Layered Testbench Architecture for Serial Protocol using UVM

Joe McCann
Gaurav Brahmbhatt
Gaurang Chitroda
Manish Patel
Pinal Patel

Synopsys Inc.,
Dublin, Ireland
www.synopsys.com

eInfochips,
Sunnyvale, USA
www.einfochips.com

ABSTRACT:

This paper proposes an organized, robust, manageable and efficient UVM compliant testbench for verifying any serial protocol (e.g. PCIe, USB etc.). This verification environment uses a "Layered Testbench Architecture". The layering uses two types of hierarchies: Module hierarchy and Class hierarchy. Module hierarchy consists of Device Harness, Link Harness and Test Top modules, while the Class hierarchy consists of Device Layer, Link Layer and Test Layer; each of them corresponds to a UVM environment. Every layer in the Class hierarchy consists of a virtual sequencer, testcase can access any sequence in the hierarchy owing to this virtual sequencer hierarchy. To avoid polluting lower layer with unnecessary dependencies from its parent, we use UVM configuration (uvm_config_db) hierarchy. APIs make sure that testcases are user friendly and readable. By extending this layered testbench approach to create whitebox environment for each layer of serial protocol, we can verify any serial DUT thoroughly.

Table of Contents

Table of Figures

1. Introduction

With the increasing demand for speed in computing and communication, the serial protocols (e.g. PCIe, USB, Ethernet etc.) have evolved heavily over the years. Nowadays, these protocols are common in any SoC (System-On-Chip) related to computing, data processing and communication. Typically a serial protocol consists of a **link** or an **interconnect** that connects two devices following the protocol. The diagram in Figure 1.1 shows two devices (Device A and Device B) connected through a serial link/interconnect:

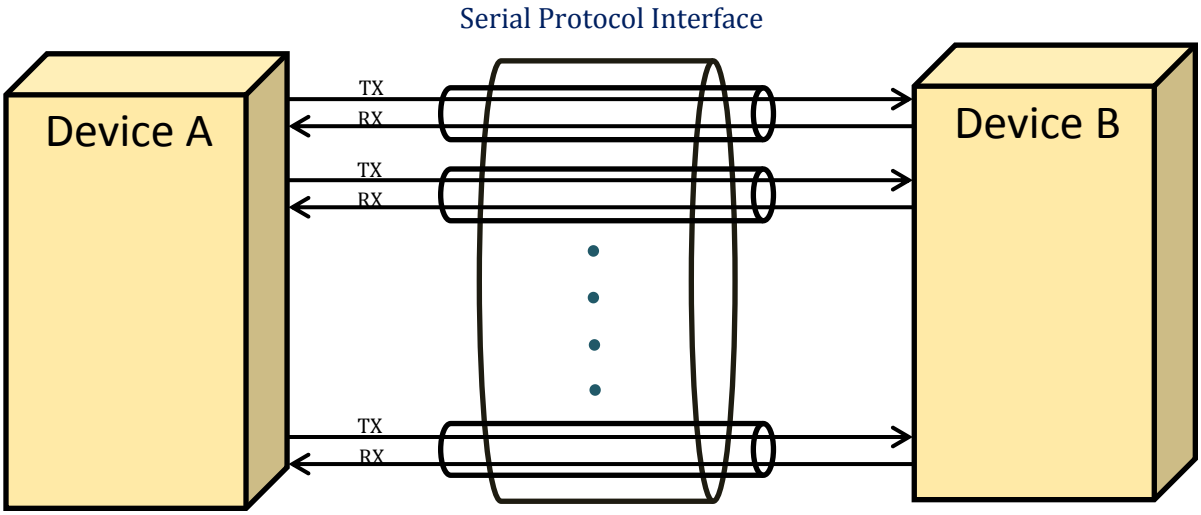

Figure 1.1 Serial Protocol

The continuous evolution in these serial protocol standards have resulted in higher design complexity, which in turn requires thorough verification. The verification of DUT (Design Under Test) compliant to a serial protocol needs another device to connect and communicate with it. This other device is called as "Remote Link Partner". While the remote link partner for a serial protocol DUT can be a VIP (Verification Intellectual Property), we may also choose another instance or a replica of the DUT as the other device.

The serial protocol standards keep on adding new mandatory and optional features, which result in regular design enhancements. To keep up with the pace of design improvement, the testbench architecture must be easily expandable. The design IPs allow many design variants (configurations) based on different optional features. To use the same testbench for different variants of the same design IP, we need testbench to be very flexible. As we intend the same testbench to work equally well with different designs and with different remote link partners (VIP or DUT Replica), we need the testbench architecture to be highly reusable. All these testbench requirements (flexible, expandable and reusable environment) are fulfilled by using a "Layered Architecture".

1.1 What is Layered Testbnech Architecture?

In each serial link, there are two devices to control. Many times, it is desirable to perform a certain set of operations so that both the devices are synchronized and are in a similar state of operation. To do this, we use data encapsulation and abstraction provided by SystemVerilog OOP. The idea is to have different abstraction layers in the testbench, wherein the higher layers

are devoid of complexities of the lower layers. The diagram in Figure 1.3 represents the basic idea of the layered testbench architecture:

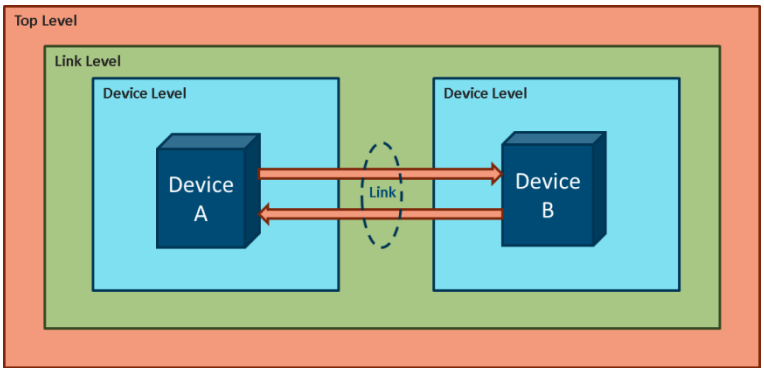

Figure 1.2 Basic Layered Testbench Architecture

There are three basic layers in this testbench architecture: Device Layer, Link Layer and Top Layer. As mentioned earlier, one device is DUT and the other device can be VIP/RDUT (DUT Replica or Remote DUT). This basic layering remains the same for both the DUT-VIP and DUT-RDUT connections. (Note: It is also possible to have VIP-VIP connection, but it may not be of interest.)

The device layer is related to the functionalities and protocol rules for each device, which describe the device behavior when configured for a specific setting. The device layer implementation varies based on type of device. The link layer relates to the interconnect/link level functionalities and rules. It is a higher abstraction level, responsible for configuring both the devices. In addition, the link layer also registers the traffic exchanged between two devices for the purpose of data integrity checks and score boarding. The top layer may consist of only a single link or multiple links, each of them will have two devices connected with each other.

Apart from these layers, it is also possible to have “Whitebox Layers”, which relate to different layers within the protocol standards. Typically, serial protocols have Application Layer, Transaction Layer, Data Link Layer and Physical Layer as part of their functionality. One may choose to do detailed verification of each layer functionality and in this case, “deep” layers under Device Layer can be introduced by using a whitebox approach.

Using SystemVerilog polymorphism and UVM factory concepts, we can extend this architecture for variety of serial protocol designs. UVM’s reusability, modularity, configuration and sequence mechanisms help in developing robust, manageable, flexible and scalable verification environment.

2. Basics of Layered Testbench Architeture

The Layered testbench architecture is mainly partitioned into two distinct hierarchies.

1) Module based hierarchy, which includes all the modules and physical interfaces.
2) Class based hierarchy, which includes the test environment and all the testcases. This is automatically instantiated and executed at run time, via UVM run_test() method call.

2.1 Module Hierarchy

A module-based hierarchy consists of different layers like Device Harness, Link Harness and Test Top modules. Each of these layers (modules) consists of other module instantiations, physical interface instances required port connectivity and additional procedural logic required. The diagram in Figure 2.1 shows the module hierarchy used in testbench:

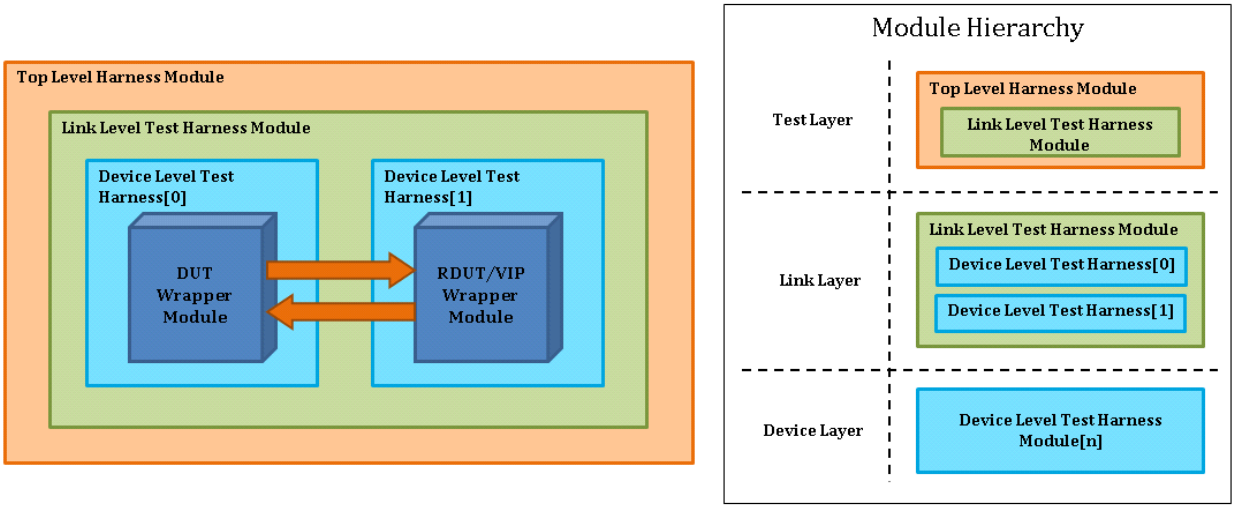

Figure 2.1 Module Hierarchy

Device Level Test Harness Module:

The Device Test Harness module is where a device is instantiated; the device could be DUT, RDUT or VIP. The physical interfaces used to connect to DUT's ports are also instantiated here. Usually for a DUT/RDUT/VIP, a wrapper module is created and other modules like clock-reset generator, PHY module (if not a part of DUT), etc. are instantiated and connected within this wrapper module. Optionally, SVAs can be bound inside wrapper module for DUT/RDUT, as all the ports and physical interfaces are directly accessible there. All the physical interfaces are set through uvm_config_db from this level for the usage of testbench components. The device level module consists of parameters to indicate link number, device index (0 or 1) and parameter to indicate device type (DUT/RDUT/VIP). The following code snippet explains how a DUT/RDUT/VIP wrapper and DUT/RDUT/VIP device harness modules are created.

Notes:

1. A macro `GET_ENV_PATH_STR provides a full hierarchical path to a particular device environment. For example, `GET_ENV_PATH_STR(0,1) may return a string looking like "test_top.m_link_env[0].m_device_env[1]".

2. An easy way to determine whether one needs a DUT-to-VIP connection or a DUT-to-RDUT connection is to have a define macro. For example, a macro `DUT_TO_RDUT indicates it is DUT-to-RDUT connection. This allows a command-line based control to change the type of connection by simply passing +define+DUT_TO_RDUT at compile time.

```
// ---------------------------------------------------------------
// Module: dut_device_wrapper
// ---------------------------------------------------------------
module dut_device_wrapper(Link_Intf link_if);

  // Important Parameters
  parameter LINK_INDEX=0; // Belonging to which link
  parameter DEVICE_INDEX=0; // Device0 or Device1
  parameter IS_DUT=0; // DUT or VIP
  // Connectivity Signals
  wire clock, reset;
  ...
  // Interface Instantiations
  Intf1 u_if1( clock, reset );
  Intf2 u_if2( clock, reset );
  ...
  // Set virtual interfaces using UVM config_db
  // for testbench components'usage
  initial begin
    uvm_config_db#(virtual Intf1)::set(uvm_root::get(),
                   `GET_ENV_PATH_STR(LINK_INDEX,DEVICE_INDEX),
                   "intf1", u_if1);
    uvm_config_db#(virtual Intf2)::set(uvm_root::get(),
                   `GET_ENV_PATH_STR(LINK_INDEX,DEVICE_INDEX),
                   "intf1", u_if2);
  end
  // Module Instantiations
  clk_rst u_clk_rst ( ..., clock, reset, ... );
  phy     u_phy_device ( ... );
  dut     u_dut_device( ...,
                        clock,
                        reset,
                        .signal1(u_if1.signal1),
                        .signal2(u_if1.signal2),
                        .signal3(u_if1.signal3),
                        ...
                        .signal10(u_if2.signal10),
                        .signal11(u_if2.signal11),
                        .signal11(u_if3.signal12),
                        ...
                      );

endmodule : dut_device_wrapper
// ---------------------------------------------------------------

// ---------------------------------------------------------------
```

```
// Module: vip_device_wrapper
// ---------------------------------------------------------------
module vip_device_wrapper(Link_Intf link_if);
  // Important Parameters
  parameter LINK_INDEX=0; // Belonging to which link
  parameter DEVICE_INDEX=0; // Device0 or Device1

  // VIP device instantiation
  // This maybe specific to VIP
  ...
  initial begin
    uvm_config_db #(string)::set(
           uvm_root::get(),
           {dev_env_path,"*"},
           "vip_inst_name",
           vip_env_inst_name
    );
  end
endmodule : vip_device_wrapper
// ---------------------------------------------------------------

// ---------------------------------------------------------------
// Module: device_test_harness
// ---------------------------------------------------------------
module device_test_harness ( Link_Intf link_if );
  parameter LINK_INDEX = 0;
  parameter DEVICE_INDEX = 0;
  parameter e_device_env_type DEVICE_HARNESS_TYPE = DUT;
  parameter RDUT_ENABLE=0;

  // generate device wrapper
  generate
    case(DEVICE_HARNESS_TYPE)
      // dut device
      DUT:
          if (RDUT_ENABLE==0) begin: g
             dut_device_wrapper #(
              .LINK_INDEX   (LINK_INDEX  ),
              .DEVICE_INDEX (DEVICE_INDEX),
              .IS_DUT       (1)
              ) u_wrap (.link_if  (link_if));
          end
`ifdef DUT_TO_RDUT
          else begin: g
             dut_device_wrapper #(
              .LINK_INDEX   (LINK_INDEX  ),
              .DEVICE_INDEX (DEVICE_INDEX),
              .IS_DUT       (1)
              ) u_wrap (.link_if  (link_if));
          end
`endif
      // vip device
```

```
      VIP:
          begin: g
             vip_device_wrapper #(
              .LINK_INDEX   (LINK_INDEX  ),
              .DEVICE_INDEX (DEVICE_INDEX),
              .IS_DUT       (0)
              ) u_wrap (.link_if  (link_if));
      end
    endcase
  endgenerate

  // let the uvm world know if this device is a RDUT instance
  initial begin
    uvm_config_db#(int)::set(uvm_root::get(),

$sformatf("link_env[%0d].deviceenv[%0d].*",
                            link_index, device_index),
                            "is_rdut",
                            rdut_en);
  end
endmodule : device_test_harness
// ---------------------------------------------------------------
```

Link level Test Harness Module:

The Link Level Test Harness module consists of two instances of Device Test Harness. For each link, devices are accessed using indices 0 and 1, which allow easy hierarchical reference. Each of Device[0] and Device[1] can be a DUT or a VIP, and hence allowing DUT-RDUT, DUT-VIP and VIP-VIP type of connections. Two serial protocol devices are connected through an interconnect at this level of hierarchy. A unique Link index parameter is used for making a reference to a particular link in case a protocol allows a hierarchical structure with multiple links.

```
// ---------------------------------------------------------------
// Module: link_test_harness
// ---------------------------------------------------------------
`define NO_OF_DEVICES 2
module link_test_harness ( Link_Intf link_if );
  parameter LINK_INDEX = 0;
  // Interface array
  Link_Intf link_if[`NO_OF_DEVICES];
  // Generate test harness
  genvar i;
  generate
    for (i=0; i<`NO_OF_DEVICES; i=i+1)
    begin: g
      // Device test-harness instances
     device_test_harness #(
           .LINK_INDEX(LINK_INDEX),
           .DEVICE_INDEX(i),
           .DEVICE_HARNESS_TYPE( get_dev_env_type(LINK_INDEX,i))
           ) u_device_th (.link_if (link_if[i])
      );
```

```
    end
  endgenerate
  // Connection between serial protocol interconnect wires
  // This varies from protocol to protocol
  ...
  ...

  // Following function is used to determine whether a particular
  // device(Device0/1) is of type DUT or VIP.
  // Macro `DUT_IS_DEVICE_0 is defined through a command-line
  // argument. This is used for device index assignment.
  function int get_dev_env_type(int link_index, int dev_index);
    `ifdef DUT2RDUT
      return DUT;
    `else
       ìfdef DUT_IS_DEVICE_0
         case(dev_index)
           0 : return DUT;
           1 : return VIP;
         endcase
      `else
         case(dev_index)
           0 : return VIP;
           1 : return DUT;
         endcase
      `endif // DUT_IS_DEVICE_0
    `endif  // DUT2RDUT
  endfunction: get_dev_env_type

endmodule : link_test_harness
// ---------------------------------------------------------------
```

Top Module:

The Top-level testbench module is at the highest level in the hierarchy, which encapsulates the whole testbench and the DUT. In this module, the link level test harness is instantiated, based on number of links required. For a simple single link case, there will be only a single instance of link level test harness. However for a multi-link case, generate block can be used to create multiple instances of link harness. What lies in between multiple links and how different link communicate among each other is out of scope of this document and can be protocol specific. This encapsulated test level harness approach permits reuse of the actual interface instantiation; mapping and we can take multiple instances of virtual interface as per requirement.

The UVM test class hierarchy communicates with the module hierarchy (and the DUT), via virtual interface handles to each physical interface instance. The physical interfaces are instantiated within the <dut/vip>_device_wrapper wrapper modules. The virtual interface handles are passed from wrapper modules to the UVM configuration database, and subsequently retrieved from the database in the appropriate device-level verification environment, within the UVM class hierarchy.

```
// ---------------------------------------------------------------
// Module: top
// ---------------------------------------------------------------
module top ();
  generate
    for (i=0; i<`NO_OF_LINKS; i=i+1) begin
      // Link level test-harness
      link_test_harness #(.LINK_INDEX(i)) u_link_th();
    end
  endgenerate
  // Run Test
  initial begin
    run_test();
  end
endmodule : top
// ---------------------------------------------------------------
```

2.2 Class Hierarchy

2.2.1 Environment Hierarchy

A class-based hierarchy includes Device environment (layer), Link environment (layer) and Test layer. This layering allows a systematic but still easy access and maintainance. The testbench modules and class hierarchy has been layered to align with each other as closely as possible, which enables ease of maintenance and reuse of all components in the UVM based class hierarchy. In a class hierarchy, we use polymorphism to our advantage as we override many base classes while creating the actual environment.

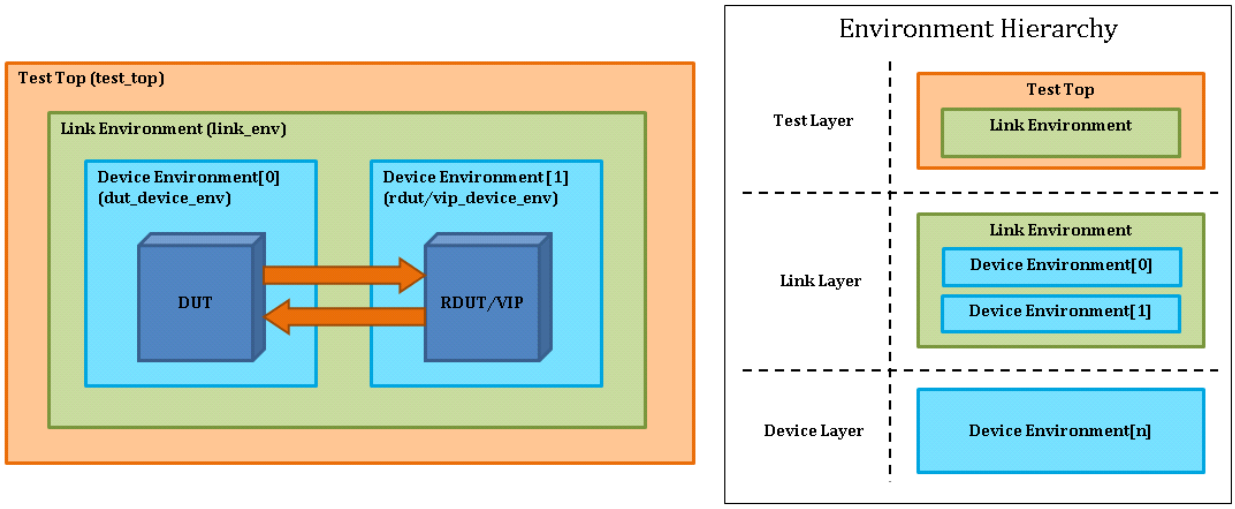

Figure 2.2 Environment Hierarchy

Device Layer (Device Environments):

A “Device Environment” is a base class with all common functionality for a device following a particular serial protocol. We override this class and create a "Custom Device Environment” based on a specific DUT. This way, we can reuse the same testbench infrastructure for variety of DUTs. The other device in the link (Remote Link Partner) can be of two types: a VIP Device or a Replica of DUT. For DUT device, an extended “DUT Device Environment” is created, while for VIP device “VIP Device Environment” is created. The device environment consists of handles of configuration descriptor member and device level virtual sequencer. Following is the pseudo-code for base class device_base_env:

```
// ---------------------------------------------------------------
// Class: device_base_env
// ---------------------------------------------------------------
class device_base_env extends uvm_env;
  // Device Config handle
  device_cfg m_device_cfg;

  // Virtual Sequencer Handle
  device_vseqr m_device_vseqr;

  // UVM Factory Registration
  `uvm_component_utils(device_base_env)

  // Constructor
  function new( string name="device_base_env",
                uvm_component parent=null);
    super.new(name, parent);
  endfunction : new

  // Build Phase
  function void device_base_env::build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(device_cfg)::get(this, "",
                                        "device_cfg",device_cfg))
      `uvm_fatal("CFGDB/GET","Failed to get device_cfg from UVM config
DB.")
    uvm_config_db#(device_cfg)::set(this,"*","device_cfg",
                                    m_device_cfg);
    m_device_veqr = device_base_vseqr::type_id::create
                    ("m_device_vseqr",this);
  endfunction : build_phase
endclass : device_base_env
// ---------------------------------------------------------------
```

The following code shows extended classes for “DUT Device Environment” and “VIP Device Environment”:

```
// ---------------------------------------------------------------
// Class: dut_device_env
// ---------------------------------------------------------------
class dut_device_env extends device_base_env;
```

```
  // DUT level config descriptor
  dut_cfg m_dut_cfg;
  // DUT virtual sequencer
  dut_vseqr m_dut_vseqr;
  // Register shadow model
  dut_reg_model m_reg_shadow;
  ...
  ùvm_component_utils(dut_device_env)

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Factory Overrides
    set_inst_override_by_type( “m_device_vseqr”,
                               device_base_vseqr::get_type(),
                               dut_vseqr::get_type()
                             );
    if(!$cast(m_dut_vseqr, m_device_vseqr))
      `uvm_fatal("CAST","Dynamic cast failed to m_dut_vseqr.")
    if(!$cast(m_dut_cfg, m_device_cfg))
      `uvm_fatal("CAST","Dynamic cast failed to m_dut_cfg.")
    ...
  endfunction : build_phase
  ...
endclass : dut_device_env
// ----------------------------------------------------------------

// ----------------------------------------------------------------
// Class: vip_device_env
// ----------------------------------------------------------------
class vip_device_env extends device_base_env;
  // VIP Component
  vip_comp m_vip_comp;
  // VIP level config descriptor
  vip_cfg m_vip_cfg;
  // VIP virtual sequencer
  vip_vseqr m_vip_vseqr;
  ...
  ùvm_component_utils(vip_device_env)
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Factory overrides
    set_inst_override_by_type("m_device_vseqr",
                               device_vseqr::get_type(),
                               vip_vseqr::get_type());
    if(!$cast(vip_vseqr, m_device_vseqr))
      `uvm_fatal("CAST","Dynamic cast failed to vip_vseqr.")
    if(!$cast(m_vip_cfg, m_device_cfg))
      `uvm_fatal("CAST","Dynamic cast failed to m_vip_cfg.")
    ...
  endfunction : build_phase
  ...
```

```
endclass : vip_device_env
// ----------------------------------------------------------------
```

Link Layer (Link Environment):

The "Link Environment" consists of two instances of device environment. Either "DUT Device Environment" or "VIP Device Environment" overrides each device environment handle. The device_base_env instance is not used directly, and is essentially a placeholder for the appropriate device environment class, which is extended from it. Each supported device environment class derives from the device_base_env class, and adds whatever additional functionality is required to support that particular device type. The device_base_env instances in the link_env are overridden during UVM build_phase(), to the appropriate device environment class, via a UVM factory override.

```
// ----------------------------------------------------------------
// class: link_env
//  link-level verification environment
// ----------------------------------------------------------------
class link_env extends uvm_env;
  // link-level configuration descriptor
  link_cfg m_link_cfg;

  // device-level environments (2 per link)
  // this is an array of base-class handles, that are overridden by
  // the derived device environments via a uvm factory override.
  device_base_env m_device_env[2];

  // Arrays of derived device env handles, actual allocation depends
  // on configured device types of link partners.
  dut_env m_dut_env[2];
  vip_env m_vip_env[2];

  // link level virtual sequencer
  link_vseqr m_link_vseqr;

  // constructor
  function new(string name="link_env", uvm_component parent=null);
  endfunction : new

  //  uvm build phase method
  function void build_phase(uvm_phase phase);
    dut_cfg   m_dut_cfg[2];
    super.build_phase(phase);
    // get the link-level configu descriptor from parent
   if(!uvm_config_db#(link_cfg)::get(this,"","m_link_cfg",m_link_cfg))
     `uvm_fatal(get_name(),"failed to get link_cfg.")

    // create the device level envs
    foreach (m_device_env[i])
    begin
      // factory override, to support multiple env types
      case(m_link_cfg.m_device_cfg[i].m_device_envtype)
        dut_device:
```

```
          set_inst_override_by_type(
            $sformatf("m_device_env[%0d]",i),
            device_base_env::get_type(),
            dut_env::get_type()
          );
        vip_device:
          set_inst_override_by_type(
            $sformatf("m_device_env[%0d]",i),
            device_base_env::get_type(),
            vip_env::get_type()
          );
        default:
          `uvm_fatal(get_name(),
            $sformatf("unsupported m_device_envtype=%s",
            m_link_cfg.m_device_cfg[i].m_device_envtype.name())
          )
      endcase

      // create env
      m_device_env[i] = device_base_env::type_id::create(
        $sformatf("m_device_env[%0d]",i), this);
    end

    // create link virtual sequencer
    m_link_vseqr = link_vseqr::type_id::create("m_link_vseqr",this);

  endfunction : build_phase
endclass : link_env
// ---------------------------------------------------------------
```

Test Layer:

At the "Test layer" (or in Test Top module), we may have either a single or multiple link environments (or link harness instances). The top level of the UVM class hierarchy for each simulation is a test class that is selected dynamically at run time from all compiled test cases. All test-case classes derive from base_test, which in turn derives from uvm_test. The base_test class, as shown in Figure 2.3, instantiates the top-level environment (link_env). If protocol supports multi - link then using polymorphism concept we can create multiple instances of Link Env for each link by overriding the base class. The link_env is architected so that any number of DUT / VIP device environments can be derived from the device_base_env class and used as a link partner.

2.2.2 Configuration Hierarchy

It is extremely important that we do not pollute each layer with unnecessary information or dependencies from its parent layer and this architecture is well equipped to achieve this. We add UVM configuration (uvm_config_db) members at each layer. Therefore, we have a Cfg class based hierarchy as well. As mentioned in the Figure 2.3, a device environment consists of "Device Config", a link environment consists of "Link Config", and testcase consists of "Test Config".

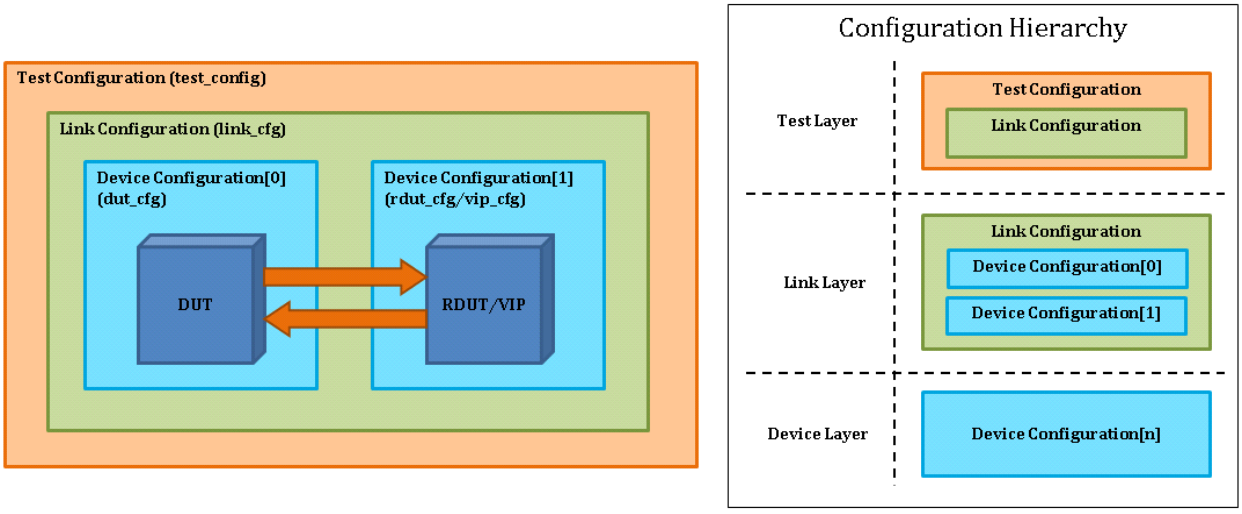

Figure 2.3 Configuration Hierarchy

Device Config:

The Device Config class is a device level configuration descriptor base class, which contains all device environment random configuration members that are common to all derived DUT / VIP environment. Each derived device environment extends this class to add device specific configuration descriptor members. The device environment base class gets a reference to the descriptor from the uvm_config_db(). The descriptor is device type agnostic, i.e. it is the responsibility of the derived device environment to translate the configuration members, if necessary.

Link Config:

The Link Config class contains all link-level environment random configuration parameters, and instantiates a device environment configuration descriptor instance per device environment (DeviceCfg), which are then passed to the device environments via the UVM configuration database, before the device environments are built. A "Link Config" consists of two instances of "Device Config", which in turn will be the references to configuration members of "Custom/Overridden Device Environment".

Test Config:

The Test Config class contains all top-level test configuration parameters required by the test cases. The base_test class instantiates the top-level environment and the top-level configuration descriptor (test_cfg). test_cfg consists of handle/reference of "Link Config" descriptor.

Following code snippets explain the hierarchy of UVM configuration descriptor:

```
// ------------------------------------------------------------------
// class: device_cfg
// Device-level configuration descriptor
// ------------------------------------------------------------------
class device_cfg extends base_cfg;
  ...
endclass : device_cfg

// ------------------------------------------------------------------
```

```
// class: dut_cfg
//  dut specific configuration descriptor
// ------------------------------------------------------------------
class dut_cfg extends device_cfg;
  // pre and post_randomize() methods can be override
  // and can do dut specific implementations
  // dut device specific logic can be implemented here
  ...
endclass : dut_cfg

// ------------------------------------------------------------------
// class: vip_cfg
// vip specific configuration descriptor
// ------------------------------------------------------------------
class vip_cfg extends device_cfg;
  ...
endclass : vip_cfg

// ------------------------------------------------------------------
// class: link_cfg
// link level configuration descriptor
// ------------------------------------------------------------------
class link_cfg extends base_cfg;
  // variable: m_device_cfg[]
  device_cfg m_device_cfg[2];

  // Constructor
  function new (string name);
    super.new(name);
    foreach(m_device_cfg[i])
    begin
     ...
     // cast derived device config descriptor to base class handle
     case(device_env_type)
      // dut derived device config descriptor
      dut_device:
      begin
        if(!$cast(m_device_cfg[i], dut_cfg::type_id::create(inst)))
          `uvm_fatal("cast",$sformatf("dynamic cast failed to %s, from
dut_cfg type.",inst))
      end
      //  vip derived device config descriptor
      vip_device:
      begin
        m_vipindex  = e_deviceindex'(i);
        if(!$cast(m_device_cfg[i], vip_cfg::type_id::create(inst)))
        begin
          `uvm_fatal("cast",$sformatf("dynamic cast failed to %s, from
vip_cfg type.",inst))
        end
      end
    endcase
```

```
  end
endfunction : new

endclass : link_cfg
// ----------------------------------------------------------------------

// ----------------------------------------------------------------
// class: testcfg
//  Default test-level configuration descriptor
// ----------------------------------------------------------------
class test_cfg extends base_cfg;

  // link-level configuration descriptor(s)
  // the index value should be number of links present
  rand link_cfg m_link_cfg[1];

  function test_cfg::new(string name);
    super.new(name);
    // create link-level config descriptor(s)
    foreach(m_link_cfg[i])
    begin
      // create link descriptor
      m_link_cfg[i] = link_cfg::type_id::create(
        $sformatf("m_link_cfg[%0d]",i),
      );
    end
  endfunction: new

endclass : test_cfg
// ----------------------------------------------------------------
```

3. Sequencer Layering and APIs

3.1 Sequencer Hierarchy

Just like the environment hierarchy, we also create the virtual sequencer hierarchy. This is shown in Figure 3.1.

Link Level Virtual Sequencer:

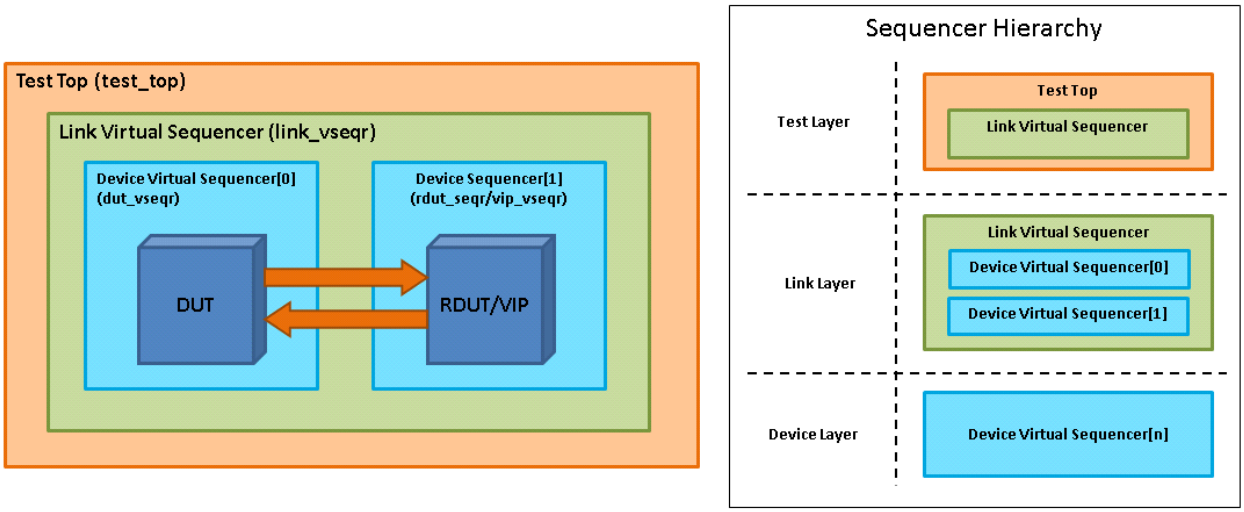

Figure 3.1 Sequencer Hierarchy

Device Level Virtual Sequencer:

The device environment consists of a handle of "device virtual sequencer" (device_base_vseqr class). "DUT device virtual sequencer" or "VIP device virtual sequencer" eventually overrides this. In device_base_vseqr, we have references to all lower level sequencers.

Link Level Virtual Sequencer:

Link virtual sequencer is a top-level virtual sequencer in the test environment utilized by the testcases to execute link level virtual sequences. This top-level control of the test environment lies within link virtual sequencer. The link environment consists of a handle of "link virtual sequencer" (link_vseqr class), while this class link_vseqr consists of two instances of device_base_vseqr. While building the environment, dut_device_vseqr or vip_device_vseqr overriddes each of these device_base_vseqr instances. A hierarchy of sequencers is created by assigning these device virtual sequencer instances to the references of the same in each device environment.

The testcase assigns a virtual sequence to run as the default sequence in each of the UVM phases. (Note: If needed, we can turn this off and run only the required default sequences.) Each test will assign a virtual sequence to the uvm_main_phase(), which is essentially the main user test phase, where data transfers are performed. As link_vseqr contains references to device_base_vseqr, device-level virtual sequences can be executed from the link-level virtual sequences.

3.2 APIs (Application Programming Interfaces)

We often realize that there are few common operations to perform from every test before it performs its specific actions. For example, power-on-reset, register shadow model creation, configuration register programming, tying few DUT signals to default values, and so on. These operations are “always required”, “performed frequently”, or both. To handle such activities from testcases, we can create Boot-up like sequences that perform these common tasks.

Further, we can create tasks that execute a set of sequences that perform a series of operations. These tasks are the APIs; they embed multiple sequences within and reduce the chances of test sequences being faulty.

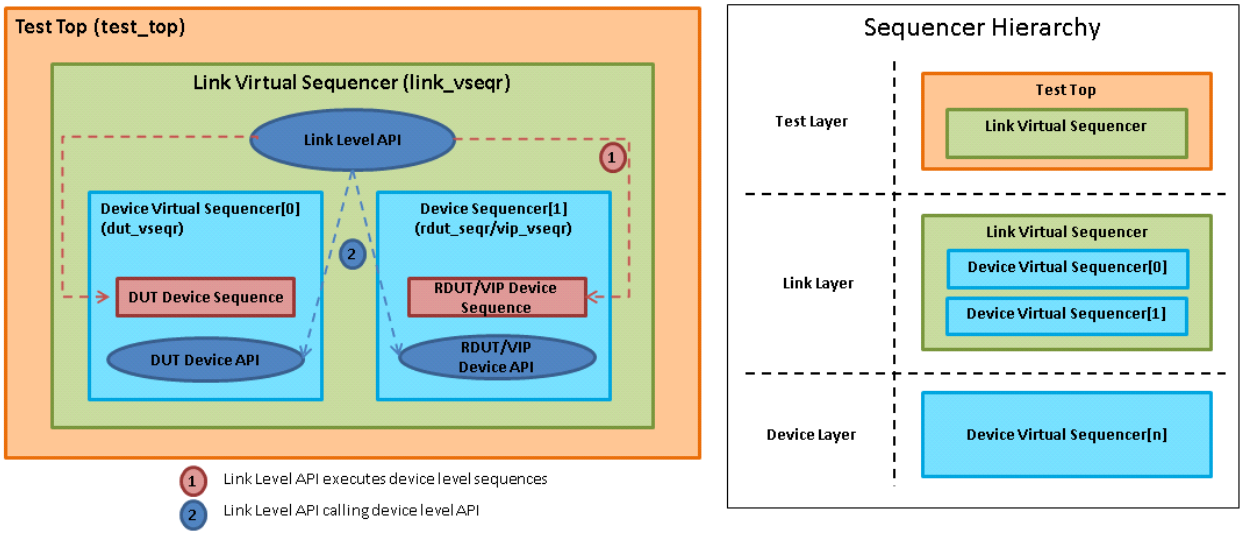

Figure 3.2 API Hierarchy

As discussed in the earlier section, using sequencer hierarchy any lower level sequence can be called using link virtual sequencer. We can use this to our advantage to implement the link environment API tasks calls, which in turn call other sequences and execute them on appropriate sub-sequencers. APIs make the testcases very easy and efficient by avoiding complexity of calling too many low level sequences manually. Figure 3.2 shows this by dotted path marked as number 1.

We can also create APIs hierarchy. This is done by having device level APIs for both the DUT device and the VIP device. The link level APIs call required device level APIs. This is of great advantage, as any change in device level API would automatically reflect in link level API. Figure 3.2 shows this by dotted path marked as number 2.

Figure 3.3 shows the sequence execution flow from link level to DUT/VIP level in the layered testbench architecture:

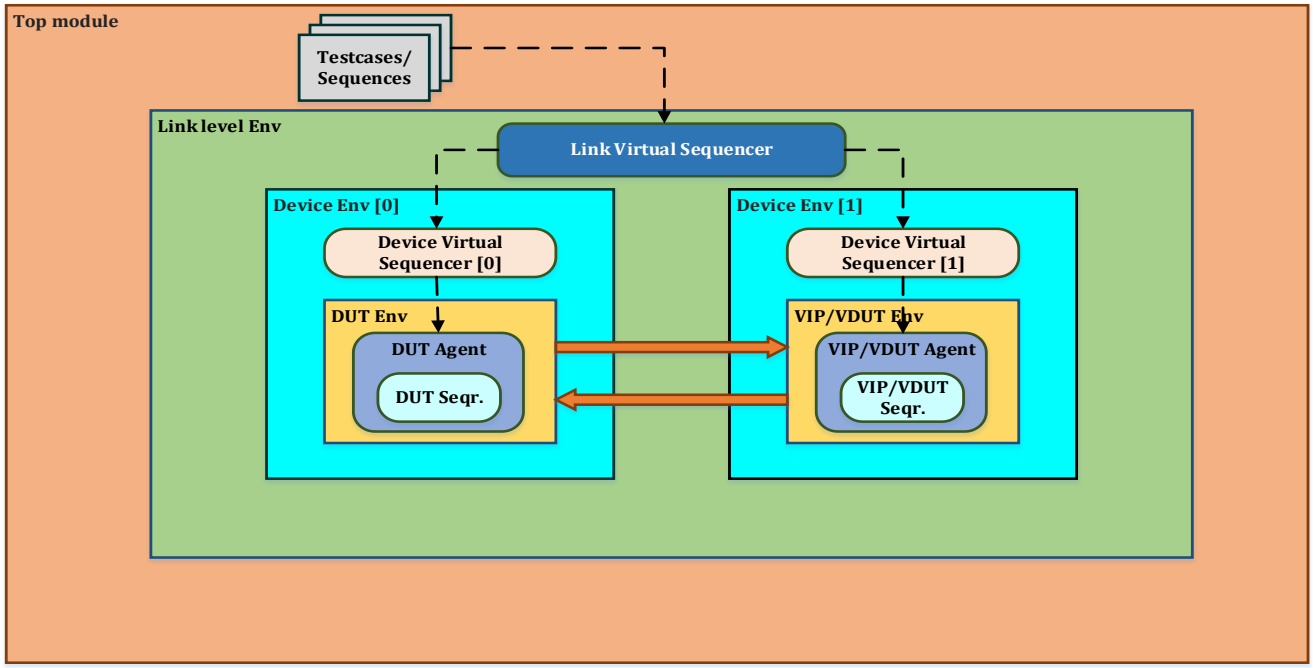

Figure 3.3 DUT Level Sequence Flow in Layered Testbench

From the testcases, link level sequence executes on the link virtual sequencer. The link level sequence will in turn execute device level sequences on the device virtual sequencers (for both device0 and device1). Further, a device virtual sequencer (0 or 1) will call other sequences that run on specific lower level sequencers. Based on type of device, DUT or VIP sequences are executed on the DUT device sequencer or VIP device sequencer.

We can create device level and link level default sequences for all the phases of UVM. By doing this, we exploit both UVM phases and hierarchical sequences to our advantage because all common operations are very well segregated among these default sequences. This ensures that a proper flow of operations is maintained and minimize the issues with test sequences.

Following code snippets show some example of APIs in the layered testbench:

Example 1: Device and Link Level Sequences for reset and related API task

```
// ---------------------------------------------------------------
// DUT Reset Default Sequence
// ---------------------------------------------------------------
class dut_reset_default_vseq extends device_base_vseq;
  ...
  ...
endclass : dut_reset_default_vseq

// VIP Reset Default Sequence
class vip_reset_default_vseq extends device_base_vseq;
```

```
  ...
  ...
endclass : vip_reset_default_vseq

class link_reset_default_vseq extends link_base_vseq;
    // this sequence issues reset for dut device
    dut_reset_default_vseq m_dut_reset_seq[2];
    // this sequence issues reset for vip device
    vip_reset_default_seq m_vip_reset_seq[2];
    // Execution of reset sequences for both device
    ...
endclass : link_reset_default_vseq

// ------------------------------------------------------------------
// task: api_link_reset
// issue a full reset for both the devices,
// this task in turns calls 2 below device level's sequence
// ------------------------------------------------------------------
task api_link_reset();
  link_reset_default_vseq l_link_reset_default_vseq;
  // link_reset_default_vseq calls dut anv vip level(2 device level)
  // sequences,which executes reset seq of device level's
  l_link_reset_default_vseq=link_reset_default_vseq::type_id::create
                            ("api_seq");
  if( !l_link_reset_default_vseq.randomize() )
    `uvm_fatal(m_id,"l_link_reset_default_vseq.randomzie failed")
  l_link_reset_default_vseq.start( this );
endtask : api_link_reset
```

Example 2: Calling default phase sequences through a single API by specifying a UVM phase name.

```
// ------------------------------------------------------------------
// task: api_link_rerun_default_phase_vseq
// this task starts spawning the default link virtual sequence for
// specified phase and subsequent phase specified.
//
// Whenever we apply reset, we need to execute all configurations
// required to achieve default working condition of device. For this,
// multiple sequences for phases like reset, post_reset,
// pre_configure, configure, post_configure must be executed.
// We can achieve this easily by calling following API with argument
// “reset_phase”.
// ------------------------------------------------------------------
task linkvseqr::api_link_rerun_default_phase_vseq(string phase);
  link_base_vseq l_vseq;
  link_base_vseq vseq_q[$];
  `uvm_info(m_id,$psprintf("request to re-run default link vsequences
from %0s phase",phase ),UVM_MEDIUM)
  case(phase)
    "pre_reset_phase":
    begin
```

```
      ...
    end
    "reset_phase":
    begin
      l_vseq= link_reset_default_vseq ::type_id::create();
      vseq_q.push_back(  l_vseq ) ;
      l_vseq= link_post_reset_default_seq ::type_id::create();
      vseq_q.push_back(  l_vseq ) ;
      l_vseq= link_pre_configure_default_seq ::type_id::create();
      vseq_q.push_back(  l_vseq ) ;
      l_vseq= link_configure_default_seq ::type_id::create();
      vseq_q.push_back(  l_vseq ) ;
      l_vseq= link_post_configure_default_vseq ::type_id::create();
      vseq_q.push_back(  l_vseq ) ;
    end
    "post_reset_phase":
    begin
      ...
    end
    "pre_configure_phase":
    begin
      ...
    end
    "configure_phase":
    begin
      ...
    end
    "post_configure_phase":
    begin
      l_vseq= link_post_configure_default_vseq ::type_id::create();
      vseq_q.push_back(  l_vseq ) ;
    end
  endcase

  while(vseq_q.size()>0)
  begin
    link_base_vseq l_vseq;
    l_vseq=vseq_q.pop_front();
    if(!l_vseq.randomize())
      `uvm_fatal(m_id,$psprintf("failed to randomize %0s",
                                l_vseq.get_name()))
    l_vseq.start(this);
  end
endtask : api_link_rerun_default_phase_vseq
//---------------------------------------------------------------------
```

4. WhiteBox Layer

The functionality of serial protocols like PCIe, USB, Ethernet is often divided into multiple protocol layers like,
1. Physical Layer
2. Data Link Layer
3. Transaction Layer
4. Application Layer

It could be advantageous (especially for design IPs) to model detailed functionality of each protocol layer in the testbench and carry out a thorough verification for each. This approach requires an access to the internals of DUT, hence we call this "Whitebox" approach. For an instance, we may choose to create whitebox environments for Transaction, Data Link and Physical layers. This is not mandatory exercise, but doing whitebox verification for serial protocol DUT provides a very high level of confidence by verifying the functionality of each layer in depth. Each whitebox environment becomes a sub-layer of "Dut Device Environment". Figure 4.1 shows how whitebox layers are created within "DUT Device Environment".

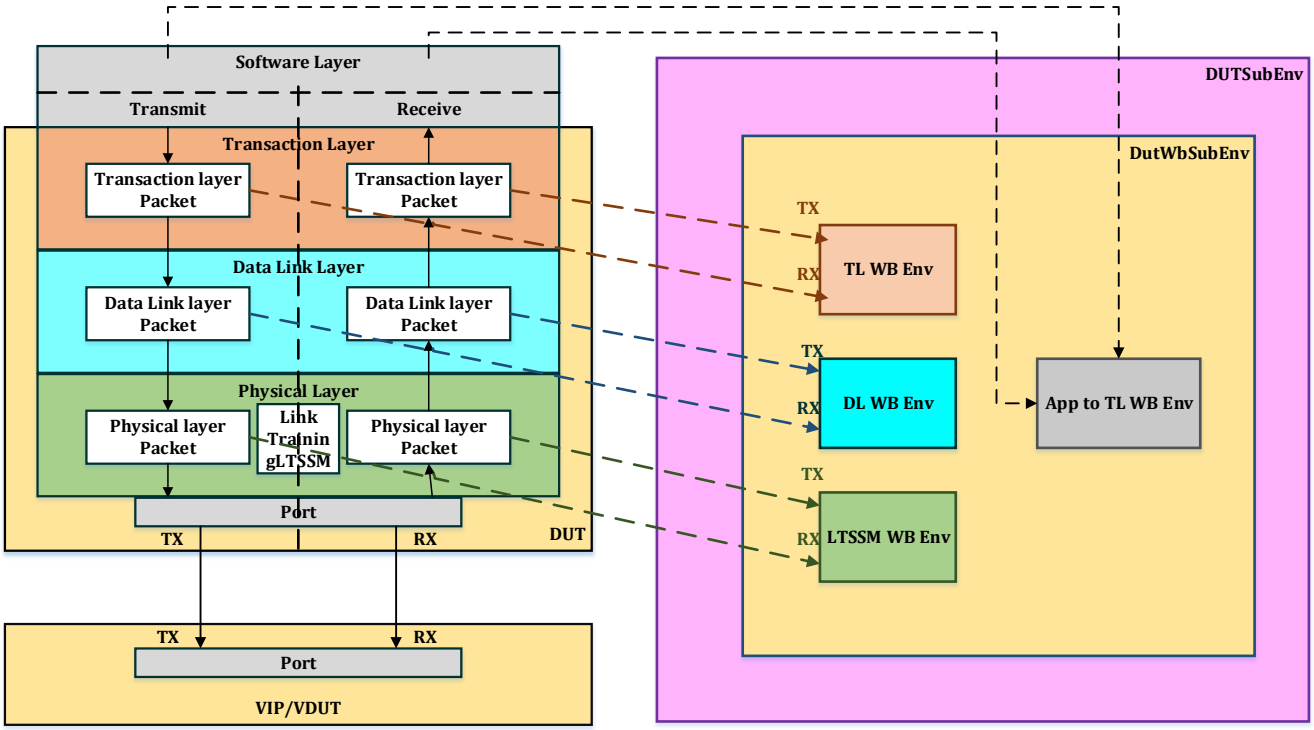

Figure 4.1 WhiteBox Environment

The whitebox environment typically contains the monitors that snoop DUT's internal signals, the emulation models and the scoreboards for the particular protocol layer. We can use random error injection and callback mechanism for generating stimulus that can trigger whitebox checkers and hit corner case scenarios. The completeness of verification of each protocol layer is

measured by both the functional coverage and the stimulus coverage. Each whitebox environment can be switched ‘on’ or ‘off’ as per necessity using a simple parameter passed from the command line. The following code shows example of whitebox environment creation for the physical layer of serial protocol:

```
// ---------------------------------------------------------------
// Class: dut_device_env
// ---------------------------------------------------------------
class dut_device_env extends device_base_env;
  ...
  ...
  // DUT level config descriptor
  dut_cfg m_dut_cfg;
  // Physical layer whitebox enviroment
  pl_wb_env m_pl_wb_env;
  ...
  ùvm_component_utils(dut_device_env)

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ...
    // Set a dut config member within whitebox environment
    uvm_config_db#(dut_cfg)::set(this, "m_pl_wb_env",
                                 "m_dut_cfg", m_dut_cfg);
    ...
    // Physical Layer whitebox environment is created only
    // if corresponding config member is set to 1.
    if(m_dut_cfg.m_physical_wb_enable)
      m_pl_wb_env = pl_wb_env::type_id::create(“m_pl_wb_env”,this);
    ...
    ...
  endfunction : build_phase
  ...
endclass : dut_device_env
// ---------------------------------------------------------------
// Class: dut_cfg
// ---------------------------------------------------------------
class dut_cfg extends device_cfg;

  // Member to control construction of
  // physical layer whitebox environment
  rand bit m_physical_wb_enable;
  ...
  // By default, we always enable wb environment
  constraint c_default {
    ...
    soft m_physical_wb_enable==1;
  }
  ...
  function void post_randomize();
    super.post_randomize();
```

```
    // Plusarg based control to turn off wb environment
    if($value$plusarg("PL_WB_EN")==0)
      m_physical_wb_enable=0;
  endfunction
  ...
endclass : dut_cfg

// ---------------------------------------------------------------
```

5. Advantages of a Layered Testbench Architecture

The layered testbench architecture is very useful approach for verifying serial protocol based DUT. Though it is not only limited to serial protocols as we can use similar approach for other design as well. Following are few of the important advantages of this architecture:

- Adaptability

  The main advantage of this approach is its adaptability. This is because of the strong usage of SystemVerilog polymorphism concept. One can create generic base classes and easily use the same for implementing testbench for different DUTs or protocols.
- Expandable Architecture

  This testbench architecture can be easily expanded to verify a different topologies of serial protocol connections. For example, a PCI Express protocol hierarchy may have a single link or many links connected through switches, the same testbench expands itself to fit to these requirements.
- Works well with different IP variations

  With minimum modifications, this testbench can be used to verify different variations of the same IP design as well as other DUTs.

- Less Implementation Efforts

  This layered testbench architecture initial turnaround time may be slower. But once the basic infrastructure is in place, it takes very less implementation efforts due to the scalable nature of the environment and guarantees an early success.

- User Friendly

  As explained earlier, the hierarchical sequences and APIs make testcase writing very easy and user friendly.

- Higher Rate of discovering bugs

  With whitebox environments in place and the constrained random stimulus, this architecture has been proven in finding many corner design bugs as compared to legacy testbenches.

6. Results/Summary

The Layered Testbench Architecture is a proven way to verify serial protocol designs. This approach renders flexibility, scalability to the environment and makes the testbench manageable. This is of great advantage when we are verifying complex protocol with loads of features. The ability of this architecture to mould itself for different topologies (for example, DUT-to-VIP or DUT-to-DUT connection), different IP variations and different designs and different protocols makes it highly reusable.

Hierarchical sequences and APIs make test sequences accurate, user-friendly and readable. This ensures that minimum time/effort is spent on fixing bad sequence issues. Whitebox environments help us to find many corner case design bugs, hence stepping up the design quality significantly.

7. References

[1] Universal Verification Methodology (UVM) 1.1 Class Reference

[2] PCI Express ® Base Specification Revision 3.1 Version 1.0

[3] UVM Cookbook