Verification Prowess with the UVM Harness

Interface Techniques for Advanced Verification Strategies

Jeff Vance, Jeff Montesano,
Kevin Johnston

Verilab Inc.
Austin, Texas

www.verilab.com

ABSTRACT

In this paper we show how to create a UVM testbench with interface connections that universally work in any design simulation context. A harness is a common solution for encapsulating interfaces, binding them to the DUT, and publishing virtual interface assignments. We show how to enhance the harness with interfaces that work with both master and slave agents, in active and passive modes, with active RTL or stub modules, and can tolerate changes to design hierarchy. We accomplish this using interfaces with standard SystemVerilog features of binding and port coercion. Examples demonstrate how we can now encapsulate methods that access internal signals, change UVM agent roles between tests, and dynamically inject stimulus to any portion of a design without impact to how we connect and use interfaces from testbench components. This also allows us to efficiently run tests that verify different portions of a design using a single compile.

Table of Contents

Table of Figures

1. Introduction

1.1 Problems with DUT Connections

Verification of any design requires connecting a testbench for driving stimulus and monitoring outputs. However, the traditional ways of connecting a testbench to the Design Under Test (DUT) in SystemVerilog are burdensome and create major restrictions on the verification strategies we can apply. The most common issues are the following:

- Large design port lists require maintaining thousands of hardcoded signal assignments to the DUT instance. These signals have many separate concerns yet they are often all managed in a single place. This lack of encapsulation creates maintenance complexity.

- Connections are often tied to a specific DUT hierarchy. If the hierarchy changes with more layers of modules, or if instance names change, the connections have to be updated.

- When connecting to many instances of a design module, we must manually instantiate and maintain connections for all instances independently. Additionally, parameterized design modules often lead to maintaining parameterized interfaces to accommodate different signal widths of each instance.

- When verifying master and slave devices of a standard protocol, we must drive signals in opposite directions. This often leads to defining a master interface and slave interface separately, each connected with directional assign statements. This doubles the effort compared to using a single interface for the identical port list of masters and slaves. Directional assignments also hardcode these connections to a fixed direction.

- We often need to stub out design blocks, removing RTL drivers of signals in order to inject testbench stimulus to an internal interface. However, resolving which source drives the signals often requires compiler directives and logic to route interface signals.

- Verifying design blocks at different layers of integration requires maintaining separate testbench environments. Virtual interface assignments to the testbench often rely on hard-coded configuration database assignments, requiring maintenance between different environments.

In this paper, we show how to solve all of the above problems using enhanced applications of the UVM Harness. The UVM Harness has become common industry practice for creating well encapsulated DUT connections that scale to any number of module instances and is reusable between environments that change hierarchy. However, we can take the harness solution further and apply advanced verification strategies that would otherwise not be achievable in a typical project schedule.

1.2 Solution Summary

The first part of this paper presents solutions that can be applied to any existing testbench environment. We give an overview of the UVM Harness with port-based interfaces and how this is different from other ways of connecting to a design. We then show how to enhance the harness to

support bidirectional signals with port coercion and allow a single interface type to work with parameterized signal widths. We demonstrate how this solution is much simpler and more flexible than other ways of connecting interfaces while providing better encapsulation. We then show additional benefits that come from the harness such as allowing testbench code to directly access RTL parameters of design modules, changing master and slave roles of agents, and working with stub modules. Finally we show how to make the harness polymorphic so we can encapsulate methods that require access to internal design signals and make these features extendable.

2. UVM Harness Overview

2.1 Traditional DUT to Testbench Connection

The traditional way of connecting a DUT to a UVM verification environment is to declare one or more interfaces inside a testbench module and connect the DUT and interface(s) together (Figure 1).

```
module tb;
  ...
  if_type dut_if();
  dut dut_inst (.clk(dut_if.clk),
                .rst(dut_if.rst),
                …);
```

Figure 1. Traditional connection between DUT and testbench interface.

The connection between interface and verification component (VC) is done using the UVM configuration database (Figure 2, Figure 3).

```
module tb;
  ...
  initial begin
    uvm_config_db#(if_type)::set(null, "*.dut_driver", "dut_vif",
       dut_if);
...
endmodule

class dut_driver extends uvm_driver;
  virtual if_type vif;
  function build_phase(...);
    uvm_config_db#(virtual if_type)::get(this, "", "dut_vif", vif);
  ...
Endclass
```

Figure 2. Connection between VC and testbench interface.

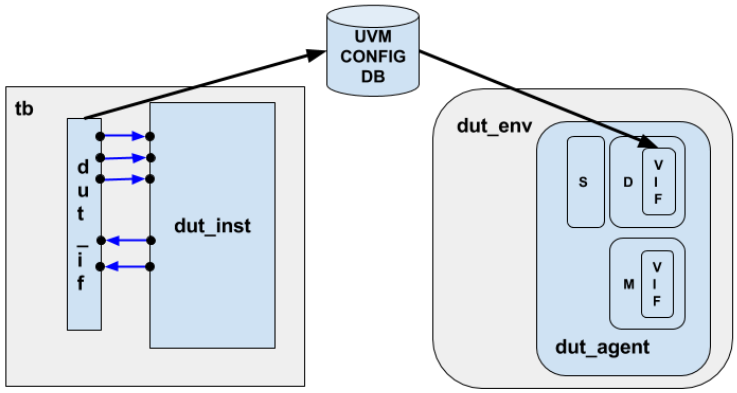

Figure 3. Traditional Interface Handling in UVM.

While this method has been considered adequate for a while now, it is less than ideal in a few ways:

- All connections are defined in the testbench file that instantiates the DUT; it becomes hard to manage many different interfaces in a single file. For advanced verification strategies we often want to manage each interface independently.

- The connections are all hardcoded up-front. This means to change connections, you have to edit the file that manages all the DUT connections. For advanced verification strategies we will want to change connections for different environments and connect things at different module hierarchies.

- Sometimes projects generate lists of wires to connect to the DUT so that interface assignments can be done elsewhere. This creates the extra burden of maintaining large lists of wires and assign statements while limiting connections to a single direction.

The testbench developer needs to spend time re-compiling and debugging when there are changes to the DUT's ports or hierarchy. As outlined in [1], there is a better way to connect a DUT and a verification environment, that being the UVM Harness technique.

2.2 UVM Harness Connection Methodology

2.2.1 Using Interface Ports

The harness technique requires that we define all signals as net ports of the interface instead of variables internal to the interface (Figure 4). While this is slightly less conventional, we will show how this allows more flexibility in how we assign connections and manage directions. We can still define additional signals inside the interface (e.g. to be used with clocking blocks, variables that do not connect to the DUT, etc.), but the connections to the DUT will always be made to the interface ports. Note that this will not work with ports declared as type logic since this makes them variable ports rather than net ports. The ports must be declared as wires (or just declare them as inputs which defaults to wire).

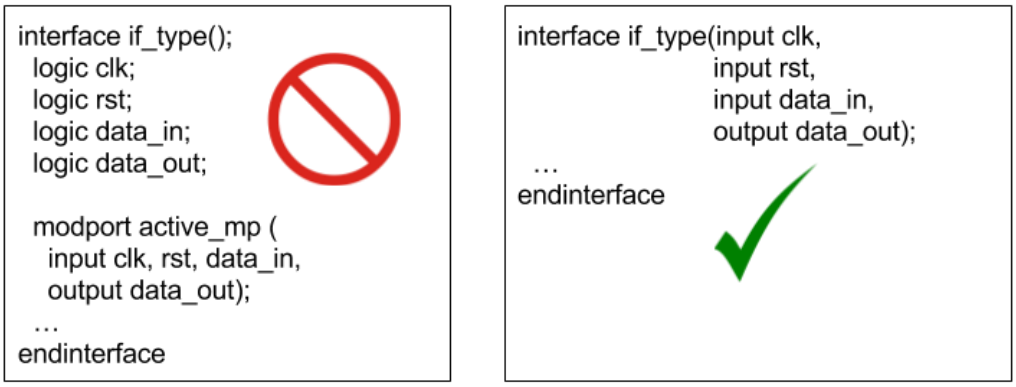

Figure 4. UVM Harness methodology requires interfaces to use ports.[1]

2.2.2 Interface to DUT Connection

Before reviewing the full details of this technique, let's first examine how it makes use of the SystemVerilog bind directive to place an interface inside the module of a DUT rather than outside the DUT module. The SystemVerilog bind directive will amend the definition of a module so that all instances of that module are altered. The UVM Harness technique uses this amending capability to redefine a DUT module to instantiate an interface inside of it (Figure 5), thereby eliminating the need for the testbench module to instantiate any interfaces. While it is also possible to bind to specific instance names, in this paper we strictly bind by module name ensuring every instance of the target module gets an interface. The syntax for this is `bind <module_type> <interface_type> <interface_name>`.

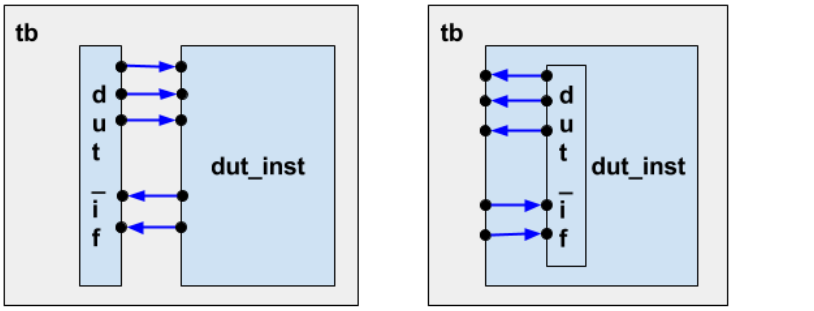

Figure 5. Traditional vs. UVM Harness interface placement.

Should the design hierarchy change during the project, the testbench will not need to be updated, since the bind directive amends the module definition regardless of where it is instantiated (Figure 6).

[1] In section 3.2 we will make the case for why specifying directions on ports, as shown in Figure 4, is not recommended.

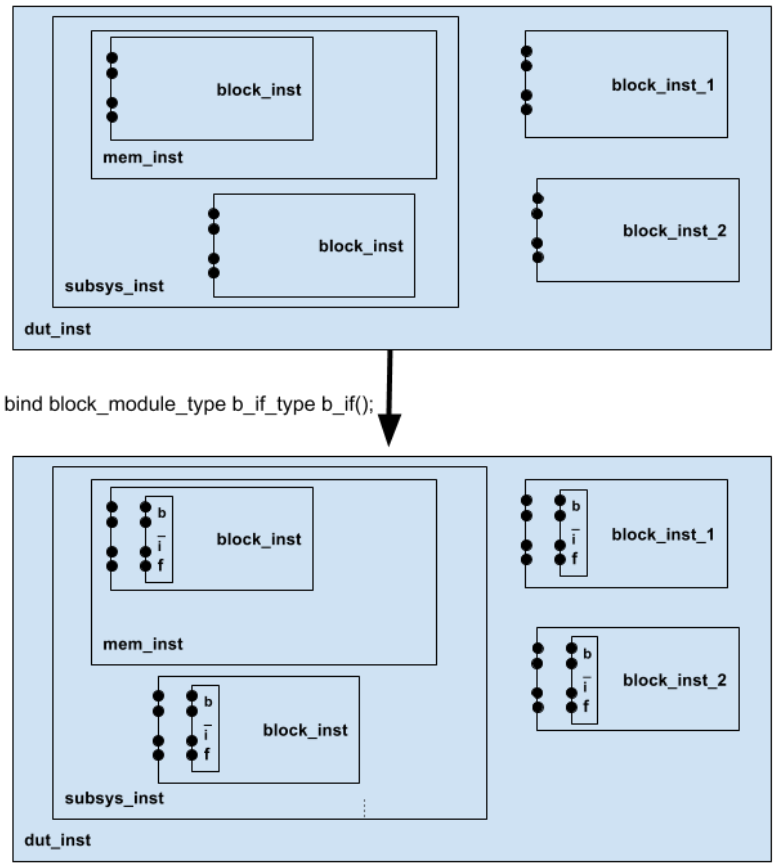

Figure 6. Bind directive amends module definition and thereby affects all module instances.

Having moved the interface instantiation to inside the module definition, the UVM Harness technique then encapsulates the connections between DUT and testbench. This is done as follows:

1. Define an interface[2] with all signals declared as ports, with each interface port corresponding to the name of a DUT port (henceforth called the **DUT interface**)
2. Define a wrapper interface (henceforth called a **harness**)
3. Make the harness instantiate the **DUT interface**
4. Connect the **DUT interface**'s ports to the DUT using an upward reference to the *module name* (as opposed to the module *instance name* – see below)
5. Bind the **harness** into the DUT

The SystemVerilog LRM defines how hierarchical references are resolved (see [2], Sections 23.6 – 23.8). Downward hierarchical references are what most people are used to: we can reference down a hierarchcal path to an instance name of any module below the current scope. However, a harness requires upward referencing to the module it is inside of, because we want the **DUT interface** to

[2] In the interest of clarity, we are limiting ourselves to one interface per harness in this discussion. However, a harness can define, instantiate, and connect multiple interfaces should a given DUT require it.

reference the DUT's ports in order to make a connection between them (Figure 7). Note that according to [2] upward references are always to a *module name*, while downward references are always to an *instance name*.

```
interface dut_harness(); // "harness"
  // instantiate the "DUT interface" and connect to the DUT ports
  // The bind target module name is "dut"

  dut_if_type dut_if(.if_clk(dut.clk),
                     .if_rst(dut.rst),
                     .if_data_in(dut.data_in),
                     .if_data_out(dut.data_out)
                     …);
  ...
endinterface

// bind the "harness" into the DUT
// bind <module_type> <interface_type> <interface_name>";
bind dut dut_harness harness;
```

Figure 7. Harness instantiates DUT interface and connects it to module ports using upward referencing.

Note that while a bind directive can be specified in any module or interface, because our usage of it in this technique is independent of the DUT hierarchy, we like to place it in the same file as the harness, immediately after the harness definition (Figure 7).

2.2.3 Interface to Agents Connection

Next we'll talk about how the UVM Harness technique makes the interfaces bound into the DUT available to VCs. As in the traditional approach, the UVM configuration database is used to publish interfaces from the module-based world to the class-based world. What's different is that the UVM Harness technique encapsulates the call to `uvm_config_db::set()` inside the harness itself[3] (Figure 8).

```
interface dut_harness();
  dut_if_type dut_if(.if_clk(dut_clk),
                             .if_rst(dut_rst),
                             …);
  function void set_vif(string path);
    uvm_config_db#(dut_if_type)::set(null, path, "dut_vif", dut_if);
  endfunction
endinterface
```

Figure 8. Interface connection to VC encapsulated in the harness.

[3] As described in [1] there are several ways of doing this; here we are only discussing the one we found most suitable in our own work.

The encapsulation also involves a function call, which gives the testbench module the ability to set the path within the verification environment to where the interface will be published. This function allows for multiple instances of a harness (e.g. to connect to several instances of the same protocol block on a DUT) without any naming conflicts in the configuration database (Figure 9, Figure 10).

```
module tb;
  dut dut_inst();
  …
  initial begin
      dut_inst.b_inst_1.harness.set_vif("*.env.b_agent_1.driver");
      dut_inst.b_inst_1.harness.set_vif("*.env.b_agent_1.monitor");
      ...
      dut_inst.b_inst_n.harness.set_vif("*.env.b_agent_n.driver");
      dut_inst.b_inst_n.harness.set_vif("*.env.b_agent_n.monitor");
  end
endmodule

class b_driver extends uvm_driver;
  virtual if_type vif;
  function build_phase(...);
    uvm_config_db#(virtual b_if_type)::get(this, "", "b_vif", vif);
    …
endclass
```

Figure 9. Use of function to publish the different interfaces to different VCs.

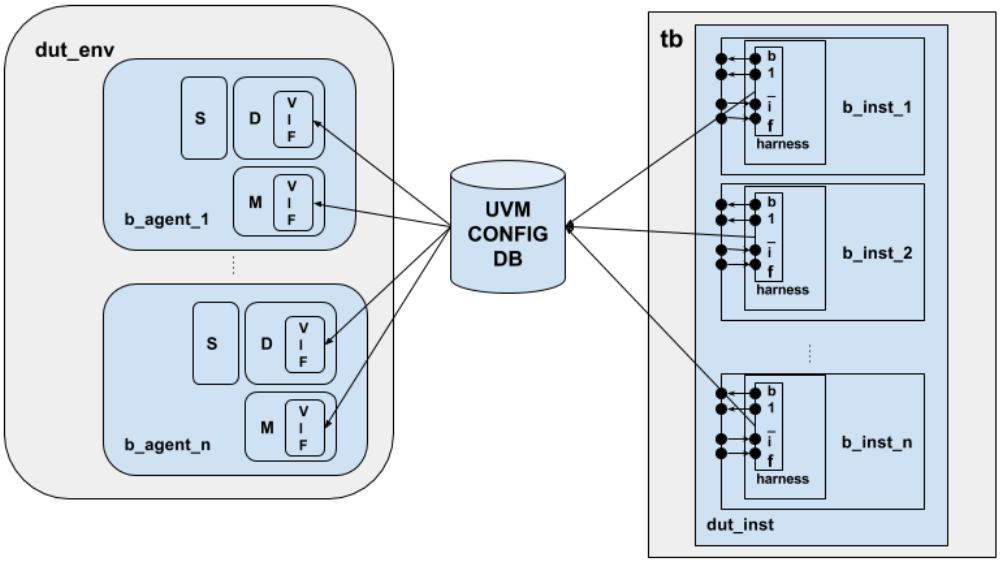

Figure 10. Multiple DUT interfaces connecting to multiple VCs via the UVM configuration database.

An additional side-benefit of the using the UVM harness technique is that the testbench module file becomes nothing more than an instantiation of the DUT and some calls to the set_vifs function of the harness. You don't need to connect any DUT ports when you instantiate it and you don't need any additional signal declarations in the top module. This is a big improvement over the long and

cluttered files many of us have become accustomed to.

3. Enhancing the Harness

The basic harness we've shown so far solves a few common problems. We can now independently manage interface connections, reuse connections between testbench environments, and tolerate changes in design hierarchy. In this section we show how to further enhance the harness to allow for more advanced verification techniques. This will allow interface connections to work regardless of different signal widths, signal directions, and working with either active RTL drivers or stubs.

3.1 Variable Width Support

Very often we have to support signals of different widths connected to the same type of interface. For example, an interface for a standard protocol may use different address widths for some agents. Some may transmit 32-bit data and others 64-bit data. Our solution applies a "max-footprint" approach to handle this. A single interface is defined to support the maximum possible width for all signals. Any instance that requires fewer bits connects only up to the width needed. All unused bits must be tied off with weak drivers. Figure 11 shows an example of how we can connect a max-footprint interface to a design module that may have a smaller signal width. We define parameters for the DUT signal width and maximum widths. We then use an if-generate statement to assign tie-offs if necessary.

```
interface dut_if_type(input bit wr_data[MAX_DATA_WIDTH-1:0],
                      …);
endinterface

interface dut_harness;
  dut_if_type dut_if(.wr_data(dut.wr_data[DUT_DATA_WIDTH-1:0]),
                      …);

  if(DUT_DATA_WIDTH < MAX_DATA_WIDTH) begin: unused_data_pullup
    assign (pull1,pull0) dut_if.wr_data[MAX_DATA_WIDTH-1:DUT_DATA_WIDTH]
      = '0;
  end
  ...
```

Figure 11. Using maximum bus width in interface definitions.

This keeps the interface definition generic and reusable for any combination of signal widths. This also means neither the interface nor the agents need to be parameterized to handle different widths. Agents are permitted to drive the full width of the interface even though some bits may not actually be connected. We generally want to avoid parameterizing things since it adds complexity, maintenance, and dependencies on compile-time options. Parameterizing signal widths would also significantly reduce the benefits of binding an interface to a module with multiple instances. The max-footprint approach allows a single bind statement to connect the same interface type to all instances

of a module, even if each instance of that module is parameterized with different signal widths.[4]

3.2 Support for Driving Signals in Both Directions

To make our interfaces as reusable as possible, directions should not be imposed on the signals in the DUT interface definition. Additionally, the way we connect these signals should not impose a direction. We want the interface to be generic and handle any situation. For example, while it is possible to define separate interfaces for master and slave agents, ideally both agents will use the same interface. As another example, signals that are outputs for an active agent become inputs when an agent is passive. We don't want to reconfigure or change interfaces for every situation that changes signal directions.

The first solution many try is to declare all ports of the interface as **inout**. However, inout ports have a major restriction: SystemVerilog requires that an **inout** port be connected to a "collapsible net." This means you can only connect to a **wire** of equal width as the **inout** port. If you attempt to connect an inout port to a signal that is a different width, there is a compile-time error. Therefore using inouts prevents us from using the max-footprint interface approach described in Section 3.1 .

3.2.1 Port Coercion

We can solve these problems by declaring all interface ports as **input** and use **port coercion** to control the signal directions. Port coercion is a little-known but standard SystemVerilog feature that allows directional ports to be coerced to inout ports based on the direction of driving statements that are compiled (see[2], Section 23.3.3.1 and [3]). This is useful because we can allow signals to be driven in either direction while avoiding the restrictions of inout ports.

VCS performs port coercion of net ports (i.e., wire ports) during elaboration if there is any statement that drives the opposite direction of the port. This commonly happens in design modules that assign an internal wire to a port that is declared as an input. VCS also coerces ports of an interface if there is any code that drives it from a virtual interface reference. For example, if you define an interface with all input ports, pass the virtual interface to the testbench environment, but the testbench does not define any driver to the virtual interface, the ports will remain inputs. If you later define a driver class that drives the virtual interface ports, the compiler will then coerce them to inout.

VCS reports coerced ports in the compile log. By default, it only reports a brief statement that port coercion is occurring. Adding the "-notice" option will print details of every port that is coerced.

3.2.2 Resolving Drivers to Ports

Port coercion on **input** ports solves both the problem of changing signal directions and resolving drivers from each side of the port. If an agent is active, the driver will drive these ports as if they are outputs. If a port has no potential drivers from the testbench, it will remain an input port and will be driven from the design. The choice of which ports are coerced will be common to all tests that run on a single compile.

Be aware that we don't always know everything that will actually attempt to drive a port at compile time. For example, some instances of an interface may be connected to an active UVM agent and drive out the ports while other instances only have passive monitors. The compiler can't distinguish

[4] This may require parameterizing the width of tie-offs to correspond to the width of the signal.

between these cases since virtual interface assignments are made at runtime. Another example is different tests may configure the same agent to be active or passive. Again, the compiler can't determine this since building an active agent isn't decided until the UVM build phase. However, this is not a problem because the compiler will make choices that work out in every scenario. If it compiles a driver class that drives to the virtual interface of that type, it will assume that the port must be coerced for all instances, even though we might not actually build the driver at runtime. This ensures all instances of the interface will work properly.

Note that we can't get the benefits of port coercion with traditional interface connections of assign statements or connecting internal interface signals to DUT ports. These approaches will always hardcode directions permanently. This solution is also much simpler than other common techniques that rely on adding parameters, compiler directives, and logic to change directions and conditionally drive high-z to avoid multiple drivers.

3.3 Testbench Access to RTL Parameters

We often need a testbench to be configured according to RTL parameters being used in the design. This is challenging because design parameters are defined at compile time, however we usually design a testbench to be dynamically configurable at runtime. This is especially challenging in projects that randomize parameters to verify combinations of different design configurations. The most common solutions are not ideal, usually involving duplicating these parameters for the testbench.

A better approach was introduced in [4], which describes a mechanism for the testbench to directly extract the RTL parameters and make them available to all classes in the testbench. We can further enhance the UVM harness by including this technique. We can bind the harness to a parameterized module, define a function to extract the parameters, and store the values in a data structure that is accessible by testbench components. See [4] for detailed examples.

3.4 Working with Stub Modules

For performance reasons, it is common in large projects to compile with some design modules swapped with stub modules. A stub module has an identical port list and uses the same parameters as a design module but with no internal logic (except for possibly tieing off output signals). However, this often creates a problem since the presence of active RTL usually impacts how we connect the testbench interfaces. Additionally, there are often many different combinations of potential stubs in a design and each combination requires its own compilation. If various tests stress different design modules with different stubs, regressions become inefficient since it is impossible to run all tests without compiling again for each combination of stubs. You can compile many stub configurations in parallel, however this does not scale for verification that requires complete flexibility[5].

As previously described, port coercion solves the problem of handling active RTL drivers. We no longer need to use compiler directives to know if the there is RTL driving and change how things are connected. This gives us an added benefit of doing more advanced manipulations of the design we

[5] For example, running tests with any stub combination of 16 CPU cores would require compiling 65,536 times.

are verifying without impacting the testbench. For example, in the following sections we show how tests can change the roles of agents to be a master or slave, depending on the presence of a stub module.[6]

3.4.1 Changing Master & Slave Roles

Most testbenches define an agent as being either a master or slave agent. Master agents model a device that issues commands while slave agents model devices that wait and respond to commands. The configuration of these devices is defined in the design specification and is therefore usually a permanent configuration of the testbench. However, there are situations when allowing an agent to change roles has major advantages, such as when we use stub modules in a design. The UVM Harness technique (Section 2.2) combined with port coercion (Section 3.2) allows us to freely change agents to act as masters or slaves without any modifications to the interfaces or how they are connected.

Allowing agents to change master and slave roles can be accomplished two different ways. The first option is to implement both master and slave functionality in a single class definition, using a configuration bit in the agent to select the role at runtime. Based on this setting, the driver changes which signals to drive and the monitor changes how it monitors the interface. The second option is to have master and slave agents extend from a common base class[7] and use UVM factory overrides to use the correct agent type for each test.

3.4.2 Example 1: Block-level Tests in a System Testbench

The first example of this technique is the situation where we want to do block-level testing within a larger system or full-chip testbench. In this case, we want to continue running tests that we wrote for an individual design block, but now that block is integrated with other design blocks in a sub-system or top SoC testbench environment. Rather than running the block-level tests in a separate block-level testbench, we can run these tests in the system testbench. This has several benefits:

- It is more efficient to maintain a single system testbench than many individual testbench environments for each block or combination of blocks. Projects often need a top system testbench anyway. This solution gives us an environment for testing any block or clusters of blocks for free. This is very powerful considering the manual equivalent would be combinatorial effort of creating separate testbenches.

- This supports verifying multiple design versions that target different performance and price-points in the market. A design with 2-core, 4-core, and 8-core versions can all use the same testbench by simply stubbing the unused cores.

- Teams running block-level tests don't have to make assumptions about clocks, resets, and other system-level integrations. All block-level tests will have better clock supplies and a more accurate reset sequence.

---

[6] When using the max-width interface approach with stubs, we found that we have to define the stub signal width to match the max-width value (not the design's signal width) in order to avoid high-Z on on the driven port.

[7] Ideally we would use an abstract base class with a mandatory factory override by each test to use a master or slave agent. However, VCS uses strict compiler rules that prevent us from registering abstract classes with the UVM factory.

- Regressions and tracking coverage become centralized and easier to manage. Coverage results between system and sub-system tests are automatically merged.

- Bug isolation can be easier. A failing scenario in a system test can be recreated more easily by regenerating the scenario for an isolated set of sub-modules or different combinations of sub-modules to quickly identify the root cause.

To run tests on a single block within a system testbench environment, we need to stub out all design modules except the block being tested. This will achieve close to the same simulation performance as running a separate block-level testbench. Stubbing out all modules except the one being tested usually only works if we can change the roles of the agents connected to the ports of that block. Figure 12 shows a slave block (e_inst) integrated into a system testbench with four masters. These four masters are connected to the four master channels of the e_inst block. Therefore the harness bound to e_inst contains four master interfaces. The figure also shows four agents, all of which belong to the same environment class.

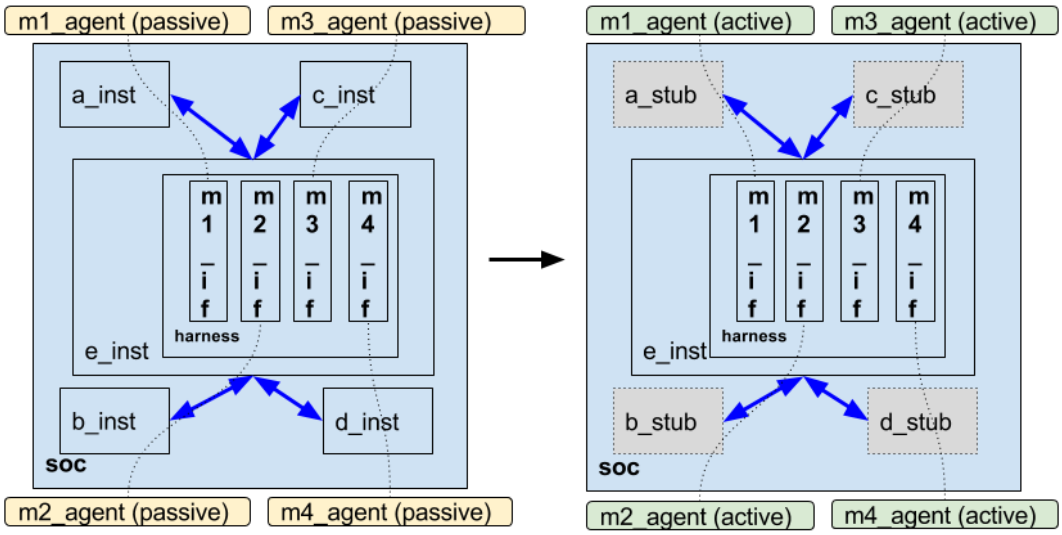

Figure 12. Stubbing surrounding modules to test an isolated module.

When running system tests, we rely on a connected design module (e.g., a_inst to provide the master stimulus to this block. The passive slave agent(s) connected to this block monitor all transactions according to the protocol rules of a slave. However, once we stub out the connected master, the testbench must provide master transactions to the block. We solve this problem by switching the block's passive slave agent to be an active master agent(s) (as shown in detail in Figure 13). Now the block-level tests can execute transactions on that agent the same as if it were a block-level testbench.

We can use the same solution for master ports of a block under test. In a system testbench, the block relies on connected design blocks to respond to requests. When running block-level tests, we stub out the connected slaves and switch the block's passive master agent to act as a reactive slave.

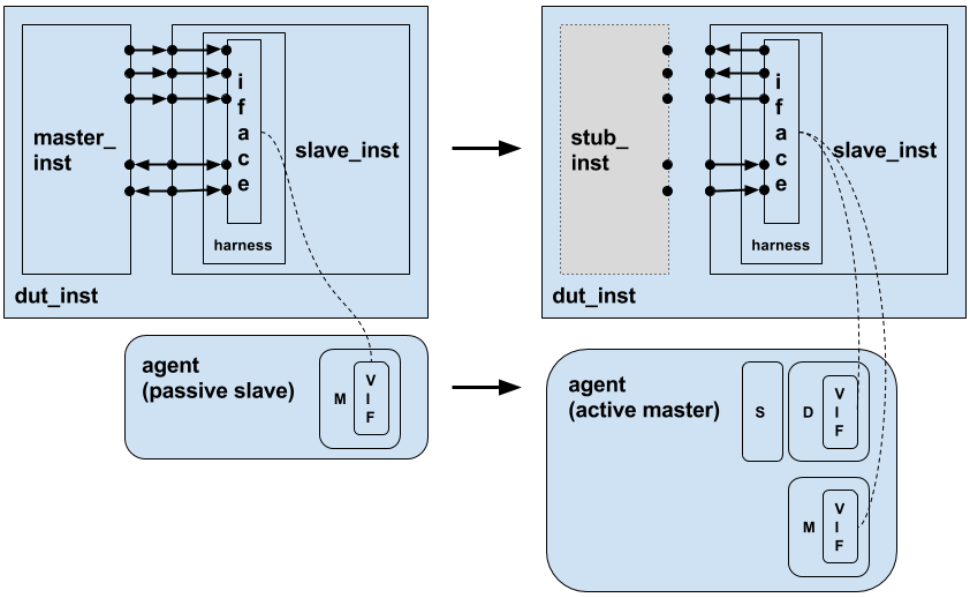

Figure 13. Isolating a slave module by stubbing masters and changing its agent to an active master.

The testbench environment needs configuration options to ensure all agents are in the correct role. In this example, we're assuming the environment class is associated with verifying a specific design module[8] and all agents in the environment are connected to ports of that module. Normally the agents connected to a block in a system testbench take on the behavior associated with that block. But when we stub out the surrounding blocks, the agents for that block change to take on the behavior of the surrounding system (Figure 12). From a testbench environment perspective, we can call this the "outside looking in" mode. The agents connected to the block represent the stimulus outside of that block, looking inward to test it. Figure 14 shows sample code for how the environment can configure each agent if a block-level test puts the system environment in this mode.

[8] Some testbenches and VIP create an environment class that contains agents for many different design modules located throughout the design, often sharing a common protocol. Our solution does not apply to this type of environment. For this reason, we recommend a methodology where each UVM environment only contains agents for a specific module under verification.

```
class soc_env extends uvm_env;
  function void build_phase (uvm_phase phase);
  super.build_phase(phase);

  if(cfg.env_role == OUTSIDE_LOOKING_IN) begin
    cfg.agent_role["a_agent"] = SLAVE;
    cfg.agent_role["b_agent"] = SLAVE;
    …
  end
  else begin
    cfg.agent_role["a_agent"] = MASTER;
    cfg.agent_role["b_agent"] = MASTER;
    …
endclass
```

Figure 14. Environment Configuration of Agent Roles.

3.4.3 Example 2: Stub-in Testbench Stimulus

Another technique now available to us is to replace a single design module with a stub in order to test the system connected to that block. A "stub-in" allows us to replace that design block with testbench stimulus that is applied outward to the connected design modules. In this configuration, we can apply constrained random stimulus directly to ports deep in the design hierarchy. We can call this environment configuration "inside looking out" mode and put corresponding configuration fields in the environment class for each test to control. Note that our motivation for using stubs in this situation is not always for performance. We can stub-in testbench stimulus as a verification strategy for creating hard-to-reach scenarios that would otherwise be too hard (or impossible) to reach from external stimulus.

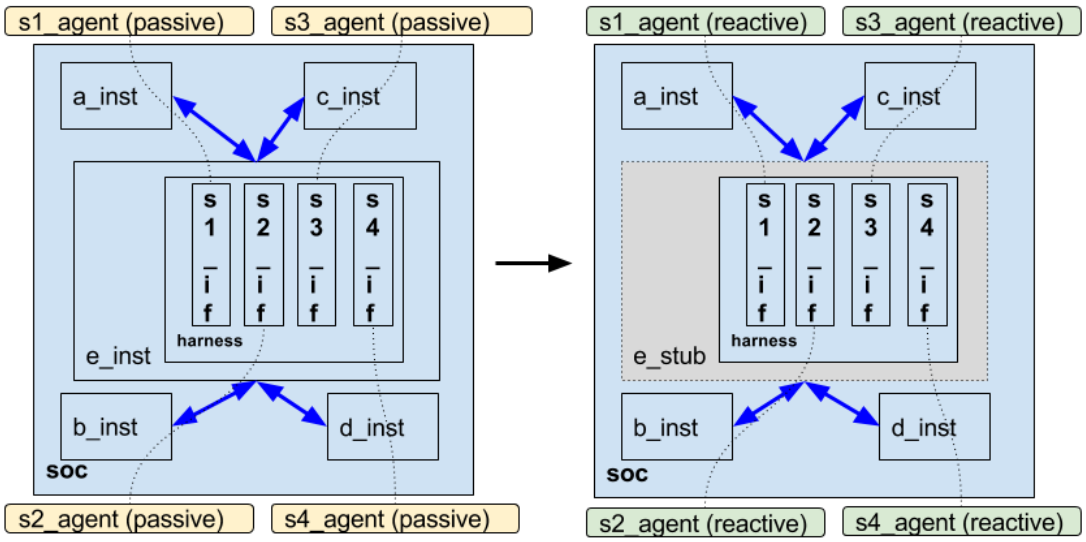

Figure 15. Stub an internal module to replace with testbench stimulus.

Unlike the previous example, this situation allows us to keep the agents of the block in the same master/slave role, but we must change them from passive to active. Passive master agents for the block will become active and drive transactions to the connected RTL slaves. The block's passive slave agents will become reactive slaves, responding to the requests of the connected RTL masters.

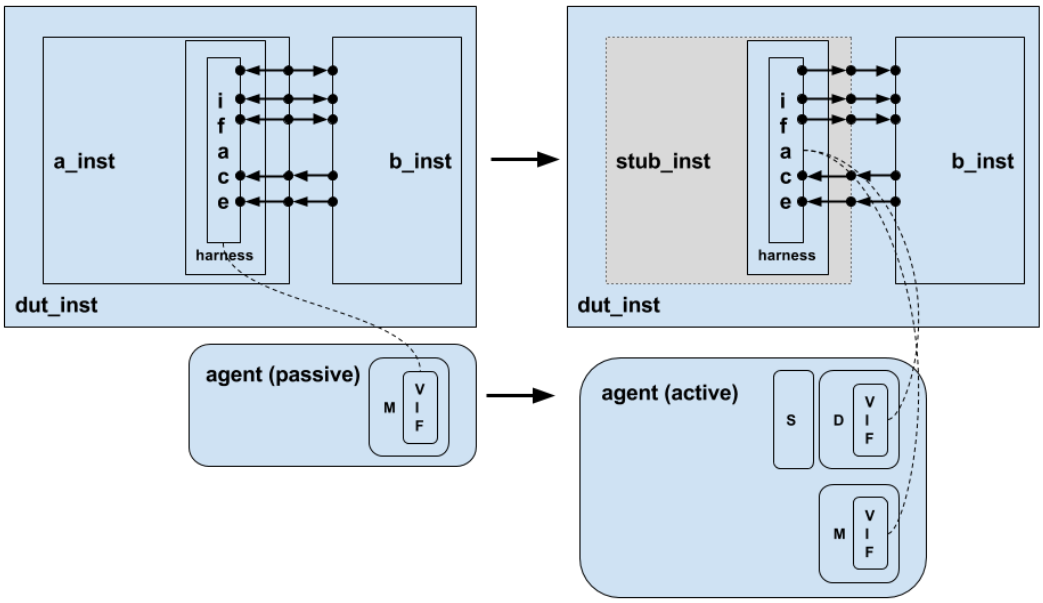

Figure 16. Stubbing an internal module, replacing it with testbench stimulus.

The harness interface connections with port-coercion allows us to do this since the compiler will never see multiple drivers on a net. Recall that we declared all interface ports as inputs so when we run with the real design module instantiated, it is the only thing driving the connections. With testbench stimulus stubbed-in, our testbench coerces the interface ports to inout to drive stimulus instead.

3.4.4 Example 3: Bus Interconnects

The third example we'll consider is how to configure agents for a bus interconnect. An interconnect interfaces with many design blocks, both masters and slaves. We can consider the interconnect itself to be a design block and apply the same techniques as the previous two examples. We can stub all the master and slave blocks connected to the interconnect and test it in isolation. We can also stub the interconnect, replacing it with testbench stimulus that acts like the interconnect. However, the fact that it interfaces with many design blocks adds more options for how to configure the agents connected to all of these blocks.

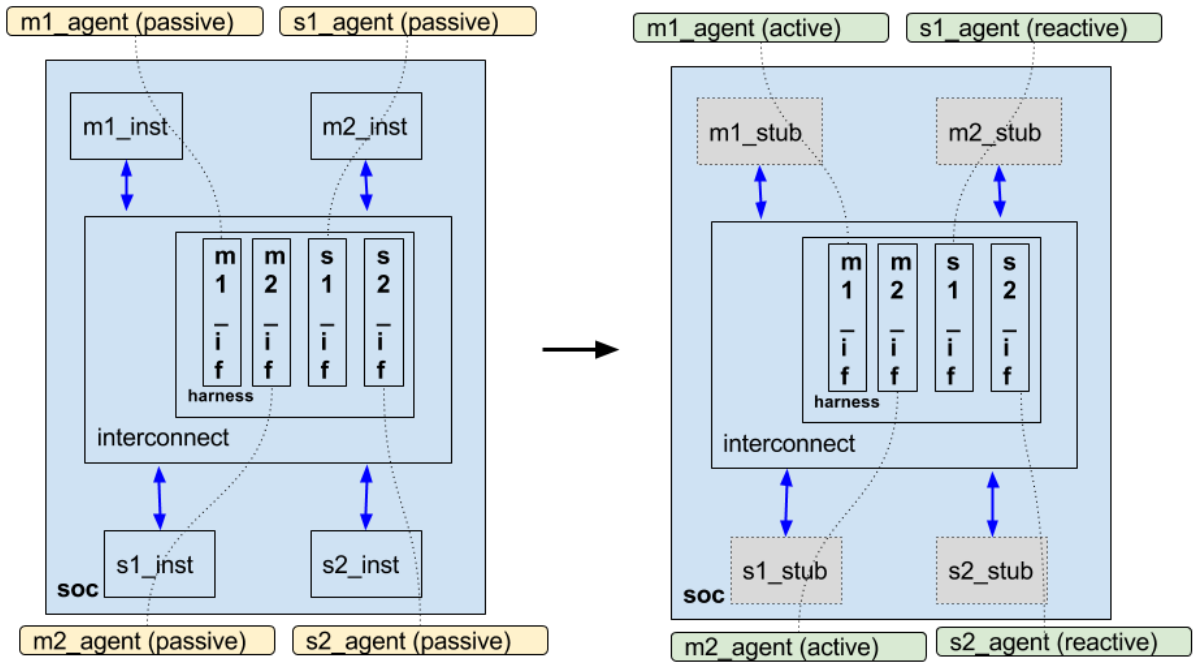

Figure 17. Interconnect going from all RTL modules to stubbed-out masters and slaves.

The left side of Figure 17 shows an interconnect with several master and slave devices. Each port of the interconnect has an agent and each connected block has its own agent. This raises the question: which agents should we use if we stub all blocks except the interconnect? There are two options:

- **Option 1:** use the approach from Example 1 (3.4.2) where each agent connected directly to the interconnect would change to the "outside-looking-in" mode (right side of Figure 17). The passive slave agents would become active masters and the passive master agents would become reactive slaves.

- **Option 2:** Use the agents of all the *other* blocks that were stubbed out. In this case, the agents of the interconnect don't change. The agents of connected master blocks stay in a "master role", but are configured as active to drive stimulus. Agents of slave blocks become reactive slaves.

A similar choice occurs when we stub the interconnect itself. Again, we have two options:

- **Option 1:** Use the approach from Example 2 (3.4.3) where we switch the interconnect's agents to "inside looking out" mode. The agents of the interconnect will drive the stimulus to all of the surrounding blocks.

- **Option 2:** Use all the agents of the *surrounding* blocks to take on this responsibility.

Both options are valid, however note that option 2 may require changing configurations of agents that are in many different environments. Option 1 may be easier to manage since all the agents to be reconfigured exist in a single environment class associated with the interconnect module.

3.4.5 Binding to Stub Modules

One challenge that comes with using stub modules is that we may need to continue using a harness that is connected that module's signals. However, we bind a harness using the module name and now that module is being replaced with a stub that may have a *different* module name. This will fail compilation if we continue using the harness. There are several potential ways of dealing with this problem, but unfortunately none are perfectly ideal:

- Ensure stubs modules always have the same name as the original module and use different library mappings to ensure the correct one is used. See [2] Section 33.8 for more details. However the LRM says using a library mapping forces us to bind by instance name.

- Use stubs with different module names and create two versions of the harness: a stub-specific harness that is only bound to the stub module and an RTL-specific harness that is bound to the design module. Both harnesses are identical except for the module name used for the upward references. Publishing virtual interface connections is the same so the difference is transparent to the testbench.

- Use stubs with different module names and use compiler define statements for the module name to bind to and use for upward references in the harness.

- Use an if-generate statement to choose between two bind statements. The first binds a stub-specific harness to a stub and the second binds an RTL-specific harness to the design module. However this must be done inside a module such as the top testbench module.

4. Dynamically Swapping RTL and Testbench Stimulus

In the previous section, we showed how we can simulate different combinations of stubs in the same testbench without impacting the interface connections. However, this normally requires recompiling for each new combination of stubs we want to simulate. This section shows an alternative solution that allows us to enable testbench stimulus anywhere in the design without recompiling. This approach also lets us dynamically switch between RTL stimulus and testbench stimulus within a single test execution.

This solution does not use stubs and instead compiles the entire system to be used for all tests. Any driver that is enabled for testbench stimulus may be connected to the same signals being driven by active RTL. In order to resolve multiple drivers, we have to use force statements in the interface to override the internal RTL stimulus. Figure 18 shows how we can add force statements inside a harness. These force statements are only activated when the agent is active and enabled to drive forced values. We can put a **`force_en`** control bit in the DUT interface to control when forcing is enabled from each agent. In the next section, we show how we can't put the control bit directly in the harness unless we take extra steps to provide an API object to pass to agents.

```
interface dut_harness();
  dut_if_type dut_if(.if_data_in(dut.data_in),
                     .if_data_out(dut.data_out)
                     …);

  always@(*) begin
    if(dut_if.force_en) begin  //put force_en ctrl bit in interface
      force dut.data_in = dut_if.if_data_in;
      …
    end
    else begin
      release dut.data_in;
      …
    end
  end
endinterface
```

Figure 18. Use force statements in the harness to override active RTL stimulus.

The main advantage of this approach is even without using stubs, we can still activate testbench stimulus from any agent in a design without conflict of active RTL being present. A single compile can still randomly generate stimulus throughout a large system from any point. Since the forced values are only applied when the agent is enabled to drive, we can be sure that no agent interferes with internal signals when we aren't injecting stimulus.

The second advantage is we can now temporarily inject testbench stimulus in the middle of a test, then disable the driver and restore the normal RTL stimulus. This can be useful for generating scenarios that would otherwise be hard to reach. For example, we can wait for a design block to complete initialization traffic, then enable the block's agent to inject test stimulus instead of the default design behavior. Another example is we could temporarily inject transactions that put the design in an anomolous state, then disable the driver to verify if the system can recover to a normal state.

5. Encapsulating Methods in the Harness

We can take advantage of the encapsulation provided by a harness by putting additional features inside that are associated with that interface. For example:

- Some projects require direct access to internal signals to preload memories from files.
- Some tests may need to force internal signals to inject errors.
- It is common to put assertions in an interface for relationships of the signals specific to that instance. However, now we can put assertions in a harness to assert relationships between signals in different interfaces that are bound to the same scope.

As an initial attempt to implement this idea, we could consider passing a virtual interface reference to the harness itself. This is in addition to the configuration database publishing of the internal interfaces connected to the DUT. Conceptually, this would allow any component to get a direct reference to the harness and access the functions we define. However, it turns out in most cases

this is illegal. The SystemVerilog LRM says it is illegal to pass a virtual interface if that interface makes a reference outside of it (See [2], Section 25.9). The most useful methods we could define in a harness are ones that will reference signals inside the design, but doing this prevents us from being able to pass a reference to access these functions.

The solution to this problem is to define a class inside the harness that contains an API to these methods. The API object contains wrapper methods to the harness methods. Testbench components can call the API methods through a handle to a harness API. However, we need several extra steps to make this work. A class defined inside an interface is not known outside the scope of that interface. Therefore no testbench code can directly declare a handle of this class type. The solution is to apply the Polymorphic Interface technique (introduced in [5]) to the harness. Figure 19 shows an example. We accomplish this with the following steps:

1) Define tasks and functions that access design signals directly inside the harness. These methods must make an upward reference to the module name the harness is bound to, similar to how we made connections. These can reference any sub-module below it with a hierarchical reference. (Remember, from Section 2.2.2 , the initial upward reference must be a module name, but the continuing downward path must use instance names).

2) Define an abstract base class for the harness API. This class has a pure virtual method (no implemented body) that corresponds to each of the methods in the harness. Also, this abstract class must be defined inside a package so we can import it into the scope of the harness.

3) Import the abstract class package inside the harness definition. This will allow code inside the harness to reference this class type.

4) Create a nested class definition inside the harness that extends the abstract class defined in Step 2. Define the implementation of all the pure virtual methods declared in the base class. Define these to simply call the associated methods in the harness, passing any arguments. Since the class definition is inside the harness, all methods in the class can directly access harness methods. This allows users to indirectly invoke the harness methods that reference design modules.

5) Create an instance of the API class inside the harness and publish a reference to this instance in the configuration database. However, you must use the *abstract base class* as the type for configuration database.

6) Any component that requires access to the API must declare a handle to the *abstract base* class. (Remember, no code outside the harness will know the extended type definition exists). Components must get the API reference from the configuration database (again, using the base type).

7) Finally, invoke the API methods using the reference retrieved from the configuration database. Even though this is a handle of the base class, polymorphism ensures that we execute the extended method definitions that call the harness methods.

See Reference [5] for more details of polymorphic interaces.

```
interface dut_harness();
  import uvm_pkg::*;
  import abs_pkg::*;

  // The bind targeted module name is "dut"

  dut_if_type dut_if(.if_data_in (dut.data_in),
                     .if_data_out(dut.data_out)
                     …);

  task harness_force_sig1(int data);
    force dut.sub_mod.sig1 = data; // "dut" is a module name
  endtask                          // "sub_mod" is an instance name

  task harness_release_sig1();
    release dut.sub_mod.sig1;
  endtask


  class harness_api extends harness_api_abstract;
    function new(string name="");
      super.new(name);
    endfunction

    task force_sig1(int data);
      harness_force_sig1(data);
    endtask

    task release_sig1();
      harness_release_sig1();
    endtask
  endclass

  harness_api  api;  //instance of API class

  function void set_vif(string path);
    api = new("harness_api");
    uvm_config_db#(dut_if_type)::set(null, path, "dut_vif", dut_if);
    uvm_config_db#(harness_api_abstract)::set(null,"*",
                   "harness_api", api);
  endfunction

endinterface
```

Figure 19. Use an internal class to provide an API for manipulating internal DUT signals.

We can also use this technique to improve on the example in the previous section that enables signal forces from an agent. Rather than placing the `force_en` control bit in the DUT interface, we can put

it in the harness and provide functions for enabling and disabling forced signals. This would allow the DUT interfaces to be more generic and not require customizing with control bits.

6. Limitations and Workarounds

The harness solution we've shown requires interfaces with all signals defined as ports. Ideally we would always create our own interfaces to meet our needs, but this is not always possible. Many projects rely on VIP that come with interfaces defined without ports. In this situation, we can still get some of the benefits of a harness, but we lose the ability to control directions with port coercion. The partial solution is to do the following:

1. Define the harness, instantiating the DUT interface that doesn't have ports.

2. Within the harness, use assign statements to connect the internal interface signals to the DUT, using upward hierarchical reference to the module name.

3. Bind the harness to the appropriate module.

```
interface dut_harness();

  dut_if_type dut_if();

  assign dut.clk = dut_if.clk;
  assign dut.rst = dut_if.rst;
  assign dut.data_in = dut_if.data_in;
  assign dut_if.data_out = dut.data_out;
  ...
endinterface
```

Figure 20. Workaround for using interfaces without ports in a harness.

With this setup, we cannot change signal directions, manipulate master/slave roles, and may face problems working with active RTL drivers. However, we still get the benefits of well encapsulated connections that are easy manage and are independent of module hierarchical paths.

7. Future Work

The UVM Harness solution we've shown in this paper gives us a very flexible testbench that can adapt to all kinds of simulation configurations. However, there are some design configuration changes that most testbenches would not be able accommodate without rewriting the code or maintaining separate testbenches. For example, the system design may parameterize the number of instances of sub-modules that we need to interact with. Additionally, sub-modules may parameterize the number of independent "channels" of signals (e.g., multiple ports of a common protocol). These changes require a different number of agents as well as new interface instances to bind and publish to those agents.

In future work, we will show how to expand the UVM Harness solution even further to support these situations. Since we are no longer restricted by interface connections, we are now free to do much more advanced manipulations of UVM testbench environments. Additionally, we will explore how to take advantage of the Dynamic Configuration feature of VCS, which lets us swap selected design

modules with stubs for each test without recompiling. In theory, we should now be able to simulate any portion of a design with any combination of stubs with a single compile.

8. Conclusions

Most common practices for connecting a testbench to a design are inflexible and place major restrictions on verification strategies. Verification methodologies have advanced to powerful constrained random techniques, yet typical interface connections hold us back from applying the most efficient strategies for reaching our verification goals. We've shown how enhancing the UVM harness with port coercion and max-width interfaces makes testbench connections easier to maintain and universal for any simulation context. Free of the constraints of traditional testbench connections, we now have the flexibility to accomplish much more in a single testbench than what is normally possible. We can control stimulus from any point in a testbench, regardless of hierarchy. We can change the roles of agents to verify any portion of a design in isolation without creating a separate testbench. Testbench connections work with either stubbed or active design modules, giving us both performance benefits and the ability to reach scenarios that would normally be hard to verify. Finally, we can dynamically swap design and testbench stimulus, giving the flexibility to verify features in any portion of the design with a single compile. This especially benefits large SoC designs where we can significantly expand the scope of verification possible while simultaneously reducing the manual effort required.

9. Acknowledgements

Special thanks to Verilab's Kevin Vasconcellos for his invaluable feedback during the development of this paper.

10. References

[1] David Larson, "UVM Harness Whitepaper: The missing link in interface connectivity," Synapse Design Automation, 2011.

[2] IEEE Std 1800-2012, Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language, 2012.

[3] Stuart Sutherland, Don Mills, Chris Spear, "Gotcha Again – More Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know," SNUG San Jose, 2007.

[4] Alex Melikian, Paul Marriot, "Perplexing Parameter Permutation Problems? Immunize Your Testbench," SNUG Canada 2017.

[5] Shashi Bhutada, "Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces," Mentor Graphics Verification Horizons, November 2011.