Slaying the UVM Reuse Dragon
Issues and Strategies for Achieving UVM Reuse

Mike Baird
WHDL
Willamette, OR
mike@whdl.com

Bob Oden
UVM Field Specialist
Mentor Graphics
Raleigh, NC
bob_oden@mentor.com

Abstract **-** With larger and more complex designs the gap between design and verification has grown larger. Because of this the reuse of the testbench both in new projects and within the same project has become very desirable. One of the "promises" of UVM is achieving such reuse. However, in reality, UVM reuse has been limited. This paper identifies the issues that affect UVM reuse and strategies for achieving reuse. A UVM reuse methodology will be presented that provides reuse of components from one testbench to another and within the same testbench from block to chip level.

I. INTRODUCTION

With larger and more complex designs the gap between design and verification has grown larger. The verification effort relative to the design effort has grown until today most projects spend more time on verification than design. Because of this the reuse of portions of the testbench both in new projects and within the same project has become very desirable. One of the selling features or promises of UVM is achieving such reuse. However in reality, UVM reuse has been limited. While UVM provides testbench methodology, there is quite a bit of latitude as to how UVM testbenches are architected. There are many choices that the testbench architect has to make that effect reuse. Without forethought and planning for reuse, inevitably the architect makes some choices that limit reuse. These choices include how the components are structured and packaged, what the interfaces to the various blocks are for configuration etc. and how connection and configuration information is distributed. In this paper we look at some of the major issues that effect reuse and provide tested solutions to these issues.

II. TESTBENCH ARCHITECTURE AND COMPONENT PACKAGING

In every UVM testbench there are parts that can possibly be reused and parts that cannot. One requirement then for reuse is to encapsulate and separate the reusable pieces from the non-reusable pieces. But simply lumping all the reusable parts together is not typically sufficient. Particularly when the DUT itself has multiple interfaces, buses, functional blocks etc.

The reusable parts of the testbench must be organized, grouped or packaged together to facilitate both block-to-top reuse with the same DUT and reuse with a different DUT. Additionally the architecture of the UVM testbench must provide an "interface" to the reusable piece so they can be integrated into another testbench or easily encapsulated within the existing testbench without modification. Other sections of this paper will discuss the "interface" requirements such as configuration, resource sharing and self-containment for these reusable pieces and in another section is discussed block-to-top reuse requirements. This section talks about the testbench architecture and packaging of the components.

The non-reusable parts of the testbench must also be organized or packaged such that reusable packages can be both integrated into an existing testbench or extracted from the testbench for reuse elsewhere.

To facilitate a solution to the issues of component packaging we define three different levels of architecture within a UVM testbench. The top level is called the testbench level and contains non-reusable components. The other two levels are the interface level and environment (or block) level, both of which are reusable levels.

The interface level is comprised of the components for a particular DUT interface, such as a bus, and an associated configuration object. The interface level is considered an element of reuse. It could be reused with any DUT that has the particular interface type (bus etc.).

The interface level consists of interface specific components:

- Agent that encapsulates a monitor, driver and sequencer. Optionally it has interface (API) sequences associated with it.
- Configuration object
- DUT interface

The interface level is broken into two parts.

- Components in the Hardware Description Language (HDL) module. These are the DUT interfaces instantiated along side the DUT.
- Components in the Hardware Verification Language (HVL) module. These are the agent and configuration objects with their sub-components. These are grouped using a uvm_package which is referred to as the interface package.

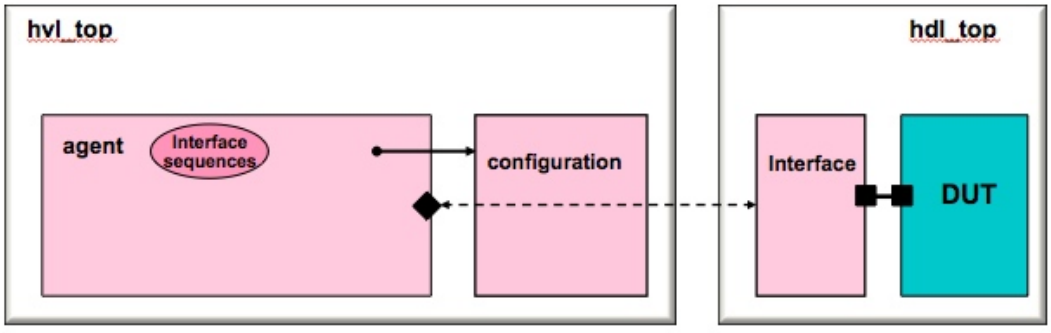

Figure 1. Interface Level Architecture

The environment level encapsulates the HVL components of one or more interfaces and DUT specific verification components such as scoreboards. It has an associated configuration object and optionally environment level sequences that form an API for the environment level. The environment level configuration object encapsulates the interface level configuration(s).

The environment level block corresponds to a block of the DUT (or the entire DUT itself). An environment level block may be encapsulated within another environment level block. I.e a block may be a sub-block to another block and so on.

The environment configuration object encapsulates the interface configuration object(s). As environment level blocks are encapsulated within other environment level blocks there is a corresponding encapsulation of environment configurations within environment configurations.

Encapsulation of environment level blocks and configurations inside a higher-level environment is discussed in more detail in the block-to-top reuse section (VI).

The environment level is grouped in an Environment Package. The environment level is considered an element of reuse. It may be reused with any DUT that has the design block corresponding to the environment.

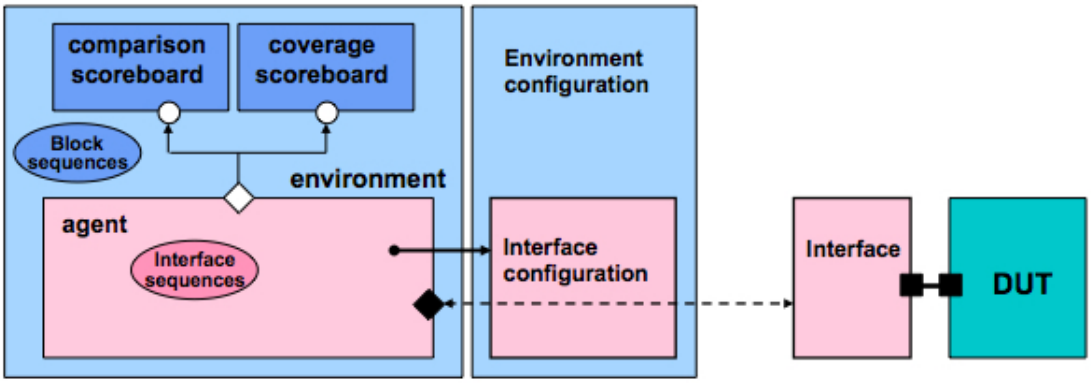

Figure 2. Environment Level Architecture

The testbench level consists of the HDL and HVL top modules, test(s), parameters and top-level sequence(s). These are all non-reuse components. The tests, parameters and top-level sequences are each grouped using UVM packages (Tests Package, Parameters Package, and Sequence Package).

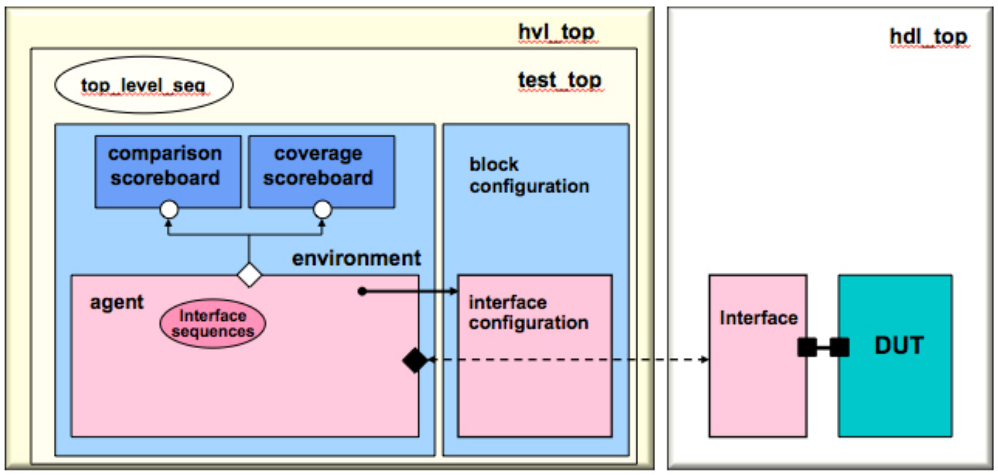

Figure 3. Tesbench Level Architecture

As an example we use a DUT which is an AMBA High Speed Bus (AHB) to Wishbone bus bridge (ahb2wb) from opencores.org. The architecture for this testbench is shown below.

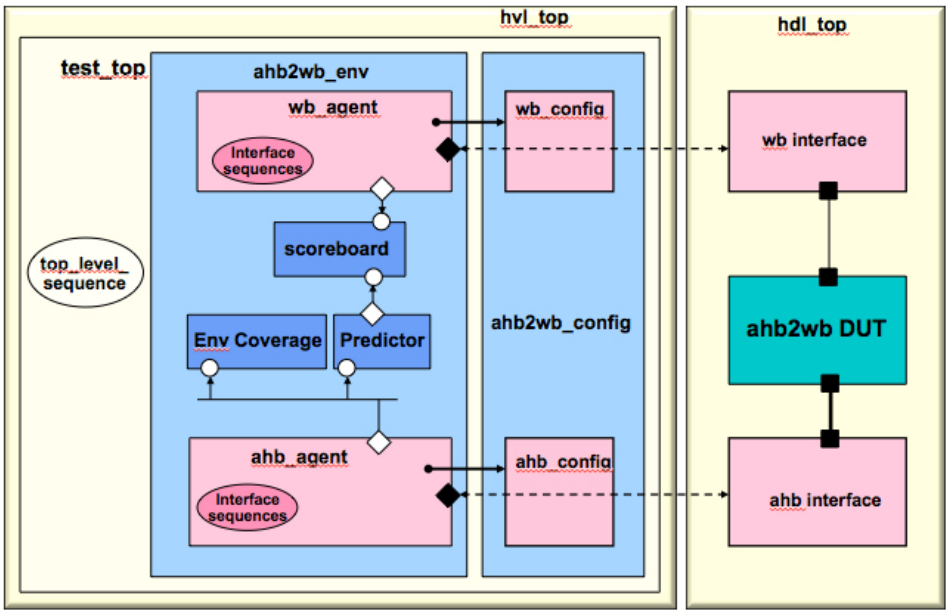

Figure 4. ahb2wb Testbench Architecture

The directory structure for the ahb2wb testbench is shown below. The three different testbench architecture level packages can be seen in this diagram. The testbench level packages are parameters_pkg, sequences_pkg, and tests_pkg. The environment level package is ahb2w2b_env_pkg and the interface level packages are wishbone_pkg and ahb_pkg. Each of these packages includes source files in its corresponding src directory (except parameters_pkg).

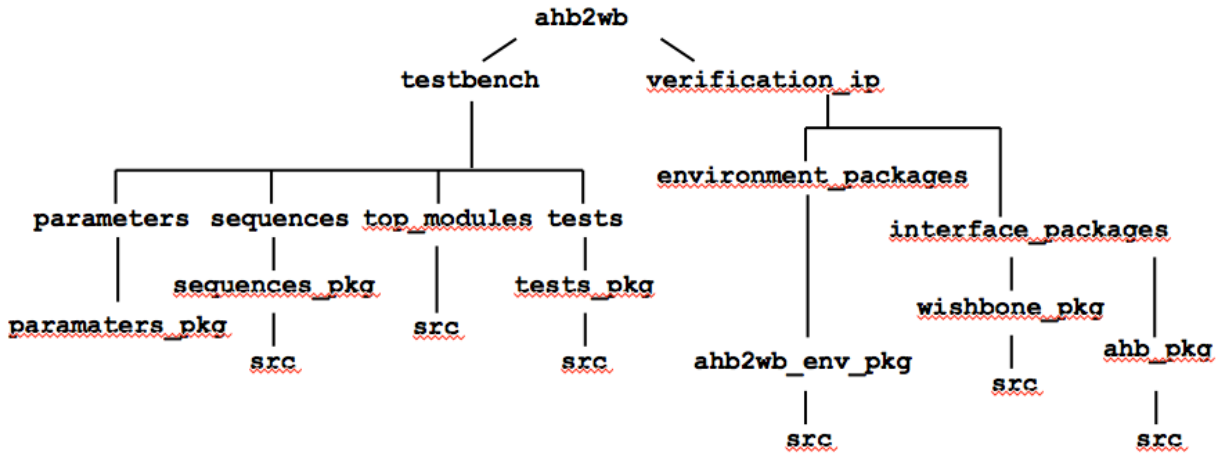

Figure 5. ahb2wb Testbench Directory Structure

If another DUT had a Wishbone interface for example, the Wishbone bus interface level package wishbone_pkg and its associated source files in the src directory could easily be extracted for reuse in that DUT. Likewise the ahb2wb environment level package, or the AHB interface level package could be easily extracted from this structure and reused within another testbench where the DUT had the same interface(s).

III. COMPONENT SELF-CONTAINMENT AND RESOURCE SHARING

One important characteristic of a reusable component is self-containment. A self-contained reusable component can automatically get its own configuration, build and configure its children, and make internal resources available to others. The uvm_config_db can be used to access these resources without requiring knowledge of component implementation. Reusable components that access external resources through uvm_config_db simplify environment development. Reusable components that share internal resources through uvm_config_db simplify test and sequence development.

The UVM configuration class provides access to a centralized database where type specific resources can be stored and received. Resources can be placed into or retrieved from the database at any time during simulation [1]. Set and get functions of uvm_config_db class are used to place resources into and retrieve resources from uvm_config_db respectively. The set and get function prototypes are listed below.

```
uvm_config_db #(T)::set(uvm_component cntxt, string inst_name, string field_name,     T value )
uvm_config_db #(T)::get(uvm_component cntxt, string inst_name, string field_name, ref T value )
```

In the above prototype T is the type of resource being stored or retrieved. The function call scope is defined by cntxt and inst_name arguments. The component's full name is identified by cntxt which is appended to inst_name argument to define a scope. The field_name argument identifies a specific entry in the scope that is being searched. The set call value argument identifies resource being stored. The get call value argument is where resources are returned. Resources that are objects will have their handle placed into value. Resources that are simple data types will have the actual value placed into the value argument.

When cntxt points to a uvm_component its full name will be appended to inst_name to form a scope. A component's full name is retrieved by UVM using the get_full_name() function. When cntxt is null then inst_name defines the scope. This methodology for simplified uvm_config_db use will set cntxt to null. This allows scope to be defined by inst_name. This technique is used to create a scope for each category of resource. All resources in a category share a common scope. A unique field_name argument then identifies a specific resource within a category. Next we will look at various categories of resources that are shared within an environment.

Resources that are typically shared within an environment include virtual interface handles, configuration object handles and sequencer handles. A virtual interface is used to give a class access to an interface. Virtual interface handles are placed in the uvm_config_db to make them available to class based components. Virtual interface handles are retrieved from the uvm_config_db by class based agents, agent configurations, or sequences. A configuration class is used to define settings that determine a component's operation. The configuration class is a container that holds all variables which control a component's operation. Components that may require a configuration class include agents, predictors, scoreboards, environments, sequences, and test classes. A UVM sequencer is used by a sequence to send transactions to the interface through a UVM driver. Sequences need access to the sequencers within an environment in order to start and coordinate stimulus activity for a test scenario. Using uvm_config_db to pass sequencer handles allows a sequence writer to use a sequencer without knowing where that sequencer resides in the environment. Connectivity to an interface is provided through a sequencer stored in the uvm_cofig_db.

The following steps outline how resources are shared for the testbench shown in Figure 8.

The first step in this methodology is to identify every interface in the simulation that will either be actively driven or passively monitored. The virtual interface handles to these interfaces will be placed in the uvm_config_db. Interfaces identified in this step are entered into a resource table. The resource table has all information necessary to place a resource into or retrieve a resource from uvm_config_db. Each row in the table lists an identified interfaces in the simulation. Each column contains information used as arguments in uvm_config_db set and get function calls. An example is given in Table 1.

Interface Description	Type	Transaction	Identifier
ahb_if	ahb_driver_bfm, ahb_monitor_bfm, ahb_configuration	ahb_transaction	“AHB_BUS”
wb_if	wb_monitor_bfm, wb_configuration	wb_transaction	“WB_BUS”
spi_if	spi_driver_bfm, spi_monitor_bfm, spi_configuration	spi_transaction	“SPI_BUS”

Table 1: AHB2SPI Design Resource Table

The Resource Table has four columns: Interface description, type, transaction and identifier. The Interface Description column is a description of each interface. It identifies primary inputs to the DUT as well as internal interfaces being monitored. The Type column identifies an interface type. This field is used as the type , #(T), parameter in uvm_config_db set and get calls for accessing a virtual interface. The type field for set and get function for the first interface is shown here. The Transaction column identifies the transaction type sent to the sequencer for this interface. A UVM sequencer specialization for this type is used as type, #(T), parameter in uvm_config_db set and get calls for accessing a sequencer. The type field for set and get functions used to access the first interface's sequencer is shown here. The Identifier column contains unique strings that are used as the field_name argument in uvm_config_db set and get function calls for virtual interfaces, configurations and sequencers. Each entry in this column must be unique. The uvm_config_db creates a separate database for each type of resource stored. This allows using the same identifier as a field_name for accessing a virtual interface, configration and its sequencer.

The resource table has all information necessary to place a resource into or retrieve a resource from uvm_config_db. This is because general scopes are used for virtual interfaces and sequencers. All virtual interfaces are placed into uvm_config_db using the scope: null, “VIRTUAL_INTERFACES”. All sequencers are placed into uvm_config_db using the scope: null, “SEQUENCERS”. All interface configurations are placed into uvm_config_db using the scope: null, “CONFIGURATIONS”. What identifies a particular interface or sequencer is the string in the ‘Identifier’ column. This string is used in field_name of uvm_config_db calls. This creates an automatic association between an interface, the virtual interface handle used to access the interface, the agents configuration and the sequencer used to place transactions on that interface. These resources are available to test writers without requiring any knowledge of environment structure. The code below shows generic scopes in red. The remaining information needed to access either a virtual interface, configuration or sequencer is contained in the Resource Table and shown below in blue.

To preserve self-containment, the configuration class for an interface agent has a string variable used to store the unique identifier. The value of the string variable is used by the configuration class to place itself in the uvm_config_db. The interface agent places the sequencer into the uvm_config_db using the same string in its configuration class.

```
// Placing the Resources for ahb_if into the uvm_config_db
// Virtual interface handles for driver and monitor BFM's
uvm_config_db #( virtual ahb_driver_bfm  )::set( null, "VIRTUAL_INTERFACES" , "AHB_BUS", value );
uvm_config_db #( virtual ahb_monitor_bfm )::set( null, "VIRTUAL_INTERFACES" , "AHB_BUS", value );
// Sequencer handle
uvm_config_db #( uvm_sequencer #(ahb_transaction))::set( null, "SEQUENCERS", "AHB_BUS", value );
// Configurarion handle
uvm_config_db #( ahb_configuration)::set( null, "CONFIGURATIONS", "AHB_BUS", value );

// Placing the Resources for wb_if into the uvm_config_db
// Virtual interface handle for monitor BFM
uvm_config_db #( virtual wb_monitor_bfm )::set( null, "VIRTUAL_INTERFACES" , "WB_BUS", value );
// Sequencer handle
uvm_config_db #( uvm_sequencer #(wb_transaction))::set( null, "SEQUENCERS", "WB_BUS", value );
// Configurarion handle
uvm_config_db #( wb_configuration)::set( null, "CONFIGURATIONS", "WB_BUS", value );

// Placing the Resources for spi_if into the uvm_config_db
// Virtual interface handles for driver and monitor BFM's
uvm_config_db #( virtual spi_driver_bfm  )::set( null, "VIRTUAL_INTERFACES" , "SPI_BUS", value );
uvm_config_db #( virtual spi_monitor_bfm )::set( null, "VIRTUAL_INTERFACES" , "SPI_BUS", value );
// Sequencer handle
uvm_config_db #( uvm_sequencer #(spi_transaction))::set( null, "SEQUENCERS", "SPI_BUS", value );
// Configurarion handle
uvm_config_db #( spi_configuration)::set( null, "CONFIGURATIONS", "SPI_BUS", value );
```

To retrieve the resources above replace the set function call with get. The value argument is where the handle to the retrieved object will be placed.

IV. Generation of a coherent configuration and register model

To generate a coherent configuration and register model a top-down approach is used with the configuration classes. The configuration classes have a hierarchical structure that parallels the environment hierarchical structure. This hierarchical structure reflects the RTL hierarchy. As RTL blocks are encapsulated in other RTL blocks, block level environments are encapsulated in other environments. Since each environment has a corresponding configuration object, block level configurations are encapsulated in other configurations. The example used in this section describes the configuration for the testbench in "BLOCK-TO-TOP REUSE" section.

The following code is located in the top level UVM test. Information required by the environment(s) and agent(s) is passed to the top level configuration through its initialize function call in line six. This information tells the configuration about the testbench in which it resides. The first argument to the initialize call identifies the simulation level. This is an enumerated type identifying BLOCK level, SUBSYSTEM level, CHIP level, SYSTEM level, etc. This information is used to configure agents as ACTIVE or PASSIVE accordingly. The second argument gives the hierarchical path down to and including the environment. The lower configuration will append hierarchy according to component names in its environment. The interface_names argument is a string array containing the interface identifiers described in COMPONENT SELF-CONTAINMENT AND RESOURCE SHARING section. These identifiers will be distributed to lower level environments. The class type of the configuration object in line six is ahb2spi_configuration.

```
1 class test_top . . .
2
3  virtual function void build_phase(uvm_phase phase);
4    string interface_names[] =  { AHB_BFM, WB_BFM, SPI_BFM } ;
5    super.build_phase(phase);
6    configuration.initialize(CHIP, "uvm_test_top.environment",interface_names);
7  endfunction
8
9 endclass
```

The following code contains part of the class description for the ahb2spi configuration. Since this is a chip level configuration it contains block level configuration classes for the ahb2wb and wb2spi environments.

Lines nine through fifteen are for generating a coherent register model. The register model is always constructed in the top level configuration. Top level configuration passes sub-register block handles down to lower level configurations through the initialize call. In this example ahb2spi is the top level configuration. Therefore the register model handle is null. This triggers construction of the register model. The wb2spi register sub-block is passed to the wb2spi configuration through the initialize call to the wb2spi_configuration in line nineteen. This argument is optional as can be seen by the initialize call to the ahb2wb configuration in line seventeen since the ahb2wb does not have a register model.

Lines seventeen through twenty-one pass information about the testbench from this chip level configuration down through block level configurations. The sim_level argument is passed down directly. Component names within the configurations environment are appended to the environment path received. In this case the chip level environment contains two sub-environments: ahb2wb_env and wb2spi_env. Interface string identifiers are passed down to sub configurations. The ahb2wb_config requires the identifiers for the ahb_if and wb_if. The wb2spi_config requires the identifiers for the wb_if and spi_if.

```
 1 class ahb2spi_configuration . . .
 2
 3  function void initialize(uvmf_sim_level_t sim_level,
 4                                   string environment_path,
 5                                   string interface_names [],
 6                                   uvm_reg_block register_model = null,
 7                                   uvmf_active_passive_t interface_activity[] = null);
 8
 9    if (register_model == null) begin
10       reg_model = ahb2spi_reg_block::type_id::create("reg_model");
11       reg_model.build();
12
13       enable_reg_adaptation = 1;
14       wb2spi_config.enable_reg_prediction = 1;
15     end
16
17     ahb2wb_config.initialize(sim_level, {environment_path, ".ahb2wb_env"},
18                              { interface_names[0], interface_names[1]} );
19     wb2spi_config.initialize(sim_level, {environment_path, ".wb2spi_env"},
20                              { interface_names[1], interface_names[2]},
21                              reg_model.wb2spi);
22
23   endfunction
```

The code below is part of the class description for the ahb2wb_config object in the ahb2spi_configuration class. The ahb2wb is a block level environment that includes agents, predictors, scoreboards, etc. Agents in this environment are configured as either ACTIVE or PASSIVE based on the value of sim_level. In BLOCK level simulations the ahb_if and wb_if are both ACTIVE. In CHIP level simulations the ahb_if is ACTIVE and wb_if is PASSIVE. This is because in chip level simulations this port is driven by wb2spi RTL. In SYSTEM level simulations both ahb_if and wb_if are PASSIVE since both ports are driven by RTL. The sim_level argument is used to configure agents as required for each simulation level.

The agents instantiation name is appended to the enviromnent_path argument. This allows the agent configuration to make itself available to only its agent through the uvm_config_db. The names of the agents are a_agent and b_agent because a parameterized environment is used for the ahb2wb environment.

The last argument to the agent configuration's initialize call is the string identifier for the interface. In this example the first identifier, index zero, is passed to the ahb agent configuration. The second identifier, index one, is passed to the wb agent configuration.

```
 1  class ahb2wb_configuration . . .
 2
 3    function void initialize(uvmf_sim_level_t sim_level,
 4                             string environment_path,
 5                             string interface_names [],
 6                             uvm_reg_block register_model = null,
 7                             uvmf_active_passive_t interface_activity[] = null);
 8
 9      if ( sim_level == BLOCK ) begin
10         ahb_config.initialize( ACTIVE, {environment_path,".a_agent"}, interface_names[0]);
11         wb_config.initialize ( ACTIVE, {environment_path,".b_agent"}, interface_names[1]);
12      end else if ( sim_level == CHIP ) begin
13         ahb_config.initialize( ACTIVE, {environment_path,".a_agent"}, interface_names[0]);
14         wb_config.initialize ( PASSIVE,{environment_path,".b_agent"}, interface_names[1]);
15      end else if ( sim_level == SYSTEM ) begin
16         ahb_config.initialize( PASSIVE, {environment_path,".a_agent"}, interface_names[0]);
17         wb_config.initialize ( PASSIVE, {environment_path,".b_agent"}, interface_names[1]);
18      end else begin
19        `uvm_fatal("CONFIGURATION", "Unknown sim_level in ahb2wb_configuration::set_activity()")
20      end
21    endfunction
```

The code below is part of the class description for the ahb_config object in the ahb2wb_configuration class. The ahb_config is an agent configuration.

The activity argument to the initialize call is used to determine if a driver BFM should be retreived. Agents that are configured as PASSIVE do not require a driver BFM. This is shown in lines thirteen through sixteen.

The agent_path argument is used to make this configuration object available only to its agent through the uvm_config_db call in line eighteen. Line nineteen is used to make the configuration for this interface available using the CONFIGURATIONS generic scope.

The interface_name argument contains the interface string identifier. It is used for retrieving the monitor BFM and driver BFM.

```
 1 class ahb_configuration . . .
 2
 3   virtual function void initialize(uvmf_active_passive_t activity,
 4                                    string agent_path,
 5                                    string interface_name);
 6
 7    active_passive      = activity;
 8    this.interface_name = interface_name;
 9
10    if(!uvm_config_db #(ahb_monitor_bfm)::get(null,VIRTUAL_INTERFACES,interface_name,monitor_bfm) )
11            `uvm_fatal("Config Error" . . .
12
13    if ( activity == ACTIVE ) begin
14       if(!uvm_config_db #(ahb_driver_bfm)::get(null,VIRTUAL_INTERFACES,interface_name,driver_bfm) )
15            `uvm_fatal("Config Error" , . . .
16    end
17
18    uvm_config_db #( ahb_configuration )::set( null ,agent_path,       UVMF_AGENT_CONFIG, this );
19    uvm_config_db #( ahb_configuration )::set( null ,UVMF_CONFIGURATIONS, interface_name, this );
20
21   endfunction
```

The top-down initialize mechanism shown in this section creates a coherent configuration and register model from top to bottom of the environment hierarchy. This flow is completed in the build_phase of the top level UVM test. As a result, the configuration is in place before all lower level components are constructed. Therefore all configuration information is in place to support component construction including topology variation based on randomized configurations.

V. BLOCK-TO-TOP REUSE

A DUT at the chip level may encapsulate sub-blocks within that chip level as opposed to a "flat" architecture. These sub-blocks in turn may encapsulate sub-blocks and so on. SoC designs for example, are typically organized this way. Testing a block within block type DUT organization would typically be done by testing each block individually (block level testing) and then testing the blocks in various combinations with each other. This later level of testing is referred to variously as sub-system, system or chip level testing. We will use the term system-level to denote a level of testing that includes sub-blocks where the "system" could represent a portion of a DUT with multiple blocks, a chip or a system with multiple chip.

In moving from the block level testing to higher levels of testing the desire is to reuse the block level testbench within the system level testbench and the system level testbench within the next level and so forth instead of a copy and paste approach to creating a new testbench for each level. I.e as the DUT "grows" from sub-block to block to chip the testbench must "grow" with it. We call this block-to-top reuse.

As DUT sub-blocks are encapsulated into parent blocks there are several issues for reuse that surface and need to be taken into account:

- The interfaces at the sub-block level may "disappear" from the block level's external point of view. This occurs in SoC designs for example where there are busses that are internal to the chip but not accessible or visible at the chip level. This means that a particular interface may be have stimulus driven from the testbench at one level and not at another level. This is referred to as the interface being "active" or "passive".
- The testbench needs to reflect or mirror this sub-block within block type architecture so that pieces may be added or removed as needed. The configuration, resource sharing and self-containment for these reusable pieces discussed in previous sections must take this into account.

- The actual connection or "wiring up" of the interfaces to the DUT must change as the sub-block interfaces are encapsulated within other blocks

To facilitate block-to-top reuse we define a number of "simulation levels".

```
typedef enum {SUB_BLOCK, BLOCK, SUB_MODULE, MODULE, SUB_CHIP, CHIP,
                                 CIRCUIT_CARD, SYSTEM} uvmf_sim_level_t;
```

Each of the defined levels of simulation corresponds to a "top-block" in the testbench. By defining the different levels of simulation, different behavior for an interface level block or environment level block in the testbench can be specified depending upon the simulation level.

A Wishbone bus to SPI bridge (wb2spi) could be defined as a block with an environment level testbench block shown below. In a BLOCK level simulation both the Wishbone bus agent (wb_agent) and the SPI agent (spi_agent) are active. They have drivers, sequencers etc. and actively drive the DUT.

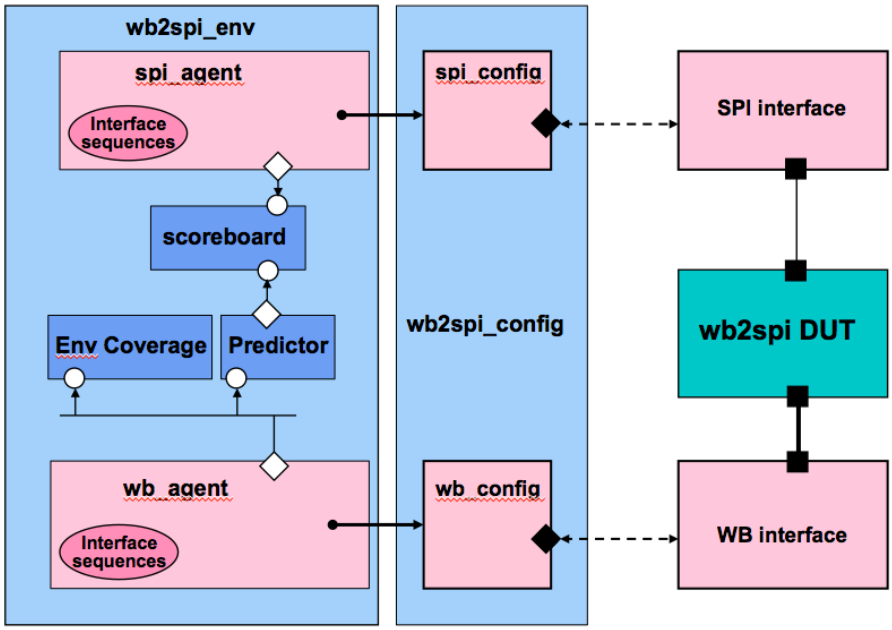

Figure 6. wb2spi Environment Level Architecture

Below is shown a block diagram of the interface details for the Wishbone bus and SPI. To facilitate block-to-top reuse the interface is broken into three parts. There is a signal interface (wb_if, spi_if) that contains nothing but wires for each port of the DUT interface. It is connected as a port to the Bus Functional (BFM) style interfaces for the driver (wb_driver_bfm, spi_driver_bfm) and monitor (wb_monitor_bfm, spi_monitor_bfm).

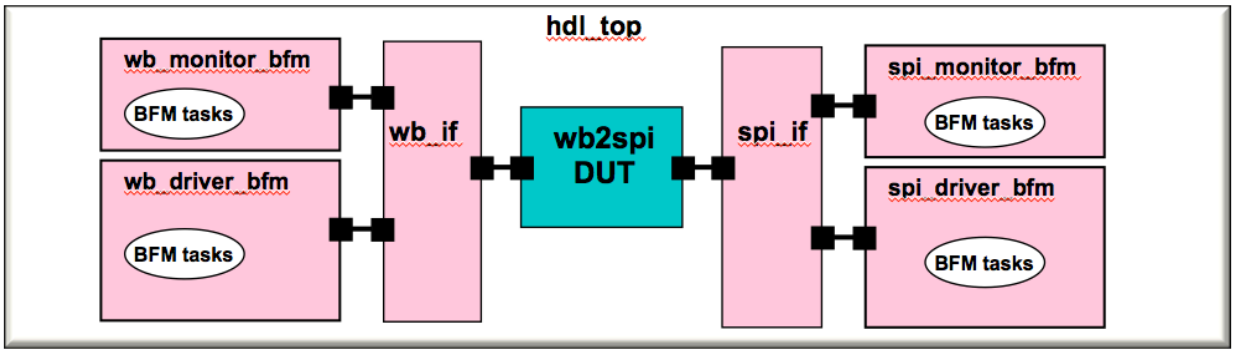

Figure 7. wb2spi Interface Level Architecture

The ahb2wb block shown earlier has a similar architecture. An ahb2spi DUT (top-block) that encapsulates an ahb2wb sub-block and a wb2spi sub-block could be considered to be at the CHIP simulation level.

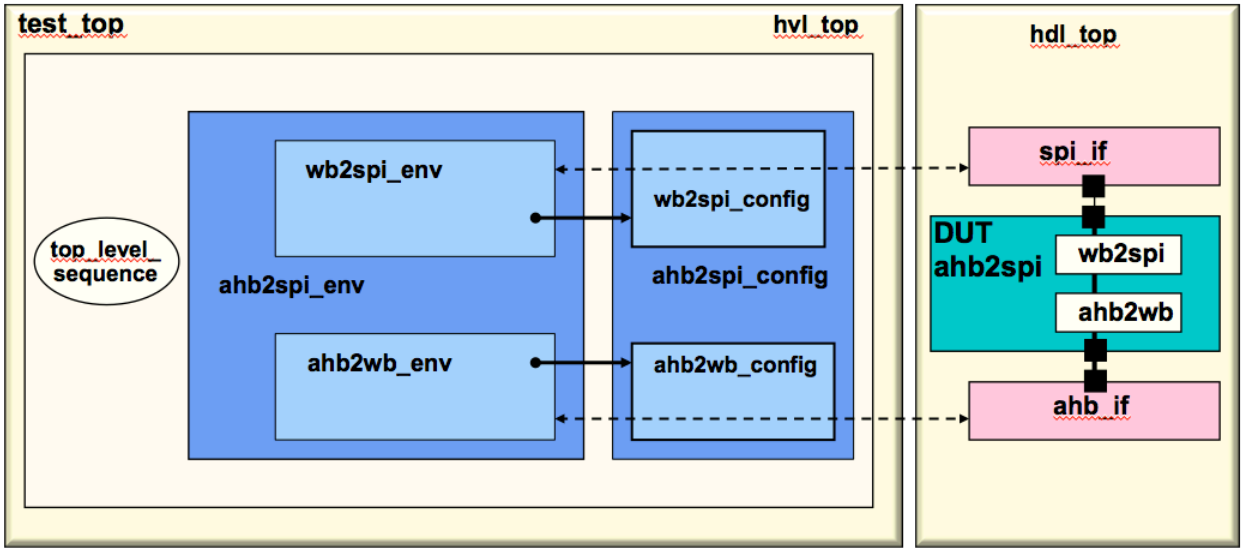

Figure 8. ahb2spi Testbench Architecture

Note that the wb2spi environment (wb2spi_env) and ahb2wb environment (ahb2wb_env) are encapsulated within the ahb2spi environment (ahb2spi_env). Similarly the wb2spi and ahb2wb configuration objects (wb2spi_config, ahb2spi_config) are encapsulated within the ahb2spi configuration object (ahb2spi_config). The ahb2spi level could be embedded within another layer and so on. The structure is regular and scalable facilitating block-to-top reuse.

Note that with the encapsulation of the ahb2wb and wb2spi blocks, the Wishbone bus is embedded within the ahb2spi DUT and is not visible at the pins of the DUT. In a CHIP level simulation the wb2spi environment level block's SPI agent is active and its Wishbone bus agent is passive (monitors only) while on the ahb2wb environment level block the ahb agent is active and again the Wishbone bus agent is passive. An agent then uses the simulation level (BLOCK, CHIP etc.) to determine if it is active or passive.

The Diagram below shows the HDL top interface level details of the ahb2spi. It illustrates why at the interface level the DUT interface is separated into three parts. Since the Wishbone bus agents are passive, only the monitor BFM interface (wb_monitor_bfm) is instantiated. There is no need for the driver BFM interface.

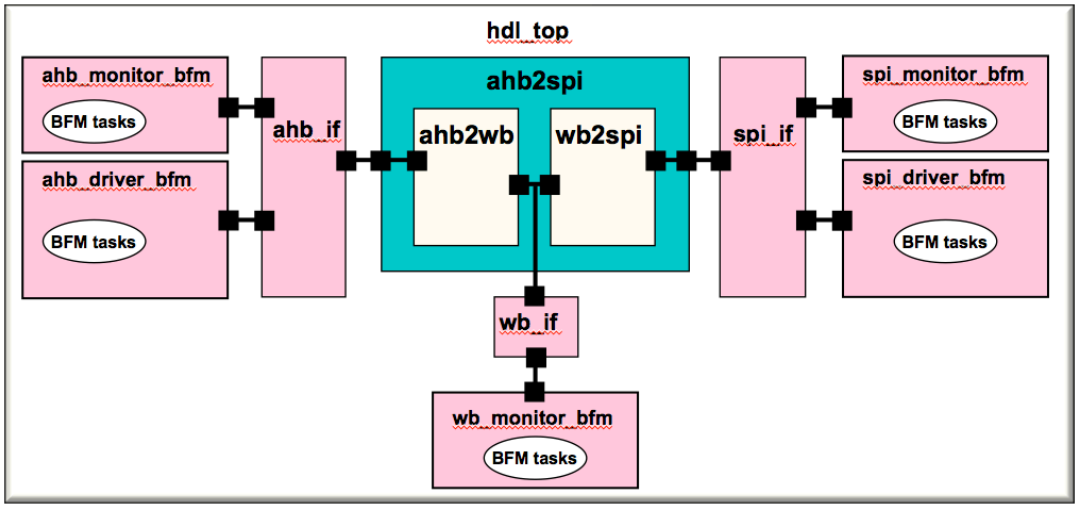

Figure 9. ahb2spi Interface Architecture

The signal interface is separate to facilitate the "wiring up" of the interfaces to the DUT at the different levels of encapsulation, that is as sub-blocks are encapsulated within higher level blocks. In the diagram above the Wishbone bus signal interface (wb_if) is connected to the Wishbone bus signals, which are buried in the DUT, either with hierarchical references or with a SystemVerilog bind statement, while earlier it was instantiated and connected directly to the pins of the block level DUT.

VI. CONCLUSIONS

SystemVerilog and UVM provides the required building blocks to support component and environment reuse. Achieving reuse with SystemVerilog and UVM depends on how those building blocks are used. A clear use model for those building blocks and discipline in using that model consistently will determine the level of reuse success realized. This paper outlined the major factors affecting reuse: self-containment, resource sharing, test bench architecture, component packaging, generation of a coherent configuration and register model. These topics were illustrated using the block to top example contained in Section V. Using these techniques enables component and environment reuse.

VII. FINAL NOTES

This paper describes reuse techniques used in the UVM Framework. The UVM Framework, UVMF, is a UVM jumpstart and reuse methodology provided by Mentor Graphics in partnership with Willamette HDL, WHDL. It has been used by dozens of companies over the last five years. UVMF includes python based generators for creating interface, environment and bench level code. Code generated is ready for simulation and emulation. UVMF training classes are available through WHDL for test writers and environment developers.

Examples provided in UVMF demonstrate technology integrations including Questa VIP, Vista SystemC modeling, iTBA portable stimulus, Questa Verification Run Manager, Visualizer debug and Veloce. UVMF is currently available through the authors of this paper. UVMF will be available as part of the Questa installation starting with the 10.5 release.

REFERNCES

[1] UVM 1.1 Class Reference, Accellera. [Online].
Available: http: //www.accellera.org/downloads/standards/uvm
[2] M. Glasser, "Configuration in UVM: The Missing Manual" in DVCon 2012. Accelera 2012
[3] Baird, Michael, UVM Framework Student Guide, Portland, Willamette HDL, Inc. 2015 (www.whdl.com)