Demystifying the UVM Configuration Database

Vanessa R. Cooper
Verilab, Inc.
vanessa.cooper@verilab.com

Paul Marriott
Verilab Canada
paul.marriott@verilab.com

Abstract

The configuration database in the UVM is a highly versatile feature that allows the passing of objects and data to various components in the testbench. However, despite its versatility, the configuration database (uvm_config_db) can be a source of great confusion to those verification and design engineers who are trying to learn UVM. The goal of this paper is to demystify the uvm_config_db for the novice user. This paper will examine the functionality of the uvm_config_db starting with its relationship to the resource database. It will discuss how the database is implemented in the UVM library, and it will explore how to add to and retrieve from the database. In addition, practical examples will illustrate how to use the database in various scenarios.

The questions that need to be answered are as follows:

- What is the *uvm_config_db*?
- When is the *uvm_config_db* used?
- How is data stored and retrieved?
- How do I debug when something goes wrong?

uvm_resource_db

The UVM configuration database, *uvm_config_db*, is built on top of the UVM resource database, *uvm_resource_db*. The *uvm_resource_db* is a data sharing mechanism where hierarchy is not important. The database is essentially a lookup table which uses a string as a key and where you can add and retrieve entries.

---

class uvm_resource_db#(type T=uvm_object)

---

static function void set(input string scope,
input string name,
T val,
input uvm_object accessor=null)

uvm_resource_db#(bit)::set("CHECKS_DISABLE", "disable_scoreboard", 1, this)

---

static function bit read_by_name(input string scope,
input string name,
inout T val,
input uvm_object accessor=null)

uvm_resource_db#(bit)::read_by_name("CHECKS_DISABLE", "disable_scoreboard", disable_sb)

uvm_config_db::set

In what ways does the *uvm_config_db* differ from its parent, the *uvm_resource_db*? The *uvm_config_db* is used when hierarchy is important. Unlike the resource database, there are only two functions that are most commonly used with the configuration database:

- set – adds an object to the *uvm_config_db*
- get – retrieves an object from the *uvm_config_db*

---

class uvm_config_db#(type T=int) extends uvm_resource_db#(T)

The classic example of *uvm_config_db* usage is with sharing a virtual interface. A SystemVerilog interface is instantiated at in the top level and now needs to be added to the *uvm_config_db* using the *set()* function.

---

static function void set(uvm_component cntxt,
string inst_name,
string field_name,
T value)

uvm_config_db#(virtual tb_intf)::set(uvm_root::get(), "*", "dut_intf", vif)

uvm_config_db#(TYPE)::set(this, "*.path", "label", value)

Argument	Description
uvm_component cntxt	The context is the hierarchical starting point of where the database entry is accessible.
string inst_name	The instance name is the hierarchical path that limits accessibility of the database entry.
string field_name	The field name is the label used as a lookup for the database entry.
T value	The value to be stored in the database of the parameterized type. By default the type is int.

uvm_config_db::get

The next method that needs to be explored is the *get()* function which is used to retrieve items from the database. It is important to note that objects are not removed from the database when you call *get()*. The actual variable is passed in as an inout formal function argument and so is performed as a copy-in-copy-out operation.

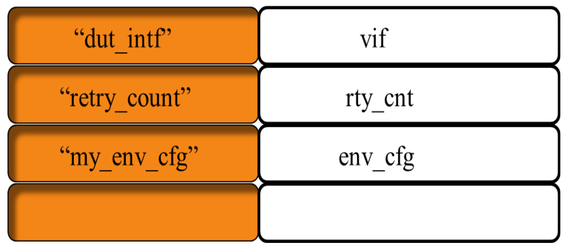
---

static function bit get(uvm_component cntxt,
string inst_name,
string field_name,
inout T value)

uvm_config_db#(TYPE)::get(this, "", "label", value)

In the following diagram, three different items have been added to the uvm_config_db: a virtual interface, an integer value, and a configuration object. Also, there is a generic calls to the *set()* and *get()* functions. To retrieve the integer value the label would be "retry_count" and the value stored in this entry would be assigned to the rty_cnt property in the object that is calling the get() function.

```
uvm_config_db#(TYPE)::set(uvm_root::get( ), "*.path", "label", value);
```




```
uvm_config_db#(TYPE)::get(this, "", "label", value);
```

Debug

The biggest source of bugs is due to the fact that many of the arguments to resource and configuration database methods are of type string. This means that typos in the actual arguments cannot be detected at compile time, but must wait until a test is actually run.

Fortunately, there are debugging facilities available to help find the source of these problems. Two run-time options are available which can be used to turn on tracing of every write and read access to the databases.

---

sim_cmd +UVM_TESTNAME=my_test
+UVM_RESOURCE_DB_TRACE

sim_cmd +UVM_TESTNAME=my_test +UVM_CONFIG_DB_TRACE

UVM_INFO @ 0: reporter [CFGDB/SET] Configuration '.agent.*.in_intf' (type virtual interface pipe_if) set by = (virtual interface pipe_if) ?*

UVM_INFO @ 0: reporter [CFGDB/SET] Configuration '.monitor.out_intf' (type virtual interface pipe_if) set by = (virtual interface pipe_if) ?*

UVM_INFO @ 0: reporter [CFGDB/GET] Configuration 'uvm_test_top.env.penv_in.agent.driver.i_intf' (type virtual interface pipe_if) read by uvm_test_top.env.penv_in.agent.driver = null (failed lookup)

Conclusion

In this paper we demystify the use of the UVM's resource and configuration databases. These are powerful facilities that are available to testbench writers that help with the configuration of the testbench itself as well as provide a repository for parameters that represent values required by different parts of the environment.

All of the code examples in this paper were from "Getting Started with UVM: A Beginner's Guide" by Vanessa Cooper and published by Verilab Publishing. Copies of the code will be made available on request to the authors.