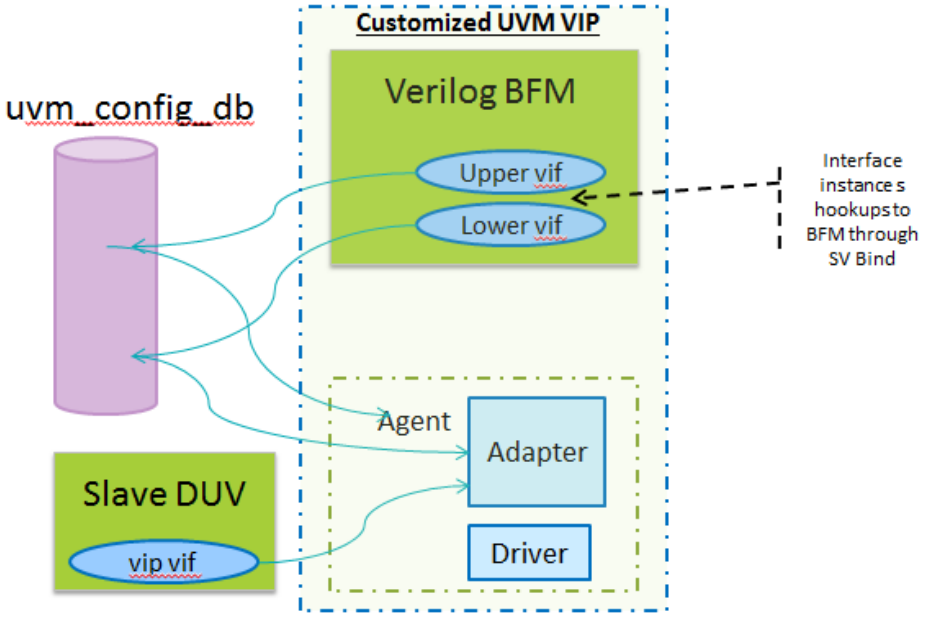

Fig. 10. Interface binding and connection between the BFM/RTL and UVM component

In Fig. 11, the slave_vip_adpr implements similarly with slave_vip_agt. In the run_phase, we use the adapter APIs to do the communication between BFM and DUV. For example, in the BFM2DUV flow, it calls the adapter_api.toggle_data1 to get a BFM.data1 data changing, and then passes the BFM.data1 to the

slave VIP virtual interface by calling adapter_api.get_data1(). There are several parallel threads to ensure a good handshake between BFM and DUV.

```
class slave_vip_adpr extends uvm_component;
   vip_adpater_api    adapter_api;
   virtual m_slave_VIP_bus_if#(SLAVE_VIP_LOWER_IF_PARAM_MAP)  vif;
   ...
   function void build_phase(uvm_phase phase);
     ...
     if(!uvm_config_db#(slave_vip_adapter_api_base)::get(uvm_top, cfg.low_vif_path,"vip_adapter_api",adapter_api))
       `uvm_fatal(get_type_name(),"Could not get adapter_api")
     if(!uvm_config_db#(virtual m_slave_VIP_bus_if#(SLAVE_VIP_LOWER_IF_PARAM_LIST))::get(this, "","vip_vif",vif))
       `uvm_fatal(get_type_name(),"Could not get vip_vif")
   endfunction
   virtual task run_phase (uvm_phase phase);
     ....
     fork
        begin // BFM2DUV
          adapter_api.toggle_data1();
          vif.data_SLV2DUV <= adapter_api.get_data1();
        end
        begin // DUV2BFM
          @vif.data_DUV2SLV;
          adapter_api.set_data1(data_DUV2SLV);
        end
        .....
     join
   endtask
endclass
```

Fig. 11. Use concrete classes APIs to bridge the BFM and DUV

IV. EXTENDING TO AN SOC

In the IP level, we created the customized UVM VIP to fully verify the slave new IP. When we deliver the slave IP to the SoC level for chip level verification, it's expected to reuse the UVM stand-alone as much as possible. In Fig. 12, at the early stage of SoC verification, the integration and verification to the master IP may be ready, however the slave IP integration is out there. We really want to verify the slave IP integration and basic function as early as possible. Based on this requirement, the master IP design could deliver the dummy stub and make the output signal as weak driving strength to avoid the multi-driven case. We reuse the IP level UVM sequences and customized UVM VIP on the SoC level, replace the master IP RTL with dummy stub in the Verilog file list, and execute the initial sequence on the VIP as active mode to the drive slave. When the master IP initial verification is available for basic integration and functions, it could do the handshake with slave IPs. We will reuse the existing customized UVM VIP integration at an early stage, and configure it in the passive mode. In this way, the VIP could help collect the functional coverage, checking the bus protocol timing by assertion.

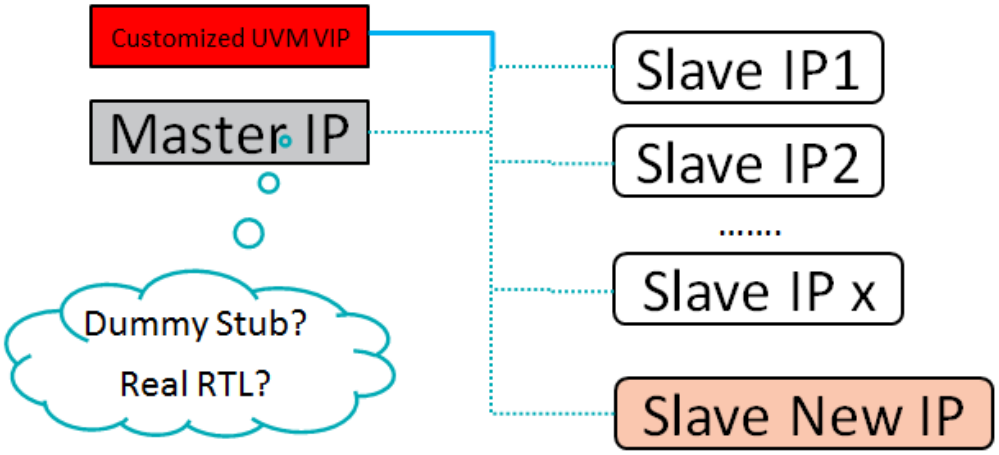

Fig. 12. Reuse customized UVM VIP from IP to SoC

V. CONCLUSION

With our verification requirements, we use the above approach to create a customized UVM VIP which reuses the decoupled Verilog BFM model, abstract classes and SV bind methodology. When the RTL designer decouples the RTL design into Verilog BFM, the verification engineer could build the VIP infrastructure and define the interface. The VIP can sync up with RTL design bus interface timing updates without any changes. By adopting the parameterized interface, the VIP can be more scalable. The base abstract class can be used by different concrete API classes in the customized VIP. Making use of the existing general iUVC template and reusing the Verilog BFM model, the creation effort can be significantly reduced from scratch.

Here is the effort compare.

- Effort to create the proposed customized UVC VIP.
  I. "Decouple the master RTL into upper/lower BFM". It depends on RTL designer.
  II. "Upper layer UVC". One person in 100% effort within 1 week.
  III. "Lower layer UVC adapter". One person in 100% effort within 2 days.
  IV. "Interface BFM tasks". One person in 100% effort within 2 days.
- Effort to create the Master UVC VIP.
  It's totally un-predictable because it depends on the complexity of bus timing. It will introduce upgrade issue when master RTL timing changes.

The proposal depicted in this paper can be extended to the other prototype such as, a PCI, I2C like interface using bi-directional interface or an AMBA like interface with ORed structure and separated channels. The key is to decouple the RTL design into a clean BFM with a simple upper bus interface. The verification engineer and RTL design engineer should consider the reusable solution for both design and verification. In general, it only needs to decouple the specific module, but not the whole design. The less it impacts the RTL design the better.

This proposal also has few limitations.

- Irritation/Error injection on lower layer interface.
  Even we could do few error injections in the adapter, but adding delayed responses, back pressure on receiving or inserting errors at lower layer interface are nearly impossible. BFM model will always drive interface in same way, based on its state machine and driver code.
- If the master RTL has minor changes, the possibility of bug missing should be low and the Verilog BFM deliver time should be short on the master IP verification point of view. However, if the master RTL has big changes, the customized UVC VIP mainly depends on verification quality and the time from master IP verification team.

To fully address the issues above, verification team should schedule the resource to create the proper lower layer UVC and build the protocol layering architecture with existing upper layer UVC.

ACKNOWLEDGMENT

We would like to thank my wife (Liangliang Li) and the AMD team (Davis.Wan and Leo.zhang) for their continued support. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] http://www.shoaibinamdar.in/blog/?p=393
[2] IEEE 1800-2009 SystemVerilog
[3] David Rich, Mentor, "The missing link: the testbench to DUT connection", DVCon USA 2012