Rockin' the polymorphism for an Elegant UVM testbench Architecture for a Scalable, Highly Configurable, Extensible DUT

Michael Baird
Willamette HDL, Inc.
6107 SW Murray Blvd, #407, Beaverton OR 97008, mike@whdl.com 503-590-8499

Frank Verhoorn
Northwest Logic, Inc
1100 NW Compton Drive #100, Beaverton Oregon 97006 fverhoorn@nwlogic.com 503-533-5800

***Abstract*- When a DUT is highly scalable, highly configurable, extensible and even highly parameterized it presents major obstacles in the development of the UVM testbench framework architecture and stimulus generation. To match these DUT characteristics in an elegant manner requires extensive use of the Object Oriented Programming (OOP) concept of polymorphism. This paper will illustrate the use of polymorphism, along with an innovative use of the string-based UVM factory to over-come these obstacles.**

I. Introduction

Northwest Logic's memory controller product is a single IP with a scalable number of highly configurable memory channels. Northwest Logic's goal was to develop a UVM testbench for the IP that was scalable and configurable to match the IP and could be easily extended support to new interface types. The desire was to reach the goal without resorting to using conditional compilation code, which would require extensive rewrites when extending the ports.

A requirement of the project was that stimulus for testing the memory controller is interface type independent. For example memory write transaction stimulus was required to be written in such a manner that it didn't matter if the actual port interface type was an AXI bus or an AHB bus.

A further constraint unique to the project was that the DUT extensively used parameters that were not available to the UVM testbench because of the architecture of the scripting system used to execute the simulations. Yet, the host interfaces connected to the DUT were heavily parameterized. This made the use of virtual interfaces for communication between the testbench and the DUT problematic.

This paper will illustrate how the project goals were met using polymorphism in the construction of the UVM framework and stimulus generation.

II. Scalable, Configurable, Extensible IP and UVM Framework

Figure 1 below shows the general UVM testbench and Northwest Logic memory controller IP architecture. The IP contains a scalable number of memory channels controllers. Each memory channel controller itself can present a scalable number of SoC bus interfaces referred to as "ports". Each port is either connected to one of several SoC host inter-face bus types or it may be unconnected. Examples of supported buses include, AMBA™ AXI, AMBA™ AHB, Open Core Protocol (OCP), and Northwest Logic proprietary interface among others. The host interface type is extensible to other types beyond those currently supported.

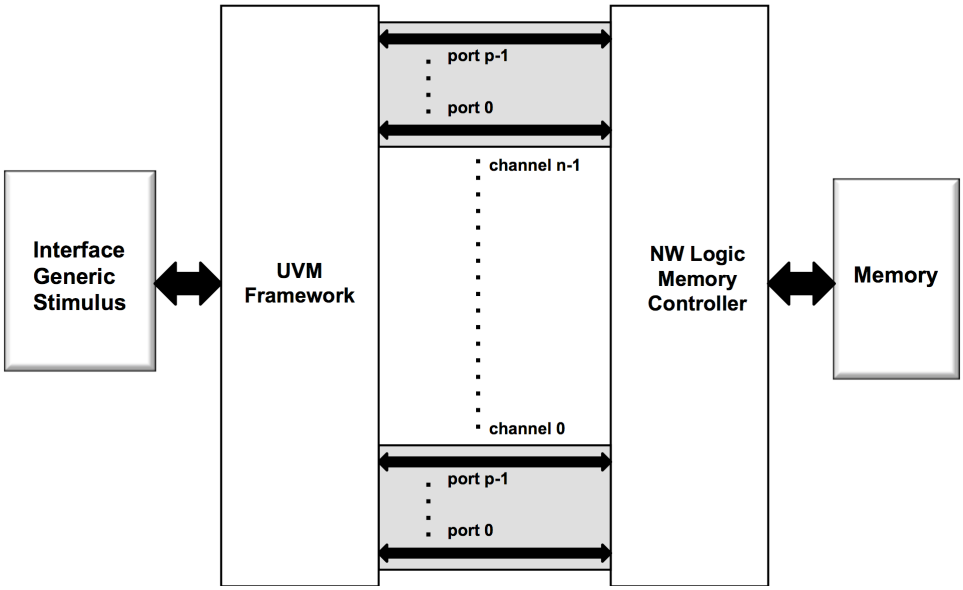

Figure 1. General Testbench and IP Architecture

The scalability and configurability of the IP places a number of requirements on the UVM testbench.

Agents:

- The number of agents for each channel must be scalable to match the number of ports in the channel.
- The type of the agent must match the port type, including "unconnected". There is no fixed pattern as to the distribution port types in a channel.
- Must be extensible to additional agents types in the future.
- Must be able to incorporate 3rd party VIP agents

Configuration objects:

- The number of configuration objects for each channel must be scalable to match the number of ports in the channel.
- Must be configurable to match the corresponding agent.

Transaction item types must be configurable to match the corresponding agent. Tests and high level sequences need to be written independent of the port type. The scoreboard for each channel needs to accept any of the possible transaction item types. Finally, the virtual interface distribution needs to be scalable and configurable.

III. UVM Testbench Framework

A. *UVM Testbench Architecture*

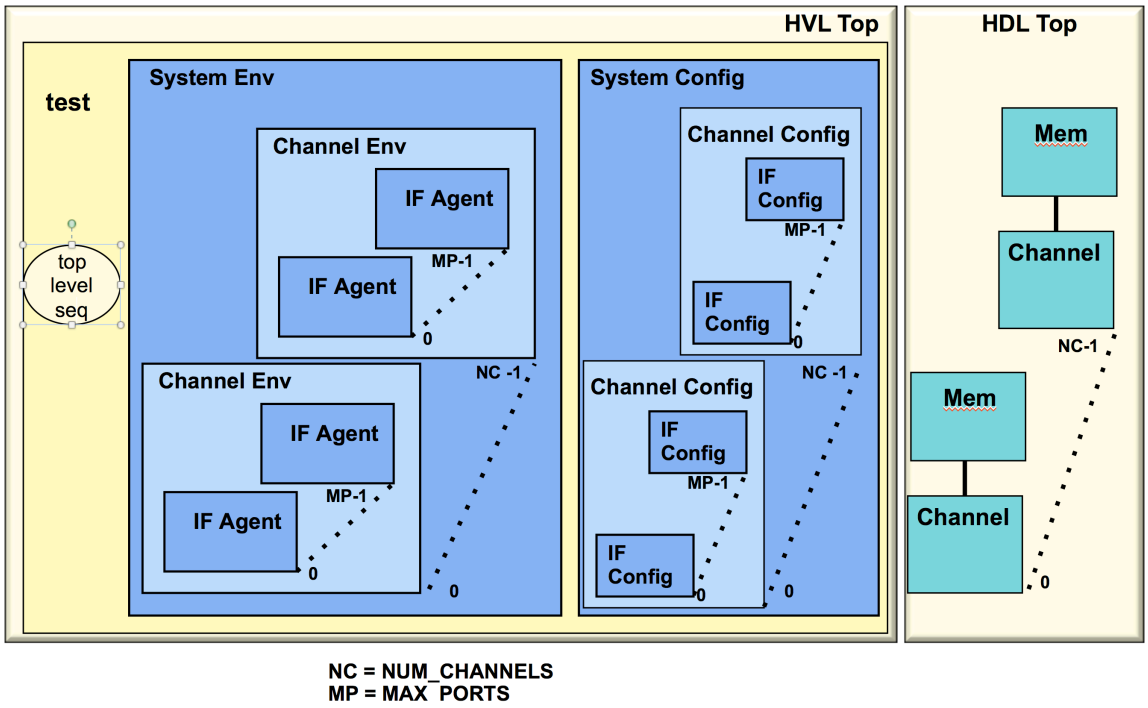

Figure 2. Testbench Architecture

Figure 2 (above) shows the DUT and testbench architecture in more detail while figure 3 (below) shows the structural hierarchy of the testbench. The top-level environment in the UVM testbench is called the system environment. The system environment has an array of channel environments, one for each each memory channel in the DUT. A channel environment contains an array of interface agents, one for each port of the channel. The configuration objects are arranged hierarchically in a manner to match the environment hierarchy. Each environment and agent has a corresponding configuration object.

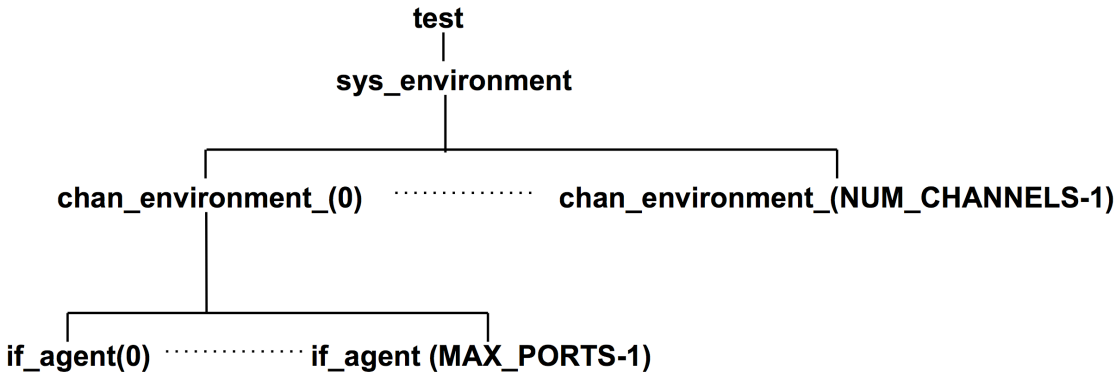

Figure 3. Testbench Structural Hierarchy

Figure 4 (below) shows the architecture of a single channel in more detail. Each channel has a scoreboard and an array of agents, one for each populated port of the channel. There is a corresponding channel configuration object for each environment and for each agent. Mirroring the configuration object structure to the environment and agent structure makes for easier scaling of the testbench to match the number of channels or ports in the testbench[3].

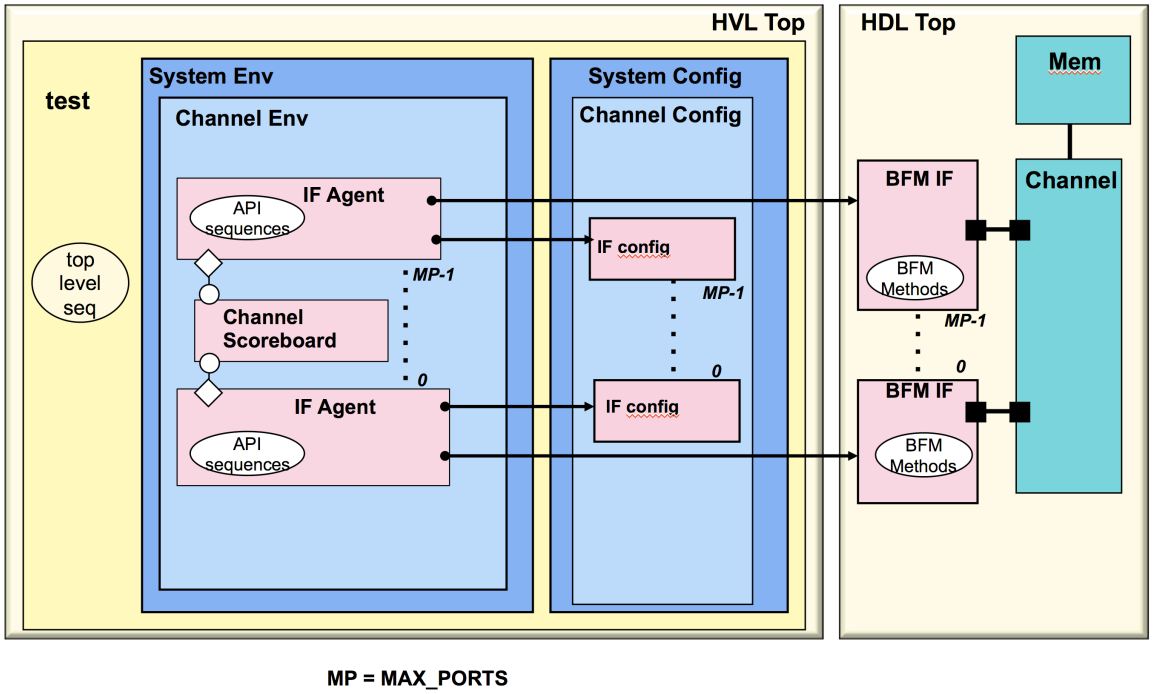

Figure 4. Single Channel Architecture

B. *Specifying the Configuration*

A top-level package includes two parameters and an array used to specify the configuration of the DUT and UVM testbench.

```
package top_params_pkg;
  parameter NUM_CHANNELS = 2;  // Num Memory controller channels
  parameter MAX_PORTS    = 8;  // Maximum num ports per channel
  // Multi-dimensional array of port types
  string if_port_types[NUM_CHANNELS][MAX_PORTS];
  ...
endpackage
```

Snippet 1. top_params_pkg

The array (`if_port_types`) is a multi-dimensional array of type `string`. The string is used as a prefix to indicate the channel type and ends with an underscore ("_") character for readability. For example `"ahb_"` for AMBA™ AHB, `"axi_"` for AMBA™ AXI, `"wb_"` for WISHBONE, `"ocp_"` for OCP. The port types are extensible to new types by simply adding additional prefix definitions. As will be illustrated the string prefix is used in the creation of agents, configuration objects and transaction items in

such a way as to make the code generic and not bus specific. Thus avoiding conditional compilation or the use of case statements or if-else structures to determine the type of object to be created.

C. Polymorphism and the Testbench – DUT Connection

Virtual interfaces are the typical way in UVM testbenches to communicate between the testbench and DUT interfaces. In a highly configurable testbench the mapping of connection type to the drivers and monitors that need them is best done polymorphically for the reasons already discussed. However virtual interfaces are not classes and so cannot be treated polymorphically. There are two solutions to this. The easiest and most commonly used solution is to wrap the virtual interface in a class derived from a base wrapper class. The second solution does not use virtual interfaces but rather uses an approach that is referred to as the Abstract-Concrete class [1] [2] approach. In this approach a proxy object is instantiated inside the DUT interface. A handle to this proxy object is provided to the drivers and monitors that communicate with the interface.

A unique constraint for the project was the interfaces were highly parameterized; yet because of the use of a legacy script to create the DUT those parameters were not available to the UVM testbench. This made the use of virtual interfaces problematic for the testbench-DUT communication, even when wrapped inside of a class. The Abstract-Concrete class approach was therefore adopted.

The figure below shows the inheritance structure of the proxy objects used for testbench-DUT communication.

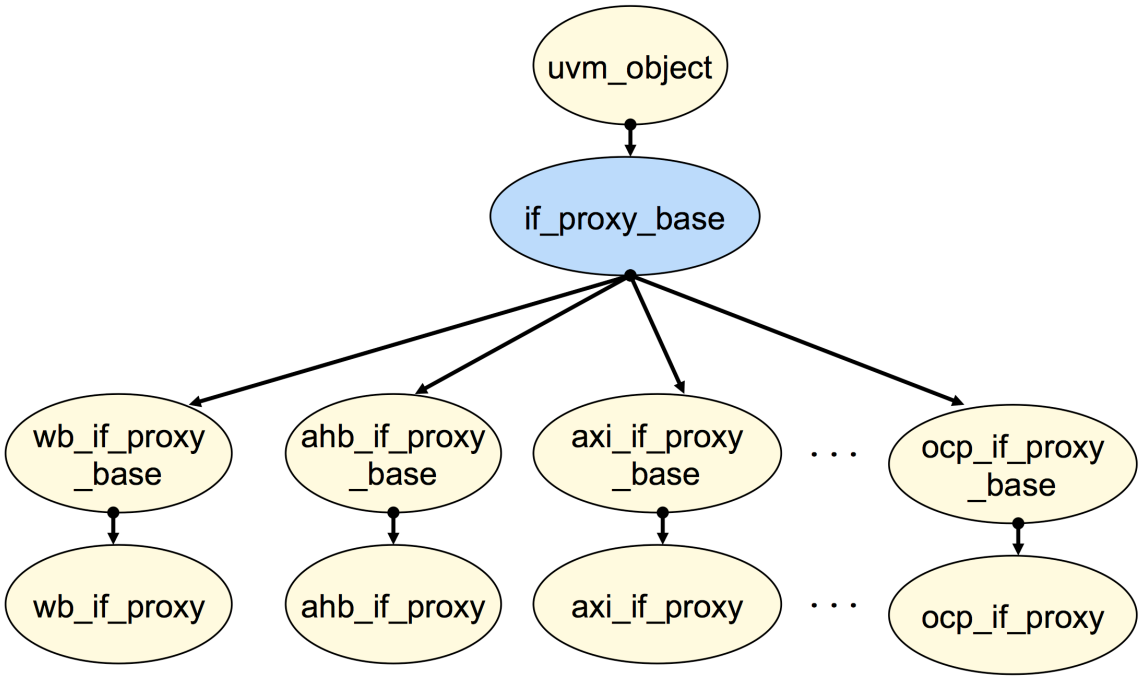

Figure 5. Configuration Object Inheritance Structure

The first step in this approach is to define a proxy base class that serves a base class placeholder when using the configuration database access methods `uvm_config_db::set()`, `uvm_config_db::get()`, and for the base type for arrays. In the diagram above `if_proxy_base` is this class. It is an empty class that is registered with the factory.

The second step is to create an interface proxy base class for each interface type (`wb_if_proxy_base` etc.). These classes are derived from the proxy base class (`if_proxy_base`) and contain empty virtual methods, which form an API to the particular interface. The `wb_if_proxy_base` class is shown below.

```
class wb_if_proxy_base extends if_proxy_base;
`uvm_object_utils(wb_if_proxy_base)

  // Constructor not shown
  // API methods
  virtual task write(input wb_txn txn); endtask
  virtual task read(input wb_txn txn);  endtask
  virtual task monitor(output wb_txn txn); endtask
endclass
```

Snippet 2. wb_if_proxy_base

The third step is to define *inside the BFM interface* an interface proxy class (`wb_if_proxy`) that is derived from the interface proxy base class (`wb_if_proxy_base`). The API methods are overridden to provide the interface behavior. An instance of the proxy object is created inside the interface and a handle to the proxy object is placed in the configuration database for the test class to fetch and distribute via configuration objects. See the `wb_bfm_if` interface code example below. Note the use of the class `if_proxy_base` in the set of the handle into the configuration database.

```
interface wb_bfm_if #(int CH_NUM = 0, int PORT_NUM = 0) ();

class wb_if_proxy extends wb_pkg::wb_if_proxy_base;
...
  // Overrides of the proxy base API methods
  virtual task write(input wb_txn txn);
    ...
  endtask
  // Overrides of read() and monitor() not shown
endclass

wb_if_proxy wb_proxy; // declare derived proxy if handle

initial begin
  wb_proxy = new("wb_proxy");  // create derived proxy if object
  // place proxy if object in config_db NOTE use of if_proxy_base
  uvm_config_db#(sys_config_pkg::if_proxy_base)::set(null,"PROXY_IF",
       $sformatf("proxy_if_ch%0d_p%0d",CH_NUM, PORT_NUM), wb_proxy );
end
endinterface
```

Snippet 3. wb_if_proxy_base

In the fourth step the test class fetches the proxy object handle from the configuration database and places them in an array. Inside the test base class (`test_base`) an array of the `if_proxy_base` type is declared (see the code below). This is a "polymorphic array" meaning while the type is a base class, any derived class object may be assigned to any element in the array. The test base class fetches the derived interface proxy objects from the configuration database and places them in this array. It uses the `if_port_types` array to determine if a port is populated and a proxy if object should exist or not.

```
class test_base extends uvm_test;
...
  // Interface proxy objects array
  if_proxy_base p_if [NUM_CHANNELS][MAX_PORTS];

    // get proxy ifs
    foreach(p_if[i,j])
      if(if_port_types[i][j] != "")  // interface present?
        // yes get proxy object
        if(!uvm_config_db #( if_proxy_base )::get(null,"PROXY_IF",
                 $sformatf("proxy_if_ch%0d_p%0d",i, j), p_if[i][j] ))
          `uvm_fatal(...)
        else `uvm_info(...)
...
endclass
```

Snippet 4. test_base

Finally the proxy object handles are distributed to the monitors and drives that need them via configuration objects. Configuration objects distribution is described in the next section.

D. Polymorphism and the Configuration Objects

For each channel of the DUT there is a corresponding channel environment (see figure 4). For each port of a channel there is a corresponding agent. For each channel environment there is a corresponding configuration object of type `chan_configuration`. For each agent there is a corresponding interface configuration object. Since a port may be one of several different interface types the interface configuration objects must also be of any interface type. A base class interface configuration type is defined (`if_config_base`) to allow for polymorphic substitution of specific derived class configuration object type. See the figure below. For VIP we create a wrapper around the VIP configuration object so it can be substituted for `if_config_base` like any other derived configuration object type.

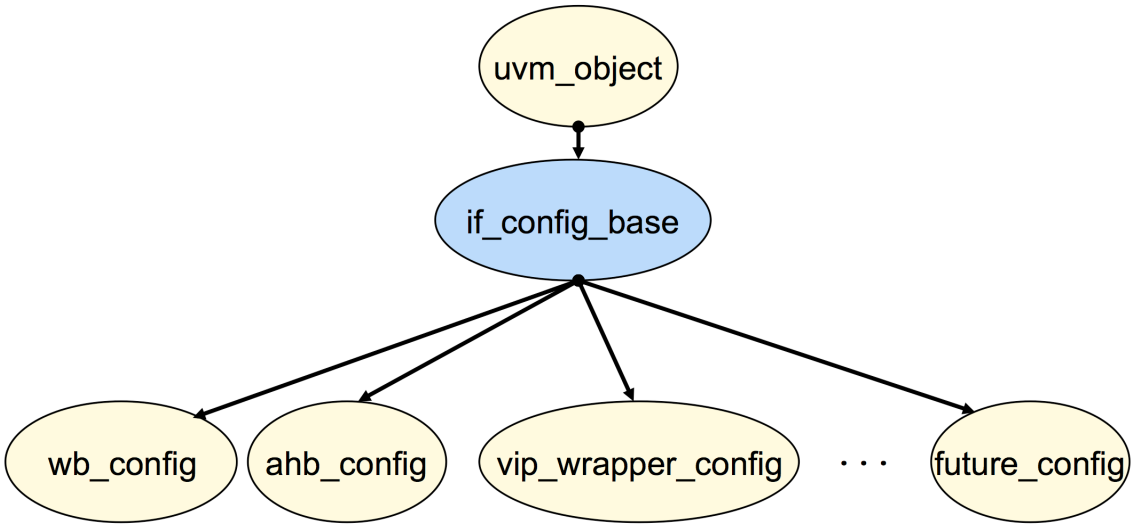

Figure 6. Configuration Object Inheritance Structure

```
class chan_configuration extends uvm_object;
`uvm_object_utils(chan_configuration)
  global_configuration global_config; // global configuration object
  int chan_number; // Channel number this config object
  // Fetch factory handle
  protected uvm_factory m_factory = uvm_factory::get();
  // Child configuration objects
  if_config_base  if_config[]; // array of if configurations
  ... // constructor not shown
  function void initialize(
      global_configuration global_config_arg, // global config object
      int chan_number_arg,                    // channel ID
      string if_port_types_arg[MAX_PORTS],    // port types array
      if_proxy_base p_if_arg [MAX_PORTS]   // Interface proxy objects
    );
```

Snippet 5. chan_configuration

Inside the channel configuration object is declared a dynamic array (`if_config`) of type `if_config_base` (see the above code snippet). The array is "polymorphic" and may be populated with any derived configuration object type. This is the array of configuration objects for each of the agents corresponding to the populated ports in the channel. The configuration object is provided with the channel ID number, the array of port types and the interface proxy objects (see the `initialize()` method arguments in the code snippet above).

```
    chan_number   = chan_number_arg;    // set local channel number
    global_config = global_config_arg;  // Set local global config
    // create IF configuration objects
    if_config = new[global_config.max_ports]; // resize array
    foreach(if_config[i])
      if(if_port_types_arg[i] != "") begin    // port connected?
        $cast(if_config[i], m_factory.create_object_by_name(
           // name of type to create
           {if_port_types_arg[i],"configuration"},
           "", // parent
           // name of object
           {if_port_types_arg[i],$sformatf("config_ch%0d_p%0d",
                                           chan_number, i)}
                          )
            );
 ...
endfunction

...
```

Snippet 6. chan_configuration continued

In the body of the `initialize()` method (see the above code snippet), first the properties `chan_number` and `global_config` are initialized and the array of configuration objects (`if_config`) is resized to the max number of ports in the channel. Then the array of interface

configuration objects is populated as the method iterates through the array creating the correct type of configuration object. A bit about the factory is in order here to understand how the correct type is created.

When an object or component is registered with the factory it is actually registered both by type and by name (a string). Even though there is only one factory with two different ways to refer to the registered types (by type or by name) we will refer to them separately as the type-based factory and the string-based factory. Alternately we refer to either creation by type or creation by name. Creation by type using the type-based factory is what most UVM developers use with this common syntax:

```
handle_name = class_name::type_id::create("handle_name", this);
```

Verification engineers rarely use the string-based factory primarily because there are issues with creation by name with parameterized classes. It may be noted that creation by name is what the method `run_test()` uses to create the test class object.

Here, by design, we do not have parameterized classes to create. So we use creation by name as a clever way of creating the correct class type which *does not* require modification if new port types are added. In this create by name approach the factory method **`create_object_by_name()`** is used. This method takes three arguments:

- Type to be created (a string)
- Parent, which we will ignore since we are not creating components here
- Object name

Since we are creating a configuration object, concatenating the prefix string from the `if_ports_type` array with the string "`configuration`" gives us the correct configuration object type to be created. For example, if the prefix is "`wb_`", indicating a WISHBONE bus interface on the particular port then a `wb_configuration` object is created and assigned to the array of configuration objects (`if_config`). Note this requires that all configuration objects be named "`*_configuration`".

This approach using the string-based factory eliminates the need to use an if-else or case structure to determine the correct type of configuration object to create. The advantage is it is easily extensible to new types where the if-else or case structure code would need to be modified each time a new type was added.

The actual initialization and distribution of the configuration objects are beyond the scope of this paper.

E. Polymorphism and the Agents

Since a port can be of any interface type each agent must be of the correct type for its corresponding port. To facilitate this a base class agent is defined (`if_agent_base`) to allow for polymorphic substitution of specific derived class agents. The UVM library `uvm_agent` base class can't be used in this case because we want to place some common stuff for all the agents in the agent base class.

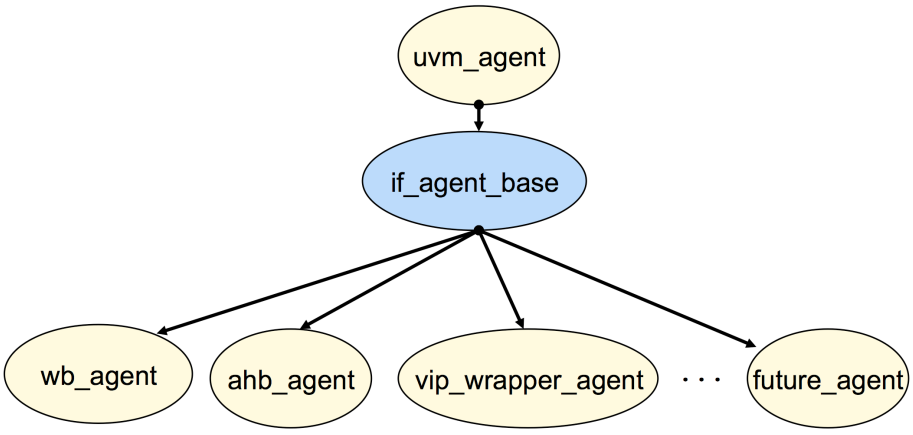

Figure 7. Agent Inheritance Structure

For VIP we create a wrapper around the VIP agent so it can be substituted for `if_agent_base` like any other derived agent type.

Inside the channel environment is declared a dynamic array of type `if_agent_base` that is sized to the maximum number of ports in the channel and may be populated with any derived agent type.

The agents are created using the same string-based factory approach as the configuration objects described earlier in section D to ensure creation of the correct agent type. The only real difference is the `create_component_by_name()` method is used instead of `create_object_by_name()` since we are creating components here. The code below shows the creation of the correct agent and the population of the array of agents inside of the channel environment's `build_phase()`.

```
class chan_environment extends uvm_env;
  if_agent_base        if_agent[];    // array of interface agents
...
  function void build_phase(uvm_phase phase);
    if_agent = new[global_config.max_ports]; // resize agent array
    foreach(if_agent[i]) // walk through every port
      if(chan_config.if_config[i] == null)
        ; // not connected - do nothing
      else begin
        // create the agent using the string based factory
        $cast(if_agent[i], m_factory.create_component_by_name(
                    {chan_config.if_config[i].if_type,"agent"},
                    this.get_full_name(),
                    {chan_config.if_config[i].if_type,
                        $sformatf("agent_ch%0d_p%0d",
                                  chan_config.get_chan_num(), i)},
                     this)
              );
        // set agent config object
        if_agent[i].set_config(chan_config.if_config[i]);
      end
  endfunction
```

Snippet 7. chan_environment

III. UVM Testbench Stimulus Generation

In order to support the configurability of the memory channel, a project requirement was that the same stimulus be used regardless of the interface type for the channel ports.

A. Polymorphism and Agent API Sequences

The testbench defined a simple API for all agents. The API consisted of write and read commands. Each of the commands was defined to contain data of 1 or more bytes. The API was implemented as sequences associated with an agent type. Each agent type had its own interface specific implementation of the write and read API sequences. The figure below shows the inheritance structure for the write API sequences. A similar structure exists for the read API sequences

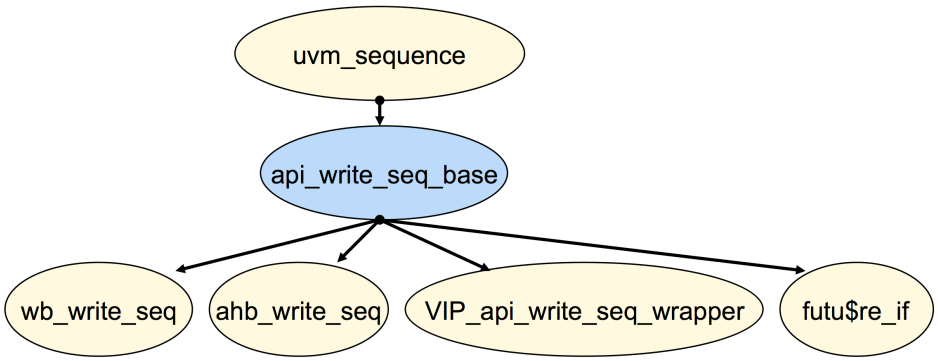

Figure 8. Agent API Sequence Inheritance Structure

To enable polymorphism the write API sequences inherit from a common base class `api_write_seq_base`. For VIP we create a wrapper around the VIP write API sequence so it can be substituted for `api_write_seq_base` like any other agent API sequence type.

```
class api_write_seq_base extends uvm_sequence #(if_txn_base);
  // registration and constructor not shown

  // API method. Does nothing by default
  // It is expected to be overridden
  virtual task init_start_seq(
          output uvm_status_e   status,            // status
          input uvm_sequencer #(if_txn_base) seqr, // sequencer
          input bit[63:0] addr,   // address
          input bit[7:0]  data[], // data array
          input int       size    // transfer size
          );
  endtask
endclass
```

Snippet 8. api_write_seq_base

The code snippet above shows `api_write_seq_base`. It has a single virtual method `init_start_seq()` that is meant to be overridden by any derived sequence such as the `wb_write_seq`. The arguments to the method contain all the information needed to do a write on the interface and is supplied by the "calling" sequence. The code snippet below shows the `wb_write_seq`. It initializes a `wb_txn` transaction item and then starts itself to communicate with the supplied sequencer in a wishbone agent.

```
class wb_write_seq extends api_write_seq_base;
// registration and constructor not shown

  wb_txn txn; // transaction item

  // override API method for this class
  task init_start_seq(
          output uvm_status_e   status, // status
          input uvm_sequencer #(if_txn_base) seqr, // sequencer
          input bit[63:0] addr,   // address
          input bit[7:0]  data[], // data array
          input int       size    // transfer size
```

```
          );
    // add behavior
    txn = wb_txn::type_id::create("txn"); // create txn
    txn.init_txn(       // Initialize the wb_txn
            WRITE,      // set as a WRITE transaction
            addr,
            data,
            size,
            );
    this.start(seqr);     // start the sequence (itself)
    status = txn.status;  // set the return status
  endtask

  task body();
    if_txn_base temp; // Temp handle
    start_item(txn);
    finish_item(txn);
    get_response(temp);
    $cast(txn,temp); // cast to derived class handle
  endtask

endclass
```

Snippet 9. api_write_seq_base

B. Polymorphism and Stimulus Generating Sequences

For the interface stimulus generating sequences a base class sequence is defined (`if_seq_base`) to allow for polymorphic substitution of specific derived class interface sequences. The figure below shows the interface sequences inheritance structure

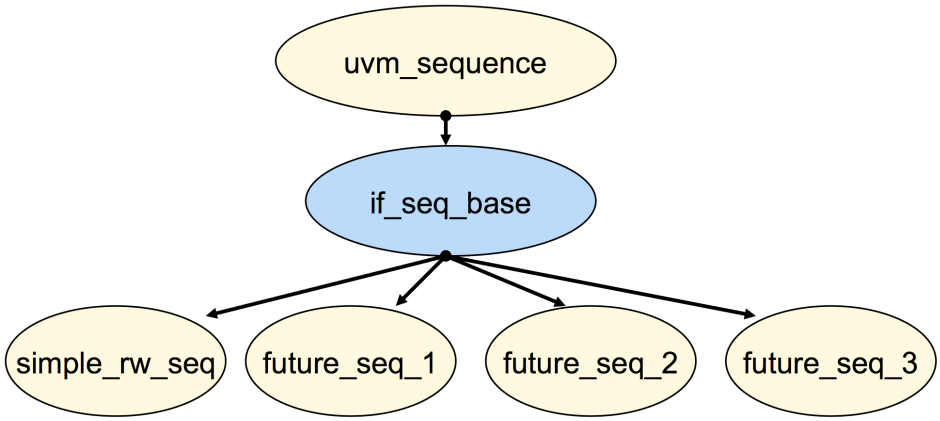

Figure 9. Interface Sequence Inheritance Structure

The base class `if_seq_base` provides `write()` and `read()` methods for use in the derived sequence types. These methods check the port type and call the appropriate agent API sequences. Below is a code snippet from `if_seq_base`. Note that in the `pre_start()` method, all the configuration related properties are initialized for the port that the sequence will be talking to.

```
class if_seq_base extends uvm_sequence;
sys_configuration    sys_config;    // memory system config object
```

```
  global_configuration global_config; // global configuration object
  // "ID" information for host interface port
  int chan_number; // Channel number
  int port_number; // port number
  // host if information
  if_config_base if_config; // handle to port host if config object
  string         if_type;   // if_type prefix for this port
  if_proxy_base  if_p_if;   // proxy handle to host if port interface
  uvm_sequencer #(if_txn_base) seqr; // sequencer handle for agent
  uvm_factory m_factory = uvm_factory::get(); // factory handle
  // constructor not shown
  task pre_start();  // Configure base sequence
    // get config object for host if port to communicate with
    $cast(if_config,
        sys_config.chan_config[chan_number].if_config[port_number]);
    if_type       = if_config.get_if_type();
    global_config = sys_config.get_global_config();
    if_p_if       = if_config.get_p_if();
    seqr          = if_config.get_seqr();
  endtask
```

Snippet 10. if_seq_base

In the code snipped from `if_seq_base` below is the `write()` method that is inherited by the derived interface sequence types. The `write()` method creates the API sequence using the string-based factory to choose the correct write API sequence type in the same manner as described earlier for configuration objects and agents. It then initializes and starts the API sequence, which will create a "write" transaction item to be sent to the driver.

```
//*************************************
  // Host interface access methods
  virtual task write(
          output uvm_status_e status, // status
          input bit[63:0] addr_arg,
          input bit[7:0]  data_arg[], // array of bytes for data
          //transfer size(in terms of data array width)
          input int size_arg           );
    // create the api sequence using the string based factory,
    api_write_seq_base write_seq;  // base class handle
    // create API sequence
    $cast(write_seq, m_factory.create_object_by_name(
                               {if_config.if_type,"write_seq"},
                               "", "write_seq"
                               )
          );
          // initialize and start api sequence
          write_seq.init_start_seq(
                      status,       // status
                      seqr,         // sequencer (from seq_base)
                      addr_arg,     // address
                      data_arg,     // data array
                      size_arg
                    );
...
endclass
```

Snippet 11. Continuation of if_seq_base

A sequence that inherits from `if_seq_base` simply calls the inherited `write()` and `read()` methods, supplying the necessary arguments to generate patterns of reads and writes to a port.

IV. Conclusions

Using polymorphism can significantly reduce the amount of code needed for a highly configurable testbench. In the Northwest Logic project polymorphism was used in the channel environments, agents, configuration objects and API sequences to make the UVM testbench truly configurable, extensible and scalable. It does require forethought and planning to properly set up the needed base classes and inheritance structure.

Using the string-based factory instead of the type-based factory allows for creating the correct type without if-else or case decision trees, making it easy to extend to new types without modifying the code. This technique can be used for configuration objects, environments, agents, API sequences etc. The downside of the string-based factory approach is the types created using it cannot be parameterized. This was not an issue for the Northwest Logic project as it was only used for derived classes that deliberately avoided parameterization.

References

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

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

[3] Bob Oden, Michael Baird, Slaying the UVM Reuse Dragon, Issues and Strategies for Achieving UVM Reuse, DVCon February 2016