Easy uvm_config_db use
A simplified and reusable uvm_config_db methodology for environment developers and test writers

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

Abstract - Effective and efficient use of uvm_config_db is still a challenge for many verification teams. Some teams avoid use of uvm_config_db and thereby miss benefits it provides. Others overuse uvm_config_db and suffer performance degradation. This paper presents a simple and balanced uvm_config_db use model that is reusable, scalable, and emulatable.

Keywords – uvm_config_db, resource, reuse, emulation

I. Introduction

UVM experience level required to architect and implement an environment is typically greater than what is required to write test cases.

Environment developers are increasingly employing component reuse in development of simulation environments. These components typically require outside resources for configuration. They also may contain resources required by other components and objects within the simulation. The uvm_config_db can be used to share resources among simulation components and objects.

Test writers are responsible for creating sequences that implement stimulus scenarios required to verify a design's features, operation, and performance. These sequences often require access to interface signals, configuration objects, and sequencers. These resources are scattered throughout the environment and can be difficult to locate. It is unnecessary to require test writers to understand environment structure and location of these resources. The uvm_config_db can be used to make these resources available to test writers without requiring knowledge of environment structure and resource location.

II. Background

A. Resources and reuse

One important characteristics 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.

B. A simplified methodology for uvm_config_db use

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.

While resources can be placed into or retrieved from the database at any time during simulation resource precedence applies. Resource precedence determines which resource is returned when multiple matches are found when using get. During build_phase all resources are placed into the database with precedence equal to *default_precedence – depth* where *default_precedence* is 1000. As a result, resources placed into the database at a higher UVM hierarchy have precedence over resources placed into the database at a lower level. After the build_phase all resources are placed into the database using *default_precedence* value. As a result, the last resource placed into the database will have precedence over all previously entered resources. [2]

```
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 components 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.

III. Resources that are typically shared within an environment

A. Virtual interfaces

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.

B. Configurations

A configuration class is used to define settings that determine a components operation. The configuration class is a container that holds all variables which control a components operation. Components that may require a configuration class include agents, predictors, scoreboards, environments, sequences, and test classes.

C. Sequencers

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.

IV. An example of resource sharing using this uvm_config_db methodology

Sections IV and V will demonstrate this uvm_config_db use model using the design given in Figure 1. It is important to note that example IV uses string literals from the table. This is done to show where string values from the table are used in uvm_config_db set and get calls. Replacing string literals in section IV with string variables allows for code reuse. Use of string variables is demonstrated in section V.

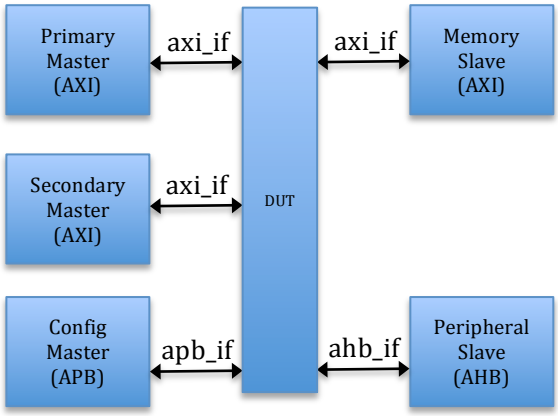

Figure 1: Example Design

A. Identify the virtual interfaces

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.

B. The Resource Table

Interfaces identified in the previous 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. For section IV only code for "Primary AXI Master" will be shown. Section V will show code for all interfaces in Figure 1.

Interface Description	Type	Transaction	Identifier
Primary AXI Master	axi_if	axi_transaction	primary_master
Secondary AXI Master	axi_if	axi_transaction	secondary_master
Config APB Master	apb_if	apb_transaction	config_master
AXI Memory Slave	axi_if	axi_transaction	memory_slave
PeripheralAHB Slave	ahb_if	ahb_transaction	peripheral_slave

Table 1: Example Design Resource Table

The Resource Table has four columns:

1. Interface Description Column

This column is a description of each interface. It identifies primary inputs to the DUT as well as internal interfaces being monitored.

2. Type Column

This 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.

```
uvm_config_db #( virtual axi_if )::set(…
uvm_config_db #( virtual axi_if )::get(…
```

3. Transaction Column

This 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 interfaces sequencer is shown here.

```
typedef uvm_seuqencer #(axi_transaction) axi_sequencer_t;

uvm_config_db #( axi_sequencer_t )::set ( …
uvm_config_db #( axi_sequencer_t )::get ( …
```

4. Identifier Column

This column contains unique strings that are used as the field_name argument in uvm_config_db set and get function calls for virtual interfaces 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 and its sequencer.

```
typedef uvm_seuqencer #(axi_transaction) axi_sequencer_t;

uvm_config_db #( virtual axi_if  )::set( cntxt , inst_name , "primary_master", …
uvm_config_db #( axi_sequencer_t )::set( cntxt , inst_name , "primary_master", …

uvm_config_db #( virtual axi_if  )::get( cntxt , inst_name , "primary_master", …
uvm_config_db #( axi_sequencer_t )::get( cntxt , inst_name , "primary_master", …
```

C. Accessing the resources

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". 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, 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 or sequencer is contained in the Resource Table and shown below in blue.

```
// Placing the virtual interface into the uvm_config_db
uvm_config_db #( virtual axi_if )::set( null, "VIRTUAL_INTERFACES" , "primary_master", value );
// Placing the sequencer into the uvm_config_db
uvm_config_db #( uvm_sequencer #(axi_transaction))::set( null, "SEQUENCERS", "primary_master",
value );

// Retrieving the virtual interface from the uvm_config_db
uvm_config_db #( virtual axi_if )::get(null, "VIRTUAL_INTERFACES", "primary_master", value );
// Retrieving the sequencer from the uvm_config_db
uvm_config_db #( uvm_sequencer #(axi_transaction))::get( null, "SEQUENCERS", "primary_master",
value );
```

D. Enabling transaction viewing

Using an identifier string from the Resource Table also enables controlling transaction viewing from the command line. Regression simulations are best run in command line mode. Failing tests can be rerun in GUI mode with transactions added to the waveform without recompilation. Interfaces can be selected for transaction viewing using the Resource Table identifier string. This requires that the agent's configuration class have a variable that controls transaction viewing and that the configuration class retrieve the value for the variable during simulation runtime using the uvm_config_db.

```
// In the configuration class of the agent

// This variable is used to conditionally enable adding transactions to the waveform viewer
bit enable_transaction_viewing = 0;

// This code uses UVM command line processing to allow optional setting of this variable
// It retrieves the value set for enable_transaction_viewing for the primary_master interface
uvm_config_db #(uvm_bitstream_t)::get(
                null,"primary_master","enable_transaction_viewing",enable_transaction_viewing));
```

```
// Add this to the command line to enable viewing of primary_master transactions in the waveform
+uvm_set_config_int="primary_master",enable_transaction_viewing,1
```

V. Example implementation for all entries in the resource table

As noted in section IV, the code in this example replaces the string literals with string variables. This enables code reuse across interfaces of the same type. The Resource Table identifier string is passed into the code through a string variable. This string variable will be shown as *interface_name* throughout this section.

A. Defining the unique identifier

```
// In a test level parameters package
// This code defines the unique string used to identify each interface in the resource database.
// A parameter is used to eliminate runtime errors due to typing errors.
//
parameter string PRIMARY_MASTER    = "primary_master";
parameter string SECONDARY_MASTER  = "secondary_master";
parameter string CONFIG_MASTER     = "config_master";
parameter string MEMORY_SLAVE      = "memory_slave";
parameter string PERIPHERAL_SLAVE  = "peripheral_slave";
```

B. Sharing and accessing the virtual interfaces

```
// In the module where the interfaces reside

uvm_config_db #( virtual axi_if )::set(
                        null , "VIRTUAL_INTERFACES" , PRIMARY_MASTER,   primary_axi_master_bus );

uvm_config_db #( virtual axi_if )::set(
                        null , "VIRTUAL_INTERFACES" , SECONDARY_MASTER, secondary_axi_master_bus );

uvm_config_db #( virtual apb_if )::set(
                        null , "VIRTUAL_INTERFACES" , CONFIG_MASTER,    config_apb_master_bus );

uvm_config_db #( virtual axi_if )::set(
                        null , "VIRTUAL_INTERFACES" , MEMORY_SLAVE,     axi_memory_slave_bus );

uvm_config_db #( virtual ahb_if )::set(
                        null , "VIRTUAL_INTERFACES" , PERIPHERAL_SLAVE, peripheral_ahb_slave_bus );
```

```
// In the agents configuration class
// Each protocol would have its own configuration class: axi_configuration, apb_configuration, etc.
// For this example xxx represents axi, apb, or ahb.

class xxx_configuration extends uvm_object;

   // This string variable holds this agents interface identifier
   string interface_name;

   // This variable holds the virtual interface handle
   virtual xxx_if xxx_bus_vif;

   // This function is used to configure the agent
   function void configure_interface(string interface_name, string path_to_agent)
      // Store the interface identifier string for use when placing sequencer in uvm_config_db
      this.interface_name = interface_name;

      // Get this agents interface from the uvm_config_db
      uvm_config_db #( virtual xxx_if )::get(
                                 null, "VIRTUAL_INTERFACES", interface_name, xxx_bus_vif );

      // Make this configuration available to the agent through the uvm_config_db
      uvm_config_db #( xxx_configuration )::set( null ,path_to_agent, "AGENT_CONFIG", this );
```

C. Sharing and accessing the sequencers

```
// In the agents class declaration file
// Each protocol would have its own agent class: axi_agent, apb_agent, etc.
// For this example xxx represents axi, apb, or ahb.

class xxx_agent extends uvm_agent #(…

   // This variable is a handle to this agents configuration class
   xxx_configuration xxx_config;

   function void build_phase (uvm_phase phase);

      // Get this agents configuration
      uvm_config_db #( xxx_configuration )::get( this , "" , "AGENT_CONFIG" ,  xxx_config );
      …
      // Place this agents sequencer in the uvm_config_db using the interface identifier
      uvm_config_db #( uvm_sequencer #(xxx_transaction) )::set(
                          null , "SEQUENCERS" , xxx_config.interface_name, xxx_sequencer );
      …
```

```
// In the top level sequence

// Define and instantiate sequencer handles
typedef uvm_seuqencer #(axi_transaction) axi_sequencer_t;
axi_sequencer_t primary_master_sequencer, secondary_master_sequencer, memory_slave_sequencer;

typedef uvm_seuqencer #(apb_transaction) apb_sequencer_t;
apb_sequencer_t config_master_sequencer;

typedef uvm_seuqencer #(ahb_transaction) ahb_sequencer_t;
ahb_sequencer_t peripheral_slave_sequencer;

// Retrieve each sequencer from the uvm_config_db
uvm_config_db #( axi_sequencer_t )::get(
                              null, "SEQUENCERS", PRIMARY_MASTER,   primary_master_sequencer );

uvm_config_db #( axi_sequencer_t )::get(
                              null, "SEQUENCERS", SECONDARY_MASTER, secondary_master_sequencer );

uvm_config_db #( apb_sequencer_t )::get(
                              null, "SEQUENCERS", CONFIG_MASTER,    config_master_sequencer );

uvm_config_db #( axi_sequencer_t )::get(
                              null, "SEQUENCERS", MEMORY_SLAVE,     memory_slave_sequencer );

uvm_config_db #( ahb_sequencer_t )::get(
                              null, "SEQUENCERS", PERIPHERAL_SLAVE, peripheral_slave_sequencer );
```

D. Enabling transaction viewing

```
// In the agents configuration class
// Each protocol would have its own configuration class: axi_configuration, apb_configuration, etc.
// For this example xxx represents axi, apb, or ahb.
// Please note that uvm_bitstream_t is required by the UVM command line uvm_set_config_int function

Class xxx_configuration extends uvm_object;

   // Flag that controls adding transactions to the wave form
   bit enable_transaction_viewing = 0;

   function void configure_interface (string interface_name, string path_to_agent)

      uvm_config_db #(uvm_bitstream_t)::get(
            null, interface_name, "enable_transaction_viewing", enable_transaction_viewing);
```

```
// In the agents monitor class
// Each protocol would have its own monitor class: axi_monitor, apb_monitor, etc.
// For this example xxx represents axi, apb, or ahb.

class xxx_monitor extends uvm_monitor #(…

   // This variable is a handle to this agents configuration class
   xxx_configuration xxx_config;

   // This variable is a handle to an xxx_transaction
   xxx_transaction trans;

   task run_phase (uvm_phase phase);
      …
     if ( xxx_config.enable_transaction_viewing )
        // The add_to_wave function of trans adds trans to a transaction viewing stream
        trans.add_to_wave(transaction_viewing_stream);
   …
```

```
// Add any of the following to the simulation command line to enable transaction viewing
// for the selected interface(s)
+uvm_set_config_int=PRIMARY_MASTER,enable_transaction_viewing,1
+uvm_set_config_int=SECONDARY_MASTER,enable_transaction_viewing,1
+uvm_set_config_int=CONFIG_MASTER,enable_transaction_viewing,1
+uvm_set_config_int=MEMORY_SLAVE,enable_transaction_viewing,1
+uvm_set_config_int=PERIPHERAL_SLAVE,enable_transaction_viewing,1
```

VI. Summary

This paper has shown a uvm_config_db use that meets the needs of environment developers and test writers alike. It allows environment developers to create interface and environment packages that support component reuse across projects and environment reuse from block to top. It does not require test and sequence writers to know details about an environments implementation. Components that use this methodology are easier for environment developers to use when creating simulation environments. Resources within environments that use this methodology are easier for test writers to access when writing sequences to implement test scenarios.

VII. Final notes

Use of uvm_config_db outlined in this paper is taken from the UVM Framework, UVMF. The UVMF is a class library that is a UVM jumpstart and UVM reuse methodology. As a UVM jumpstart it helps teams avoid common mistakes by providing base components, defining an environment architecture, defining a package structure, providing makefiles and examples. As a UVM reuse methodology it enables component reuse across projects, across sites, and from block to top. UVMF has been adopted by over twenty companies in North America and Europe. It's been used to verify hundreds of designs. It has been used on FPGA and ASIC projects by small and large teams with a wide range of UVM experience levels. The UVMF is open source and available from the author.

VIII. Reference

[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