Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces

by Shashi Bhutada, Mentor Graphics

INTRODUCTION

The SystemVerilog language is increasing in adoption for advanced verification techniques. This enables the creation of dynamic test environments using SystemVerilog classes among other things. The SystemVerilog virtual interface has been the primary method for connecting class-based SystemVerilog testbenches with a VHDL or Verilog DUT, but this construct has certain limitations, especially when the interface is parameterized. In this article we will discuss some of these limitations and demonstrate an alternative approach called as the Polymorphic Interface. The recommended approach can also work generically wherever one uses virtual interfaces and not just parameterized interfaces.

For the SystemVerilog testbench, we will use OVM infrastructure, but this same discussion applies to UVM testbenches. This article assumes SystemVerilog OVM/UVM class and interface syntactical understanding.

SYSTEMVERILOG OVM/UVM TESTBENCH

Design demands have grown exponentially over time as our fabrication capabilities and geometry sizes have dramatically improved. Verification methods can be advanced with the SystemVerilog language, which provides a whole gamut of methods. SystemVerilog amalgamates advanced methods including Object-Oriented Programming (OOP) by extending a well-known design language: Verilog. Since it is an IEEE standard and packaged with RTL simulators, advanced verification is now easily accessible.

The SystemVerilog language can be divided into five parts. One is an enhancement of Verilog RTL design constructs that captures design intent more accurately. The other four parts are constructs meant to deliver the advanced verification features: assertions; object oriented programming (OOP) or the 'class' syntax; constrained-random stimulus or automatic stimulus generation; and functional coverage.

The next big development in this area was the UVM (Universal Verification Methodology) Library, which is an Accellera standard SystemVerilog code-base jointly developed by multiple EDA Tool vendors and industry leaders. This library simplifies adoption of the object-oriented methods that are part of SystemVerilog. Simply put, UVM tames the SystemVerilog behemoth. UVM is largely based on its predecessor OVM (Open Verification Methodology) Library.

In a traditional testbench, the user is required to use a hardware description language (HDL) to also define the testbench. Unfortunately, the HDL only allowed creating static design elements. This has many disadvantages which include: testbench interfering with design interactions causing race conditions, static testbench prevents plug-n-play, verification reuse meant cut-n-paste of the test code, limited coverage of design state space and no metrics of which states were covered. In an advanced OVM/UVM testbench a user is required to use 'class' syntax and create dynamic testbench components which do not rely on pin-level interactions on the DUT but are at an abstraction above the pin-level interaction called as transaction-level (shown in Figure 1).

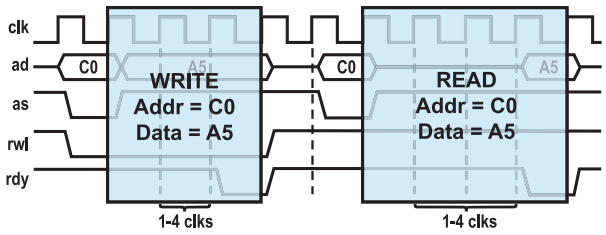


Figure 1: Transaction-level versus Pin-level activity

A typical OVM/UVM testbench (See figure 2) contains a stimulus generator that generates transaction, a driver that converts transactions into pin-level activity, a monitor that transforms pin-level activity back into transaction objects, a coverage collector to measure the transaction activity on the pins, a predictor that predicts what the expected output transaction should be based on the stimulus applied, and a scoreboard that compares observed output transaction with

the expected transaction. This approach has the following advantages: test does not interfere with design under test, dynamic test components are plug-n-play, verification reuse is possible for a given standard bus interface protocol, transaction-level code runs typically faster than pin-level interaction, larger state space can be covered when using constrained random transaction generation, and real metrics can be obtained by observing transactions around the testbench.

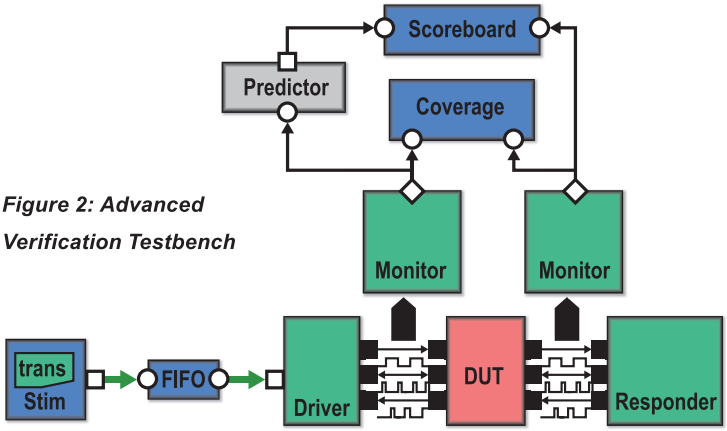


Figure 2: Advanced Verification Testbench

SYSTEMVERILOG INTERFACES AND VIRTUAL INTERFACES:

In the OVM/UVM dynamic transaction-level testbench one needs to ultimately connect with the DUT at different points. The driver needs access to design pins to generate the right pin-activity based on the transaction. Similarly, the monitor needs access to design pins to observe pin-activity. Dynamic constructs cannot access static design items directly.

SystemVerilog provides the 'interface' construct to encapsulate a set of pins on a DUT. The interfaces typically can be design dependent but often they encapsulate industry standard bus protocols for maximum reuse. A SystemVerilog interface is a static construct and resides on the static side of the testbench.

The SystemVerilog language provides the virtual interface construct that allows one to pass the 'interface' handle from the static side to the dynamic side. This connects the class-based dynamic testbench to the static HDL design. Following are the different approaches to pass the interface from static side to the dynamic side:

- Directly setting of the virtual interface handle where needed via helper function in the class.
- Package-based global storage for the virtual interface handle.
- The preferred approach is by saving virtual interface handle into the OVM/UVM configuration database. In OVM, this requires an extra wrapper class to hold the virtual interface.

LIMITATIONS WITH VIRTUAL INTERFACES:

The SystemVerilog virtual interface based approach to connect the OVM testbench and the DUT involves passing an interface handle from the top-level module to the OVM testbench. In most of the use cases virtual interfaces work just fine. As we will see in the example below, virtual interfaces become difficult to use when it comes to parameterized interfaces.

To demonstrate the problem, let's see the example code below which tries to use virtual interfaces:

```
class vif_wrapper #(type T=int) extends ovm_object;
  T m;
  function new(string name="");
   super.new(name);
  endfunction
endclass

interface bus_intf #( int aw, int dw, int tp = 8,
                      strinf proc="ppc") (
  //port declarations …
);

module top
  parameter T1=10;
  parameter T2=T2*2;
  parameter T3=T3*5;

  bus_intf #( .aw (8), dw (12), .tp (T2), .proc("arm") )
            intf_arm(…);
  bus_intf #( .aw (16), dw (32), .tp (T3) .proc("nios") )
            intf_nios(…);
```

```
//One must define unique and correct types
typedef virtual bus_intf
  #( .aw (8), dw (12), .tp (T2), .proc("arm") )
     vif_arm_t;
typedef virtual bus_intf
  #( .aw (16), dw (32), .tp (T3), .proc("nios") )
     vif_nios_t;

// If using a config object method then one can
   do the following
vif_wrapper #(vif_arm_t) vif_arm;
vif_wrapper #(vif_nios_t) vif_nios;

initial
  begin
      vif_arm=new();
      vif_nios=new();
      vif_arm.m = intf_arm;  //Pass the actual
                             handle interface
      vif_nios.m = intf_nios;  //Pass the actual
                             handle interface
      set_config_object("*","wrapper",wrapper,0);
 run_test("test");
  end
endmodule
```

The issue is compounded if we want to define a generic agent based on the parameterized interface. Each agent will be instantiated to communicate with two similar but differently parameterized interfaces. As you see above the interface specialization has to be inside the top due to the dependencies on the top-level parameters. In such an agent, what is the virtual interface handle inside the driver going to look like? It has to be specific to the specialized-type of interface it will be communicating through.

Here's the pseudo code:

```
class driver #(type T = int) extends ovm_driver#(T);
  `ovm_component_param_utils(driver #(REQ))
  transaction item;
  virtual intf (…) vif;
endclass
```

POLYMORPHIC INTERFACE:

The Polymorphic Interface approach provides a very elegant solution for sharing parameterized interfaces. This approach is also known as "Abstract/Concrete Class" approach or "Two Kingdoms" approach. This approach involves defining a pure virtual (i.e. "abstract") class and then making it concrete inside the scope of the SystemVerilog interface (or module). The abstract class in essence provides the "interface" like construct in Java, and each concrete class inside the SystemVerilog interface (or module) provides functionality for the abstract functions in the context of the SystemVerilog interface (or module).

One could have multiple levels of inheritance hierarchy each attached to a different type of SystemVerilog Interface (or module), The OVM testbench will use the abstract class handle and, via polymorphism, the function calls get forwarded to the concrete class functions inside the right interface - hence the name "polymorphic interface." As we have seen, the virtual interface is not polymorphic. In the example below, the polymorphic interface approach elegantly solves the issue with parameterized interfaces.

CREATING POLYMORPHIC INTERFACE:

Following are the steps involved in creating a Polymorphic Interface:

Step 1: Create an **abstract or virtual class**:

```
virtual class abstract_c extends ovm_object;
  function new(string name);
    super.new(name);
  endfunction
  pure virtual task wr_cycle(int addr, int data);
  pure virtual task rd_cycle(int addr, ref int data);
endclass
package abs_pkg;
  `include "ovm_macros.svh"
  import ovm_pkg::*;
  `include "abstract_c.svh"
endpackage
```

Step 2: Define interface (or module) that will contain the following things apart from regular port/parameter/modport/ task/function declarations:

- **a concrete class** that derives from the abstract class above
- a concrete class handle
- a utility function to create the concrete class instance

```
interface bus_intf #( int aw=8, int dw=8, int tp=8,
                      string proc="ppc") ();
import abs_pkg::*;
class concrete_c extends abstract_c;
  function new(string name="");
    super.new(name);
  endfunction
  task wr_cycle(int addr, int data);
    $display("wr_cycle: accessing %s interface
              pins", proc);
  endtask
  task rd_cycle(int addr, ref int data);
    $display("rd_cycle: accessing %s interface
              pins", proc);
  endtask
endclass

concrete_c concrete_inst;

function abstract_c get_concrete_c_inst();
  concrete_inst=new(proc);
  return concrete_inst;
endfunction
endinterface
```

Step 3: Use the interface in the top level testbench module to connect to the DUT just like you would normally use. We then add one extra step to **create the concrete class** (and add it as a config object) using the utility function we created earlier inside the interface as shown below:

```
import ovm_pkg::*;
import test_pkg::*;
module top;
  parameter T1=10;
  parameter T2=T1*2;
  parameter T3=T1*5;

  bus_intf #( .aw (8), .dw(12), .tp(T2), .proc("arm") )
intf_arm();
  bus_intf #( .aw (16), .dw(32), .tp(T3), .proc("nios") )
intf_nios();


  initial
    begin
      set_config_object("*","intf_arm",
        intf_arm.get_concrete_c_inst(),0);
      set_config_object("*","intf_nios",
        intf_nios.get_concrete_c_inst(),0);
      run_test("test");
    end
endmodule
```

Step 4: In the OVM testbench the **driver will call utility functions** (like wr_cycle) **provided in concrete_c via abstract_c** (polymorphism) to drive the DUT pins:

- declare a handle to the abstract class
- get the configuration object to assign the concrete object to the abstract handle
- inside the run function call the utility functions (such as wr_cycle)

```
class driver #(type T = int, string proc="ppc") extends
ovm_driver#(T);
  `ovm_component_param_utils(driver #(REQ,proc))
  ovm_sequence_item item;
  abstract_c api;

  function new(string name, ovm_component p);
    super.new(name, p);
  endfunction
```

```
  function void build();
    ovm_object tmp;
    super.build();

    if(!get_config_object({"intf_", proc}, tmp, 0)) begin
      ovm_report_error("build", {proc, " interface
                              not found"});
    end
    else begin
      if(!$cast(api, tmp)) begin
        ovm_report_error("build", {proc, " interface
                                incorrect"});
      end
    end
    endfunction

    task run();
      api.wr_cycle('hA1, 'h5C);
    endtask

endclass
```

Step 5: There are **no changes in how one creates the OVM Environment and Test**. One can create the agent with sequencer, driver and monitor. For the driver, specifically, use the one created earlier in step 4 that leverages the polymorphic interface. The agent's driver will now be customized with a different parameter which then customizes the underlying interface. Here's the example:

```
package comp_pkg;
  `include "ovm_macros.svh"
  import ovm_pkg::*;
  import abs_pkg::*;
   `include "driver.svh"
  class agent #(string proc="ppc") extends ovm_agent;
      `ovm_component_param_utils(agent#(proc))
     driver #(ovm_sequence_item, proc) drv;
     ovm_sequencer #(ovm_sequence_item) sqr;

    function new(string name, ovm_component p);
        super.new(name, p);
    endfunction

    function void build();
        super.build();
        drv = driver#(ovm_sequence_item, proc)
                        ::type_id::create("drv", this);
        sqr = new("sqr", this);
    endfunction

    function void connect();
        drv.seq_item_port.connect(sqr.seq_item_export);
        drv.rsp_port.connect(sqr.rsp_export);
    endfunction
  endclass

  class env extends ovm_env;
    `ovm_component_utils(env)
    agent #("arm") arm_agent;
    agent #("nios") nios_agent;
    function new(string name, ovm_component p);
        super.new(name, p);
    endfunction

    function void build();
        super.build();
        arm_agent = agent#("arm")::type_id::create
                        ("arm_agent", this);
        nios_agent = agent#("nios")::type_id::create
                        ("nios_agent", this);
    endfunction

endclass
endpackage


package test_pkg;
    import ovm_pkg::*;
    import comp_pkg::*;
      `include "ovm_macros.svh"
    class test extends ovm_test;
      `ovm_component_utils(test)
     env e;
```

```
    function new(string name, ovm_component p);
        super.new(name, p);
    endfunction

    function void build();
        super.build();
        e = env::type_id::create("env", this);
    endfunction
  endclass
endpackage
```

CONCLUSION:

Virtual Interfaces themselves do not have a notion of polymorphism. The virtual interface is the standard documented mechanism for communication between dynamic (OVM/UVM) testbench and the static design under test. In OVM, the preferred interface-based approach is to wrap the virtual interface in an object and save the wrapper into the OVM/UVM configuration database. In UVM, the virtual interface can be passed into the uvm_config_db directly without a wrapper class.

As shown in this paper, there are situations, such as parameterized interfaces, when virtual interface approach doesn't work due to limitations in the interface syntax itself. The Polymorphic Interface (Abstract/concrete class) approach has the following advantages over virtual interface:

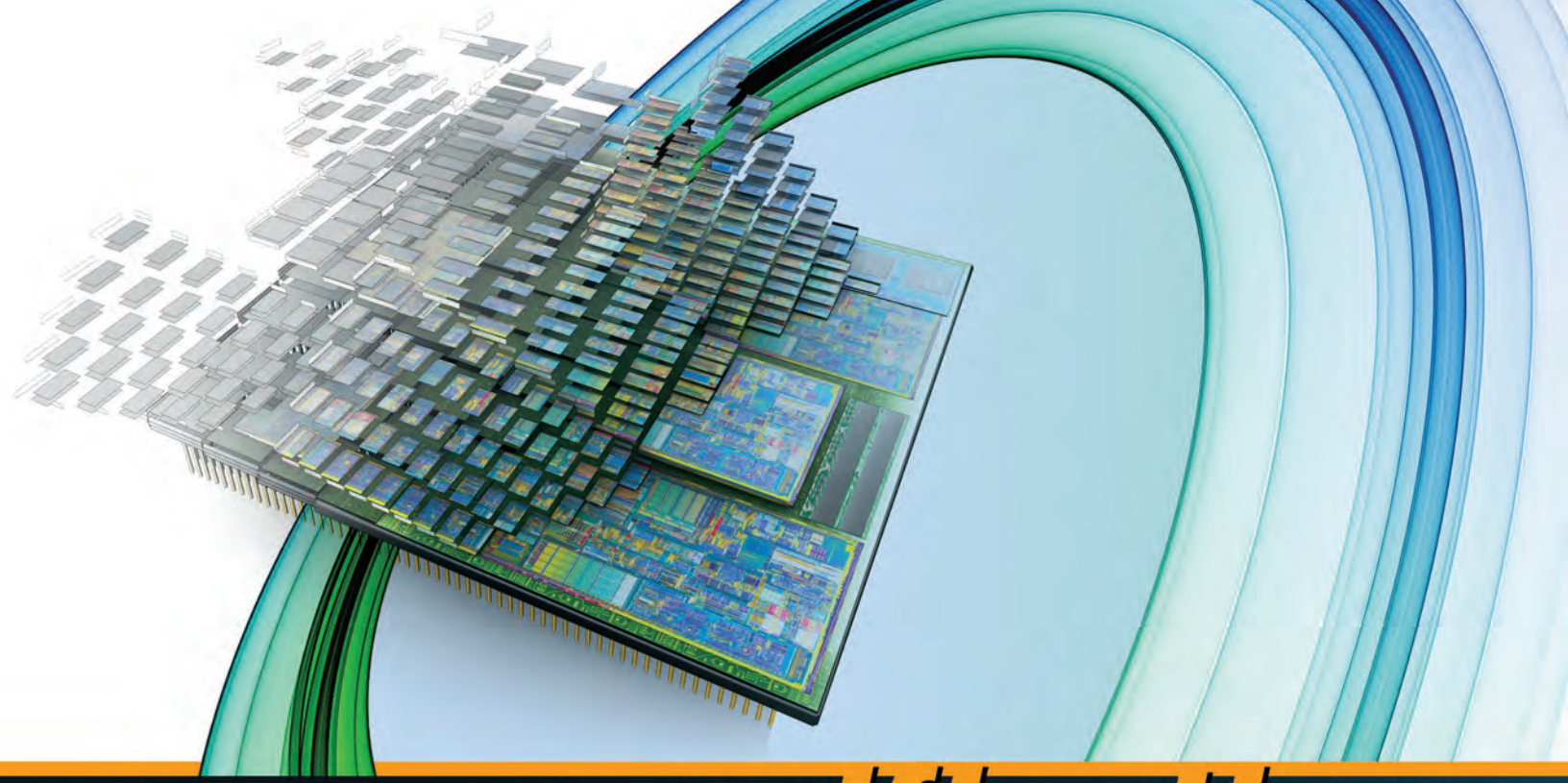
- It provides a cleaner mechanism to deal with parameterized interfaces, as shown here.
- It also provides polymorphism with interfaces, hence the name Polymorphic Interface.
- It gets rid of the extra virtual interface wrapper completely, when using the preferred the configuration database, since the approach uses SystemVerilog class syntax. The concrete class can directly be made available via OVM/UVM configuration database to the driver/monitor or any other transactors as needed.

REFERENCE:

[1] Rich D., Bromley, J. Abstract BFMs Outshine Virtual Interfaces for SystemVerilog Testbenches. DVCon 2008.

[2] Baird, M Coverage Driven Verification of an Unmodified DUT within an OVM Testbench. DVCon 2010.

[3] Bromley J. First Reports from the UVM Trenches: User-friendly, Versatile and Malleable, or Just the Emperor's New Methodology? DVCon 2011



verification HORIZONS

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Mentor Graphics®

www.mentor.com