PRACTICAL SCHEMES TO ENHANCE VERTICAL, HORIZONTAL AND PLATFORM REUSABILITY OF VERIFICATION COMPONENTS IN AMBA BASED SOC DESIGN

Ieryung Park, Nara Cho, and Yonghee Im

SK Hynix, SoC IP&DV, Bundang, Sungnam, Korea
({ieryung.park, nara2.cho, yonghee.im}@sk.com)

***Abstract*- This paper presents practical schemes to enhance the reusability of verification components for AMBA based SoC design verification. *AMBA Unified Verification System* (AUVS) can reduce the amount of time to build testbench by pre-defining AMBA agents per configuration and it can also improve *platform* reusability of verification components by providing an abstraction layer of AMBA VIPs. *Firmware-Like Sequence* (FLS) is a scheme to describe test sequences using only tasks provided by *Hardware Abstraction Layer* (HAL) which positions itself between test sequence and agent, thereby enhancing *vertical* and *horizontal* reusability of test sequences.**

I. INTRODUCTION

One research found out that 22% and 23% of the total verification time are spent on testbench build-up and test scenario coding respectively, while 36% on debugging time [1]. In other words, the total amount of time for preparing for tests – both testbench and test scenario – is relatively greater than actual verification work-hours: 45% vs. 36%. That is a good reason for why many efforts have been attempted to improve the reusability of verification components constituting testbench and testcases. Among such efforts are *SystemVerilog* and *Universal Verification Methodology* (UVM), which have boosted up such reusability [2] [3]. Although SystemVerilog and UVM can facilitate a design verification environment by providing basic verification components and methodologies, they do not guarantee the reusability of such verification components. Rather, the reusability can be improved by verification engineers only if they use or create such verification components with the concept of reusability in mind. The reusability can be categorized into three kinds: *Vertical*, *Horizontal*, and *Platform* reusability [1].

This paper presents practical schemes to improve all three kinds of reusability of verification components aiming at AMBA based SoC design that is general form of SoC. The practical scheme is composed of two major concepts: 1. *AMBA Unified Verification System* (AUVS) not only to improve verification components' platform reusability, but also to shorten testbench buildup time, 2. *Firmware-Like Sequence* (FLS) to improve test sequences' vertical and horizontal reusability by providing *Hardware Abstraction Layer* (HAL) between test sequence and agent.

II. REUSE PROBLEMS

This section summarizes three major reuse issues that verification engineers encounter and find time-consuming while at work, before introducing you to our suggested solutions.

Problem 1. Testbench build. At IP block level verification, we need nearly as many testbenches as the number of IP blocks. Each testbench build-up is considered as a series of routine jobs: agents creation and their configuration, interfaces creation and their configuration, verification components connection, sequences creation, and so on. If there were a simple and systematic way to do so, it would be a good benefit in reducing a great amount of time and effort put in building up testbenches.

Problem 2. Platform reusability of test sequences. We might need to use multiple vendors' AMBA VIP for many reasons, including VIP license issues, co-emulator synthesizability, simulator's compatibility, and even use of in-house VIPs. This situation enforces us to figure out how to effectively use different test sequences for each different VIP, so called *platform reusability* issue.

Problem 3. Vertical and horizontal reusability of test sequences. It is not unusual to see the cases wherein a sequence used at IP block level verification cannot be reused at higher level for good reasons, including the different surroundings among the levels. As a result, AMBA bus sequences used at IP block cannot be directly reused at the higher level, so called *vertical reusability* issue. Figure 1 shows an example of vertical reusability wherein AHB driver used at IP level cannot be reused at subsystem level due to AXI bus. Similarly, we run into the similar issue

when DUT is reused in another project – e.g., SFR access port is changed from AHB to AXI. That is called *horizontal reusability* issue.

In the following, section III and IV suggest separate solutions to reuse problem 1, 2 and 3 respectively. Section III introduces AUVS scheme, pre-built UVM based verification platform with wrapper agents for each bus protocol, which is devised to solve reuse problem 1 and 2 by reducing testbench build-up time and using wrapper agents independent of specific VIP. In section IV, FLS scheme, describing test sequence with only tasks of HAL, is proposed as a solution to problem 3 in that it makes test sequences free of SFR port protocols.

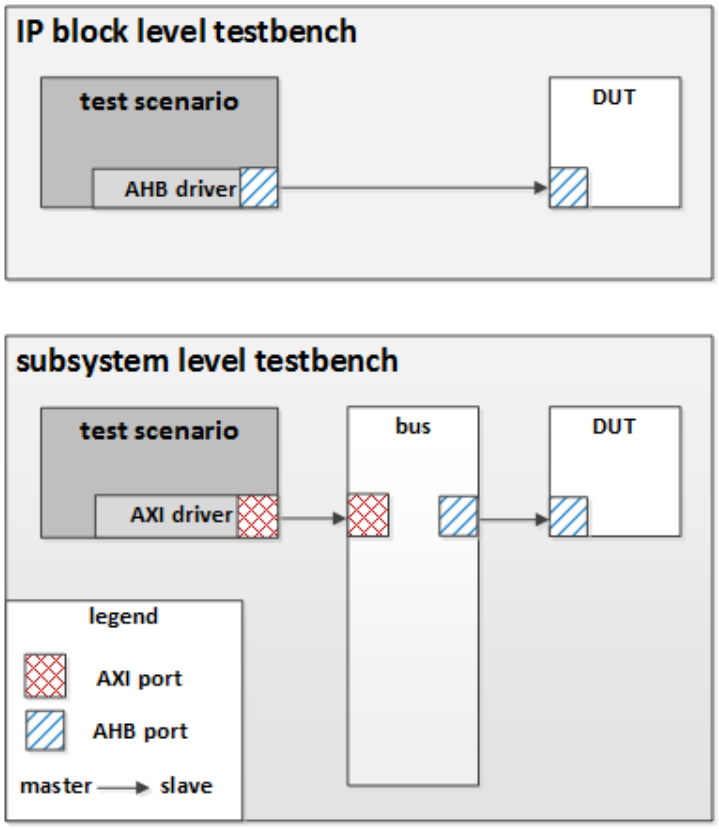

Figure 1. An example of vertical reusability

III. AMBA UNIFIED VERIFICATION SYSTEM (AUVS)

A. AUVS Overview

Figure 2 shows the overall architecture of *AMBA Unified Verification System* (AUVS), composed of *AUVS TPI sequence*, *AUVS environment*, and *AUVS harness*. AUVS can be considered a pre-built UVM based verification platform which provides users with a reusable AMBA transaction generation mechanism. Three major components are briefly introduced in this section, followed by more detail explanation along with code snippets in the later sections.

AUVS environment *creates*, *configures*, and *connects* AUVS agents. There are AUVS agents for each bus protocol – AXI, AHB, and APB – and each of them contains various vendors' AMBA VIP agents. Users can select a specific vendor's VIP, the number of its instances, and other protocol parameters for the AUVS agents. Once the AUVS agents are created and configured, they are connected to AUVS interfaces. AUVS harness generates an array of AUVS interfaces per user's configuration and registers them in UVM resource pool so that AUVS environment can fetch them whenever needed. As a result, all the AMBA agents – any kind and any number – can be available simply by instantiating this AUVS environment with proper user configuration, thereby reducing the testbench buildup time dramatically (*Problem 1*).

Besides, you don't need separate configuration classes to pass bus protocol parameters down to agents, since the AUVS interface contains the parameters inside. This approach makes a little more sense in that such parameters are relevant to each bus protocol. Although they are assigned default values, they can be overridden by user specific values from outside later if needed.

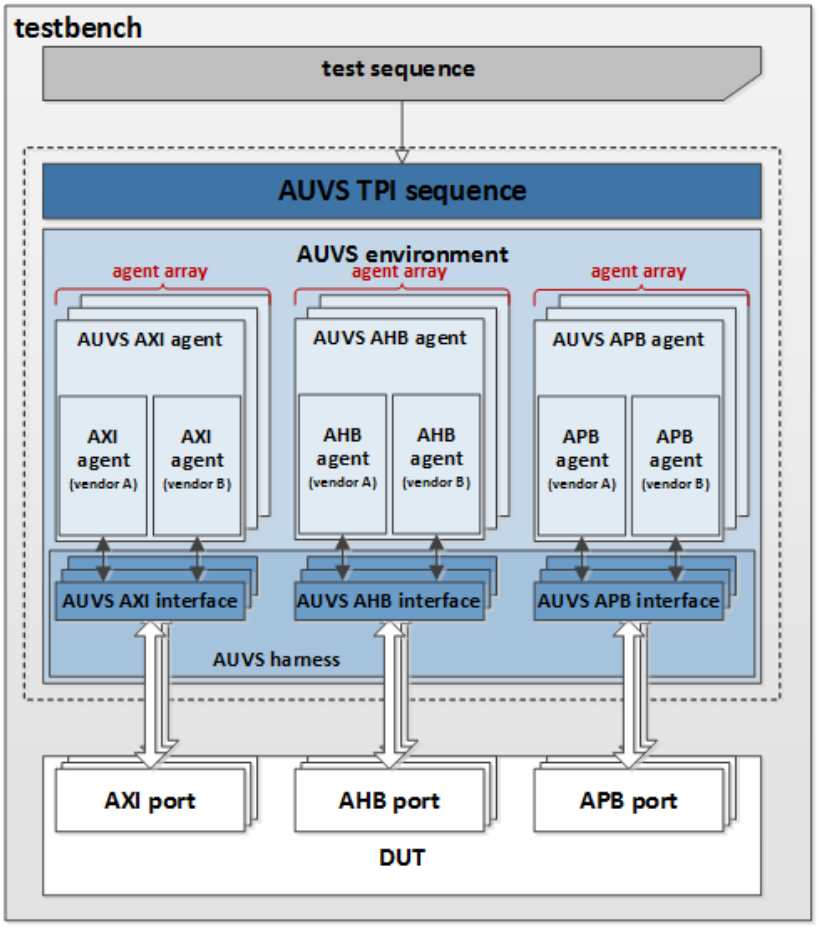

Figure 2. AUVS architecture

TPI stands for *Test Programming Interface*. TPI is a class inheriting `uvm_sequence` intended to provide all test sequences with *Application Programming Interface* (API) to access VIP vendor independent AMBA transactions. The goal is to reuse a test sequence inheriting TPI without any modification even when target AMBA VIP changes, thereby solving the so called platform reusability issue of test sequences (*Problem 2*).

Since AUVS environment is a pre-built platform, all you have to do is to use AUVS interface and AUVS TPI sequence properly. The following two sections will explain how they work from a user's point of view, starting from AUVS interface. In this paper's example codes, it is assumed that Synopsys AMBA SVT, Mentor AMBA QVIP, and in-house AMBA VIP are among target AMBA VIP candidates. It is not to waste unnecessary VIP licenses to choose which VIP for use at compile time.

B. AUVS Interface

There are a pair of AUVS interfaces per bus protocol: *master* and *slave*. Snippet 1 shows AXI master interface code as example. Various vendors' AXI VIP interfaces are pre built-in and connected. One of them will be chosen with `` `define `` at compile time. Bus protocol specific parameters such as `addr_width` and `data_width` are defined and assigned default value as mentioned previously. The parameters are referred to by AUVS environment in order to configure AUVS agents.

Snippet 1. auvs_axi_master_interface.sv

```
interface auvs_axi_master_interface(
    input   aclk ,
    input   areset_n ,
            axi_awid,
    // ...
);

// Use Synopsys SVT AXI
`ifdef AUVS_USE_SVT_AXI
    svt_axi_master_if inst(
        .common_aclk( aclk ) ,
        .aresetn( areset_n ) ,
        .awid( axi_awid ) ,
        // ...
    );
// Use Mentor QVIP or VTL AXI
`elsif AUVS_USE_QVIP_AXI
    mgc_axi inst(
        1'bz , 1'bz
    );
    assign axi_awid = inst.awid ;
    // ...
// Use in-house AXI
`else
    axi_vif inst(
        .axi_aclk( aclk ) ,
        .axi_areset_n( areset_n ) ,
        .axi_awid( axi_awid ) ,
        // ...
    );
`endif

    // Protocol parameters
    //  default value : address bit-width = 32, data bit-width = 32
    bit[7:0] addr_width = 32 ;
    bit[7:0] data_width = 32 ;

endinterface
```

C. AUVS Harness

Snippet 2 shows AUVS harness, a collection of all available AMBA interfaces. The interfaces are instantiated, assigned to virtual interface, and then registered into UVM resource pool so that testbench can fetch them through `uvm_config_db::get`. AUVS harness can be accessed using `` `include `` in testbench, because interface is not allowed to inherit in SystemVerilog. See Snippet 7 for this.

Snippet 2. auvs_harness.svi

```
interface auvs_harness ;

    // AUVS_AXI_MST_AGT_MAX is defined in compile option by user
    auvs_axi_master_interface u_auvs_axi_mst_if[0:(`AUVS_AXI_MST_AGT_MAX-1)] ;
    virtual interface
    auvs_axi_master_interface u_auvs_axi_mst_vif[0:(`AUVS_AXI_MST_AGT_MAX-1)] ;

    initial begin
        int i ;
        u_auvs_axi_mst_vif[0:(`AUVS_AXI_MST_AGT_MAX-1)] =
```

```
            u_auvs_axi_mst_if[0:(`AUVS_AXI_MST_AGT_MAX-1)] ;

        // All VIFs are registered into UVM resource pool by uvm_config_db::set
        for ( i=0 ; i<`AUVS_AXI_MST_AGT_MAX ; i++ ) begin
            uvm_config_db#(virtual interface auvs_axi_master_interface)::set(
                uvm_root::get(), "uvm_test_top",
                $sformatf("u_auvs_axi_mst_vif[%0d]",i), u_auvs_axi_mst_vif[i] ) ;
        end
    end

endinterface
```

D. TPI Sequence

Snippet 3 shows the header part of TPI sequence class. TPI sequence class provides a pair of write/read task of each bus protocol with which its inherited test sequences can generate bus traffic. The tasks communicate transaction input/output data through arguments. Argument `agt_id` selects a target agent among many available ones.

Snippet 4 picks up the AXI write task, `axi_write()` as example to explain how it works. The code shows the case in that Synopsys AXI VIP is chosen as target agent. As explained in the comments in Snippet 4, a vendor specific sequence item, or transaction, is generated, set with input arguments, configured with port timing values, and then issued. Such a vendor specific flow is pre-built in each TPI task.

Snippet 3. `auvs_tpi_sequence.sv` (header part)

```
class auvs_tpi_sequence extends uvm_sequence ;
    extern task axi_write(  int agt_id, bit[] axi_id, bit[] addr,
                            bit[] burst, bit[] size, bit[] data);
    extern task axi_read(   int agt_id, bit[] axi_id, bit[] addr,
                            bit[] burst, bit[] size, output bit[] data);
    extern task ahb_write(  int agt_id, bit[] addr,
                            bit[] burst, bit[] size, bit[] data);
    extern task ahb_read(   int agt_id, bit[] addr,
                            bit[] burst, bit[] size, output bit[] data);
    extern task apb_write(  int agt_id, bit[] addr, bit[] data);
    extern task apb_read(   int agt_id, bit[] addr, output bit[] data);
endclass
```

Snippet 4. `auvs_tpi_sequence::axi_write()` (body part)

```
task auvs_tpi_sequence::axi_write(int agt_id, bit[] axi_id, bit[] addr, bit[] burst, bit[]
size, bit[] data);

`ifdef AUVS_USE_SVT_AXI
    // generate & start SVT AXI seq_item
    svt_axi_master_transaction axi_xaction ;
    svt_configuration cfg ;

    // a. Create
    `uvm_create_on( axi_xaction , auvs_seqr.axi_seqr[agt_id].target_seqr ) ;
    start_item( axi_xaction );

    // b. Set sequence item
    axi_xaction.xact_type = svt_axi_transaction::WRITE ;
    axi_xaction.addr = addr ;
    axi_xaction.burst_type = svt_axi_transaction::burst_type_enum`(burst) ;
    axi_xaction.burst_size = svt_axi_transaction::burst_size_enum`(size) ;
    axi_xaction.data = data ;
```

```
    // c. Set delay parameters
    axi_xaction.awvalid_delay = this.awvalid_delay[port_id] ;
    axi_xaction.wvalid_delay = this.wvalid_delay[port_id] ;
    axi_xaction.bready_delay = this.bready_delay[port_id] ;
    //...

    // d. Issue traffic
    finish_item( axi_xaction ) ;

`elsif AUVS_USE_QVIP_AXI
    // generate & start QVIP AXI seq_item
    // ...

`else
    // generate & start in-house AXI seq_item
    // ...
`endif

endtask
```

E. AUVS Environment

In fact, it is enough to refer to AUVS interface and TPI sequence – explained up to now – for users in order to fully utilize AUVS. The following two sections deep-dive into AUVS environment, the underneath engine to make all this possible, so that one can understand how AUVS interface and TPI sequence are handled internally.

As shown in Snippet 5, `auvs_env` has agents (`auvs_axi_master_agent`), configs (`auvs_axi_config`), virtual interfaces (`auvs_axi_master_interface`), and virtual sequencer (`auvs_sequencer`) as member. During `build` phase, all kinds of AMBA protocols' master and slave agents are created along with their configs, followed by a routine of virtual interface retrieval from UVM resource DB, agents/configs configuration, and registration of AUVS virtual sequencer (`auvs_sequencer`). Snippet 5 only shows AXI master case as example but other bus protocols should look similar. The sequencer of each agent is connected to AUVS virtual sequencer (`auvs_sequencer`) during `connect` phase, so that TPI sequence can use it to put sequence items into specific agents later. Next comes AUVS agent which is the most important component in AUVS environment.

Snippet 5. `auvs_env.sv`

```
class auvs_env extends uvm_env ;

    auvs_axi_master_agent      u_auvs_axi_mst_agt[`AUVS_AXI_MST_AGT_MAX] ;
    auvs_axi_config            u_auvs_axi_mst_cfg[`AUVS_AXI_MST_AGT_MAX] ;
    virtual interface
    auvs_axi_master_interface  u_auvs_axi_mst_vif[`AUVS_AXI_MST_AGT_MAX] ;
    auvs_sequencer             vseqr ;

    function void build_phase( uvm_phase phase ) ;
        int i ;
        super.build_phase( phase ) ;

        for ( i = 0 ; i < `AUVS_AXI_MST_AGT_MAX ; i++ ) begin
            u_auvs_axi_mst_agt[i] = auvs_axi_master_agent::type_id::create(
                                    $sformatf("u_auvs_axi_mst_agt[%0d]",i), this);
            u_auvs_axi_mst_cfg[i] = auvs_axi_config::type_id::create(
                                    $sformatf("u_auvs_axi_mst_cfg[%0d]",i), this);

            uvm_config_db#(virtual interface auvs_axi_master_interface)::get(
                uvm_root::get(),
                "uvm_test_top",
```

```
                $sformatf("u_auvs_axi_mst_vif[%0d]",i),
                u_auvs_axi_mst_vif[i]);

            u_auvs_axi_mst_cfg[i].addr_width=u_auvs_axi_mst_vif.addr_width ;
            u_auvs_axi_mst_cfg[i].data_width=u_auvs_axi_mst_vif.data_width ;
            u_auvs_axi_mst_agt[i].vif = u_auvs_axi_mst_vif[i] ;
            u_auvs_axi_mst_agt[i].set_config( u_auvs_axi_mst_cfg[i] ) ;
        end

        // create & register virtual sequencer
        vseqr = auvs_sequencer::type_id::create("vseqr",this);
        uvm_config_db#(auvs_sequencer)::set(
            uvm_root::get(),
            "uvm_test_top",
            "auvs_seqr",
            vseqr);
        // ...

    endfunction

    function void connect_phase( uvm_phase phase ) ;
        int i ;
        for ( i=0 ; i < `AUVS_AXI_MST_AGT_MAX ; i++ ) begin
            vseqr.axi_seqr[i] = u_auvs_axi_mst_agt[i].seqr ;
        end
        // …
    endfunction

endclass
```

F. AUVS Agents

Snippet 6 shows AUVS AXI master agent as example, among other AUVS agents. This AUVS agent plays as a placeholder to contain a user selected agent and its sequencer. The target agent is created and then assigned config instance passed down by AUVS environment during `build` phase. During `connect` phase, the target agent's sequencer is connected to AUVS agent's sequencer (instance of `auvs_axi_sequencer`), which, in turn, is connected to the virtual sequencer of AUVS environment (instance of `auvs_sequencer`) as seen in Snippet 5. TPI sequence can utilize the target agent's sequencer in this mechanism which connects target agents, AUVS agents, AUVS environment, and TPI all together in terms of sequence/sequencer.

Snippet 6. `auvs_axi_master_agent.sv`

```
class auvs_axi_master_agent extends uvm_agent ;
    auvs_axi_sequencer seqr ;
    auvs_axi_config cfg ;

`ifdef AUVS_USE_SVT_AXI
    svt_axi_master_agent inst ;
    svt_axi_config svt_cfg ;
`elsif AUVS_USE_QVIP_AXI
    // ...
`else
    // ...
`endif

    function void set_config( auvs_axi_config in_cfg ) ;
        cfg = in_cfg ;
    endfunction
```

```
    function void build_phase ( uvm_phase phase ) ;
    `ifdef AUVS_USE_SVT_AXI
        inst = svt_axi_master_agent::type_id::create("inst");
        svt_cfg = svt_axi_config::type_id::create("svt_cfg");
        svt_cfg.address_width = cfg.addr_width ;
        svt_cfg.data_width = cfg.data_width ;
        inst.cfg = svt_cfg ;
    `elsif AUVS_USE_QVIP_AXI
        // ...
    `else
        // ...
    `endif
    endfunction

    function void connect_phase ( uvm_phase phase ) ;
    `ifdef AUVS_USE_SVT_AXI
        seqr.target_seqr = inst.sequencer ;
    `elsif AUVS_USE_QVIP_AXI
        seqr.target_seqr = inst.seqr ;
    `else
        seqr.target_seqr = inst.sequencer ;
    `endif
    endfunction

endclass
```

G. Testbench Implementation

Up to this point, we've seen how AUVS is constructed and how it works, from TPI down to AUVS Interface. Now it is time to show how to use them in a practical way. Please recall there are only two areas to focus on from a user's point of view: *AUVS interface* and *TPI sequence*.

Snippet 7 presents the example testbench module – `tb_top` to deal with AUVS interfaces. The module includes "`auvs_harness.svi`" file for reusing AUVS harness, and connects ports of DUT to AMBA agents' interfaces of AUVS harness. Connection between DUT and AUVS environment is finalized by assigning AMBA agent configs' parameters to corresponding interfaces.

Snippet 7. `tb_top.sv`

```
module tb_top ;
    // a. include 'auvs_harness'
    `include "auvs_harness.svi"

    // b. Connect interface signals
    DUT u_dut(
        .awvalid( u_auvs_axi_mst_if[0].awvalid ) ,
        .awready( u_auvs_axi_mst_if[0].awready) ,
        .awaddr( u_auvs_axi_mst_if[0].awaddr ) ,
        // ...
    ) ;

    // c. Overwrite protocol parameters
    initial begin
        u_auvs_axi_mst_if[0].addr_width = 32 ;
        u_auvs_axi_mst_if[0].data_width = 64 ;
    end

endmodule
```

Snippet 8 shows the example code of AUVS test sequence handling TPI sequence. This sequence inherits `auvs_tpi_sequence` and describes a test scenario by using TPI sequence provided API tasks. Note that, in `pre_body()`, `auvs_sequencer` registered in UVM resource pool is assigned to `auvs_seqr`, a member of `auvs_tpi_sequence`. This mechanism makes it possible to continue to reuse `auvs_sample_sequence` based on API tasks, even though underneath AMBA VIP agents change.

Snippet 8. `auvs_sample_sequence.sv`

```
class auvs_sample_sequence extends auvs_tpi_sequence ;
    `uvm_object_utils( auvs_sample_sequence ) ;

    task pre_body();
        // Get auvs_seqr from UVM resource pool and set it as virtual sequencer
        //  to put transactions
        if ( !uvm_config_db(auvs_sequencer)::get(uvm_root::get(),"uvm_test_top",
            "auvs_seqr",auvs_seqr) ) begin
            `uvm_fatal("auvs_sample_sequence","Get auvs_seqr fail");
        end
    endtask

    task body();
        int agt_id , axi_id , len ;
        axi_write( agt_id=0, axi_id=0, 'h4000_0000, AUVS_AXI_BYTE_8, len=15, wdata,wstrb ) ;
        axi_read(  agt_id=0, axi_id=1 , 'h4000_0000 , AUVS_AXI_BYTE_8 , len=15 , rdata ) ;
        ahb_write( agt_id=0, 'h5000_0000 , AUVS_AHB_INCR16 , AUVS_AHB_BYTE_4 , wdata ) ;
        ahb_read(  agt_id=0, 'h5000_0000 , AUVS_AHB_INCR16 , AUVS_AHB_BYTE_4 , rdata ) ;
    endtask
endclass
```

IV. Firmware-Like Sequences(FLS) with Hardware Abstraction Layer(HAL) Scheme

A. FLS Overview

Firmware-Like Sequences (FLS) is a scheme which enables to create test sequences through *Hardware Abstraction Layer* (HAL). HAL is a verification component which provides API tasks abstracting DUT's features. Such API tasks are written solely with driver tasks – `write_driver()` and `read_driver()` – to access DUT's SFR. The driver tasks use only address and data just like memory mapped block access typically used in firmware programming. In turn, `write_driver()` and `read_driver()` call virtual tasks, `bus_write()` and `bus_read()`, respectively, which allow specific – or VIP dependent – implementation later. Since bus operations of HAL are decided through *callback*, HAL based test sequences can be reused even if SFR port protocols change, achieving vertical and horizontal reuse (*Problem 3*). Figure 3 shows the overall architecture of FLS. You can notice that the upper section w.r.t. HAL is protocol independent, while the lower is protocol dependent. Let us explain how FLS major components behave in the following sections.

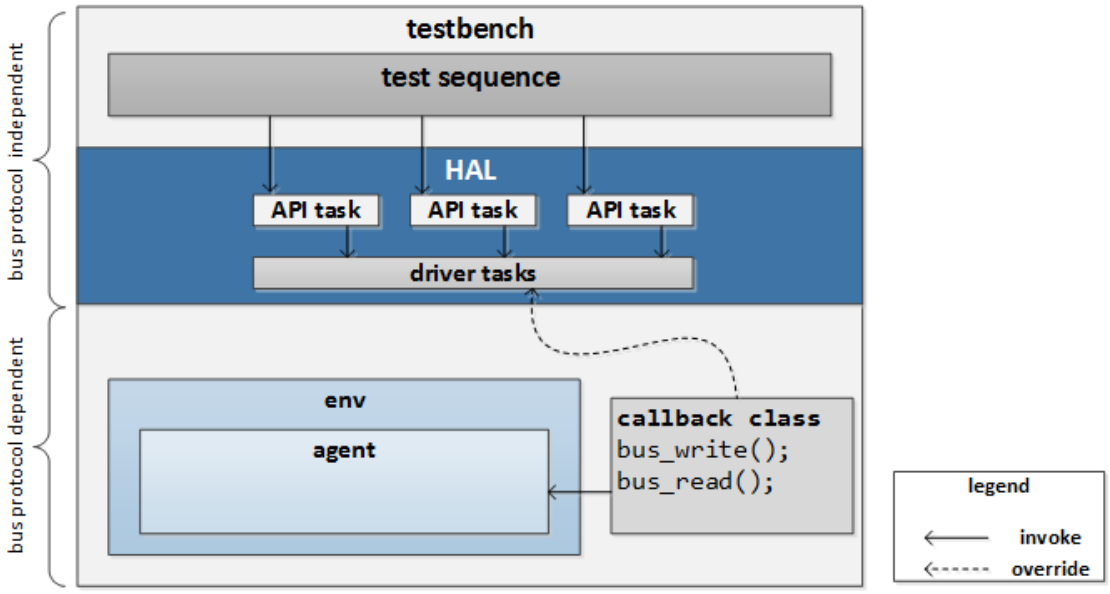

Figure 3. FLS architecture

B. HAL Base Class and HAL Callback Base Class

Snippet 9 shows `hal_base`, HAL base class, which is super class to all HAL classes. Base HAL callback class, `hal_callback_base`, is registered as callback class for `hal_base`. `hal_base` contains two driver tasks: `write_driver()` and `read_driver()`. Each of them calls a virtual task of `hal_callback_base: bus_write()` or `bus_read()`.

Snippet 10 shows the code of `hal_callback_base` class. A task declared `virtual` is expected to be implemented with a specific AMBA protocol operation.

Snippet 9. `hal_base.sv`

```
class hal_base extends uvm_object ;
    `uvm_object_utils( hal_base ) ;

    // Register hal_callback_base as callback class
    `uvm_register_cb( hal_base , hal_callback_base ) ;

    // Driver task : write_driver
    task write_driver( bit[(HAL_ADDR_WIDTH-1):0] addr , bit[(HAL_DATA_WIDTH-1):0] data ) ;
        // Use `uvm_do_callback to invoke callback task
        `uvm_do_callback( hal_base , hal_callback_base , bus_write(addr,data) );
    endtask

    // Driver task : read_driver
    task read_driver( bit[(HAL_ADDR_WIDTH-1):0] addr,
              output bit[(HAL_DATA_WIDTH-1):0] data ) ;
        // Use `uvm_do_callback to invoke callback task
        `uvm_do_callback( hal_base , hal_callback_base , bus_read(addr,data) );
    endtask

endclass
```

Snippet 10. `hal_callback_base.sv`

```
class hal_callback_base extends uvm_callback ;
    // virtual task. Real operation isn't implemented here.
```

```
    virtual task bus_write(bit[] addr, bit[] data);
    endtask

    virtual task bus_read(bit[] addr, output bit[] data);
    endtask
endclass
```

C. Testbench Implementation

In this section, let us show how to use HAL base classes in creating VIP independent sequences. DDR SDRAM host controller (a.k.a. DDR controller) is picked up for this purpose. Snippet 11 shows a DDR controller's HAL class, `ddrc_hal`, by inheriting the HAL base class, or `hal_base`. `ddrc_hal` provides API tasks which abstract the DDR controller's basic features. Note that those API tasks should be written only with `hal_base`'s tasks: `write_driver()` and `read_driver()` in this example so that `ddrc_hal` can be independent of any specific bus protocols.

Specific protocol for use can be defined in DDR controller's HAL callback class as seen in Snippet 12. In this example, AHB protocol is used to write to or read from SFR. You can notice that virtual tasks declared in `hal_callback_base` (Snippet 10) are finally implemented in `ddrc_hal_callback` class. Of course, you can change the tasks – e.g., from AHB to AXI – later in a different level testbench (*vertical reuse*) or in a different project (*horizontal reuse*).

Snippet 13 is the code that replaces `ddrc_hal_callback`'s bus tasks if testbench becomes to use AXI bus protocol for SFR access. The same thing applies to the case of using *RAL* (*Register Abstraction Layer*). It is good enough for you to use RAL operations in those bus tasks in that case.

Once DDR controller's HAL and HAL callback classes are ready, its sequences can be written by using HAL API tasks as seen in Snippet 14.

Snippet 11. `ddrc_hal.sv`

```
class ddrc_hal extends hal_base ;
    // DDR controller API tasks
    //  API tasks do not use sequence items or sequences to access SFR of DUT.
    task mr_write() ;
        write_driver( addr , data ) ;
    endtask

    task mr_read() ;
        write_driver( addr , data ) ;
    endtask

    task set_address() ;
        write_driver( addr , data ) ;
    endtask

    task set_phy_delay();
        write_driver( addr , data ) ;
    endtask

    task set_ddrc_start();
        // Read-Modify-Write DDRC_CTRL[0] to Start DDRC
        read_driver( `DDRC_CTRL, rdata ) ;
        rdata[0] = 1 ;
        write_driver( `DDRC_CTRL, rdata ) ;
    endtask

    task init() ;
        mr_write(0x2,`MR2_DATA);
        mr_write(0x3,`MR3_DATA);
```

```
        mr_write(0xD,`MR13_DATA);
        mr_write(0x1,`MR1_DATA);
        set_address(CS=1,BANK=3,ROW=15,COL=10);
        set_phy_delay(`PHY_DELAY_VALUE);
        set_ddrc_start();
    endtask

endclass
```

Snippet 12. ddrc_hal_callback.sv

```
class ddrc_hal_callback#(type T = auvs_tpi_sequence) extends hal_callback_base ;
    T seq ;

    function void set_seq( T in_seq ) ;
        seq = in_seq ;
    endfunction

    // Override bus_write & bus_read
    // Bus operations are implemented with AUVS.
    task bus_write(bit[] addr, bit[] data);
        int agt_id ;
        seq.ahb_write( agt_id = 0, addr, AUVS_AHB_BURST_SINGLE, AUVS_AHB_SIZE_BYTE_4,data) ;
    endtask

    task bus_read(bit[] addr, output bit[] data);
        int agt_id ;
        seq.ahb_read( agt_id = 0, addr, AUVS_AHB_BURST_SINGLE, AUVS_AHB_SIZE_BYTE_4, data) ;
    endtask

endclass
```

Snippet 13. AXI bus operation tasks

```
// Override bus_write & bus_read
task bus_write(bit[] addr, bit[] data);
    int agt_id , axi_id , len ;
    seq.axi_write( agt_id = 0, axi_id, addr, AUVS_AXI_BYTE_4, len=0, data ) ;
endtask

task bus_read(bit[] addr, output bit[] data);
    int agt_id , axi_id , len ;
    seq.axi_read( agt_id = 0, axi_id, addr, AUVS_AXI_BYTE_4, len=0, data ) ;
endtask
```

Snippet 14. ddrc_sample_sequence.sv

```
class ddrc_sample_sequence extends auvs_tpi_sequence ;

    ddrc_hal#(auvs_tpi_sequence) ddrc ;
    ddrc_hal_callback ddrc_cb ;

    task set_hal();
        // Create HAL and HAL callback instances and add callback instance to HAL.
        // HAL callback tasks(bus_write & bus_read) are overridden by ddrc_cb bus tasks.
        ddrc = new("ddrc");
        ddrc_cb=new("ddrc_cb");
```

```
        ddrc_cb.set_seq( this );
        uvm_callbacks#(hal_base,hal_callback_base)::add(ddrc,ddrc_cb);
    endtask

    task pre_body();
        set_hal();
    endtask

    task body();
        // Start DDRC initialization using a DDRC HAL API task
        ddrc.init();
    endtask

endclass
```

In summary, FLS scheme allows us to create test sequences independent of specific SFR port protocols by using HAL API tasks. Since bus operation tasks underneath API are implemented in a form of callback, HAL's reusability is guaranteed simply by changing such bus operation tasks even if SFR port protocols change.

V. CONCLUSIONS

This paper classifies reuse problems which SoC verification engineers can easily encounter on the job into three groups: *Testbench build*, *Platform reusability of test sequences*, and *Vertical and horizontal reusability of test sequences.* In the following, a couple of separate practical schemes as solutions to those problems are presented to improve the reusability of verification components in order to reduce the total amount of AMBA based SoC verification time.

First, *AMBA Unified Verification System* (AUVS) provides a pre-configured AMBA verification platform and VIP independent AMBA transaction generation mechanism. The adoption of AUVS helps users to reduce testbench build-up time (*Problem 1*) and remove test sequences' dependency on VIPs (*Problem 2*). Second, *Firmware-Like Sequences* (FLS) is a HAL based scheme in order to reuse test sequences even if SFR port protocols change by describing test sequences through HAL (*Problem 3*).

Applying both AUVS and FLS together, it is expected to achieve the reusability of verification components in all aspects: vertical, horizontal and platform, which means verification engineers can be more likely to concentrate on actual verification works itself rather than preparing for tests by dramatically saving time and efforts put in testbench build previously.

REFERENCES

[1] Hans van der Schoot, "UVM & Emulation," Mentor, 2014

[2] IEEE Standard for System Verilog. 2012.

[3] Accellera System Initiative. "Standard Universal Verification Methodology (UVM)."