# Demystifying the UVM Configuration Database

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

Paul Marriott
Verilab Canada
Montreal, Quebec
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.**

***Keywords—UVM; SystemVerilog; Configuration Database***

## I. INTRODUCTION

The configuration database in the UVM is a source of confusion and mystery, especially as some of its functionality is tied-in with various automation macros. Because some of the behavior is hidden from users through the use of these macros, or even just through properties of some of the UVM base classes, the functionality of the configuration database, as well its closely related cousin, the resource database, can be difficult to understand. Both databases are a powerful feature of the UVM that aim to make it easier to configure testbenches as well as setup appropriate parameters for testcases to use. This paper will provide practical examples that illustrate how to use the database in various scenarios as well as clarify some issues that can be troublesome, even for veteran users.

The questions that need to be answered for new users 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?

## II. WHAT IS THE CONFIGURATION DATABASE

The UVM configuration database, *uvm_config_db*, is built on top of the UVM resource database, *uvm_resource_db* [1]. To fully understand the *uvm_config_db*, the *uvm_resource_db* must first be explored.

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. Each of these entries is often referred to as a *resource* and we will use that name throughout this paper. When accessing the database, you first must specify the resource type as a parameter. The class header is shown below:

```
class uvm_resource_db#(type T=uvm_object)
```

It is important to note that the resource database makes use of type-parameterized classes. This means that `T` in the class header must be replaced by the actual type of the resource you want to work with. Each type of resource, therefore, gets its own specialized class.

There are several common functions of the *uvm_resource_db* class that allow you to add or retrieve data. The following table highlights the most common functions used.

TABLE I. *UVM_RESOURCE_DB* METHODS

| Methods | Description |
|---|---|
| `get_by_type` | This function gets the resource by the type specified by the parameter so the only argument is the scope. |
| `get_by_name` | This function gets a resource by using both the scope and name given when it was added to the database. |
| `set` | This function creates a new resource in the database with a value, scope, and name that will be used for retrieval. |
| `read_by_name` | This function locates a resource by scope and name and returns the value through an output argument. |
| `read_by_type` | This function locates the resource using only the scope as a lookup and returns the value through an output argument. |
| `write_by_name` | This function locates the resource by scope and name and writes the value to it. If the resource does not exist then it will be created like the set function. |
| `write_by_type` | This function locates the resource by the scope and writes the value to it. If the resource does not exist it is created. |

Let's look at a practical example of how the *uvm_resource_db* could be used. Let's assume there is a scoreboard in the testbench, and it has a *bit* "`disable_sb`" that turns off checking if the value is one. In your test, you can choose whether to turn off the scoreboard or leave it enabled. You

---

www.verilab.com

store the value of that *bit* in the *uvm_resource_db* and it is retrieved by the scoreboard. The figure below illustrates the usage of the `set()` and `read_by_name()` functions that could be used for this scenario. The function prototypes are shown in italics.

```
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)
```

It is important to note that all of the methods of the *uvm_resource_db* class are *static* so they must be called using the scope resolution operator, `::`. As mentioned previously, the classes are type-parameterized by the type of the resource so this has to be specified. In the examples above, we're using resources of type `bit`. The first argument of the `set()` function is the string scope which one can think of as a *category*. Multiple elements can be added to the database with the same scope, but the name, which is the second argument, must be unique across all elements having the same type and scope. However, if you are accessing an element by type where the scope is the only lookup parameter, then there can only be one object of that type using that scope name. The third argument of the `set()` function is the actual value that is to be stored in the database. The final argument "`this`" is for debugging. It allows messages to show where the `set`() originated. If this argument is not used, then the default is `null`. In this example, `disable` is being set to 1. When retrieving the value in the scoreboard, the `read_by_name()` function is used. The first two arguments of the function are the same. The value is being looked up by the scope and the name. The value that is retrieved is stored in the `disable_sb` *bit* through an *inout* function argument. The `read_by_name()` function also returns a *bit* value indicating whether or not the read was successful.

## III. WHEN IS THE CONFIGURATION DATABASE USED?

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. With the *uvm_config_db*, the user not only can add an object to the database, but can also specify, with great detail, the level of access to retrieval by specifying the hierarchy.

The classic example of *uvm_config_db* usage is with sharing a virtual interface. A SystemVerilog interface is instantiated at the top level of the testbench and connects to the ports of the device under test (DUT). For the UVM testbench to be able to drive or monitor this interface, it needs to have access to it. This extremely common scenario is where the *uvm_config_db* proves to be very useful. The various interface instantiations can be added to the database with the access level controlled, since it can then be retrieved by the appropriate component only if it is in the specified hierarchy.

Virtual interfaces are not the only use for the configuration database. Any object can be stored and retrieved. The previous example of disabling a scoreboard could have just as easily been done with the configuration database as opposed to the resource database. Other common uses of the configuration database include sharing configuration objects or setting whether an agent is active or passive.

## IV. HOW IS DATA STORED AND RETRIEVED?

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*

First, let's explore the set function and its arguments.

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

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

Note that all the methods of the *uvm_config_db* class are static so they **must** be called with the scope resolution operator, as is the case with the *uvm_resource_db*. Once again, type parameterization is used so the actual type for the resource, `T`, must be given. Also noteworthy, the default parameter type of the *uvm_resource_db* is `uvm_object`, whereas the default type for the *uvm_config_db* is `int`.

The figure above shows that the `set()` function has four arguments. These arguments are often a little confusing at first so let's define each.

TABLE II. *UVM_CONFIG_DB* SET METHOD

| 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`. |

Let's return to the virtual interface scenario. An interface has been instantiated in the top level and now needs to be added to the *uvm_config_db* using the `set()` function. The most basic way to do this is to use the `set()` function and allow the virtual interface to be widely accessible from anywhere within the testbench. The following figure illustrates the `set()` function used in this way.

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

The first argument is the context (`cntxt`) which is the starting point of the lookup search. The example uses `uvm_root::get()` to acquire the top-level so the search will start at the top of the hierarchy in this case. Normally `"this"` would be used as the context if the call to `set()` is within a class, but to set the virtual interface correctly in the database, the code has to be placed inside a module, so there would be no class context. The second argument is the instance name. In this example, the interface is being made globally available amongst all components so the wildcard, `"*"`, is used. The third argument is the field name which is a label used for lookup. Finally, the value argument is the actual instance of the interface.

In most cases, you do not want to make a database entry globally available. Since a global scope is essentially a single namespace, it makes reuse more difficult if everything is stored in this scope. To restrict its access, use the hierarchical path in conjunction with the wildcard character as shown below:

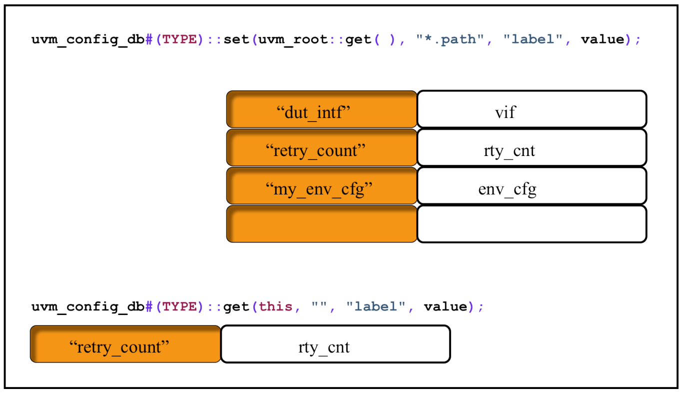
```
uvm_config_db#(TYPE)::set(this, "*.path", "label", value)
```

Adding other objects into the *uvm_config_db* is just as straightforward as adding a virtual interface. The important thing to remember is that each entry needs a unique field name or label (if the global scope is being used), or the path needs to be limited in such a way that non-unique labels do not conflict as the scopes are now limited to specific areas of the naming hierarchy

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. The figure below shows that the `get()` function is quite similar to the `set()` function. The notable differences are the return value, *bit* as opposed to *void*, and the value argument is an *inout* so that the value can be retrieved (and left unchanged if the lookup fails).

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

The context (`cntxt`) is the starting point for the search. The instance name in this case can be an empty string since it is relative to the context. The field name is the label given when the object was added to the database. The value argument is the variable that the retrieved value is assigned to. The following diagram illustrates retrieving a value from the database.



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

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

In the 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 call to the `get()` function. 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.

## V. AUTOMATED RETRIEVAL OF CONFIGURATION DATE IN THE BUILD PHASE

So far, we have described how to set and retrieve values in the configuration database using the `set()` and `get()` methods, but one question that is often asked with respect to retrieving data is: do you always have to explicitly call the `get()` function? The short answer is that it depends. In the UVM, there are mechanisms to automate the retrieval of data from the configuration database. In order to have the resource automatically retrieved two things must happen:

- First, that resource has to be registered with the factory using the field automation macros [1].
- Second, `super.build_phase(phase)` must be called in the `build_phase()` function.

When the object is created, the UVM factory will then check to see if there is an entry in the *uvm_config_db* for the registered resource. If there is an entry, then the default value will be overridden by what is in the database. Let’s take a look at a code example based on [2].

```
class pipe_agent extends uvm_agent;
  protected int  my_param = 10;


  pipe_sequencer sequencer;
  pipe_driver    driver;
  pipe_monitor   monitor;

  `uvm_component_utils_begin(pipe_agent)
   `uvm_field_int(my_param, UVM_ALL_ON)
   `uvm_component_utils_end

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(is_active == UVM_ACTIVE) begin
      sequencer = pipe_sequencer::type_id::create("sequencer", this);
         driver = pipe_driver::type_id::create("driver", this);
    end

    monitor = pipe_monitor::type_id::create("monitor", this);

    `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    if(is_active == UVM_ACTIVE)
       driver.seq_item_port.connect(sequencer.seq_item_export);
       driver.a_control_knob = my_knob; // only set in active mode
      `uvm_info(get_full_name( ), "Connect stage complete.", UVM_LOW)
  endfunction: connect_phase
endclass: pipe_agent
```

In the code above, “`my_param`” is registered using the field macro `` `uvm_field_int `` and the default value is `10`. If we are building an active agent (with `is_active` set to `UVM_ACTIVE`), then the knob is passed into the driver. Once the entry is in the database, the agent can be created using the factory. When the `build_phase()` method of that agent is called, “`my_param`” will be updated to the value that is retrieved from the configuration database, without the user having to do an explicit `get()` in the agent.

The usual place to set the entries in the configuration database is either the testcase or in the environment. Test-specific values are best set in the testcase, but structure-related values (such as active versus passive) are best set in the environment class. This is an example of a `set()` called from the testcase:

```
uvm_config_db#(int)::set(this, "env.agent", "my_param", 888)
```

The `is_active` property of the base class (`uvm_agent`) will also be set automatically, but see section VII for a caveat. One point to note, though, is to remember to call `super.build_phase(phase)` if there are properties in either the base or derived classes that need to be automatically retrieved from the configuration database (this is done using the `apply_config_settings()` method which is ultimately called from `uvm_component::build_phase()` ).

Not all projects choose to use the field automation macros. If this is the case, or if the user needs to determine whether or not the entry was added to the database before the object was created, then the `get()` method must be explicitly called.

## VI. HOW DO I DEBUG WHEN SOMETHING GOES WRONG?

As with any development project, mistakes will be made and debug will need to occur. This section covers debug techniques associated with the configuration database.

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 and from the databases, using `set()` and `get()` respectively.

For the resource database, the runtime option is specified as follows:

```
sim_cmd +UVM_TESTNAME=my_test +UVM_RESOURCE_DB_TRACE
```

The corresponding option for the configuration database is specified as follows:

```
sim_cmd +UVM_TESTNAME=my_test +UVM_CONFIG_DB_TRACE
```

Let's once again take an example from [2]. We will look at the registration and retrieval of the virtual interface required to connect the testbench to the device under test.

The interfaces are instantiated in the top-level testbench module:

```
module top;

   bit clk;
   bit rst_n;

   pipe_if ivif(.clk(clk), .rst_n(rst_n));
   pipe_if ovif(.clk(clk), .rst_n(rst_n));
```

Once we have the interface instances, we can add their virtual interface registrations to the configuration database. This is usually done inside an initial block:

```
initial begin
 uvm_config_db#(virtual pipe_if)::set(uvm_root::get(),
                                      "*.agent.*" , "in_intf", ivif);
 uvm_config_db#(virtual pipe_if)::set(uvm_root::get(),
                                      "*.monitor" , "out_intf", ovif);
 run_test( );
end
```

Note that because we're inside a module, we can't use `"this"` for the first argument of the `set()` calls. Instead, we call `uvm_root::get()` to establish the context. For the input interface, we make this available to any component with `"agent"` as a component of its name, but for the output interface, we only make this available to monitors by restricting the path to `"*.monitor"`.

In each case, as the configuration database entry is type-parameterized to the interface type, we use the actual instance of the interfaces, *ivif* and *ovif*, as the entry we register as these are valid "types" for the corresponding virtual interface types to reference.

Now to retrieve the virtual interface, say in the driver, we must do a `get()` on the configuration database:

```
class pipe_driver extends uvm_driver #(data_packet);
…
 function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   if(!uvm_config_db#(virtual pipe_if)::get(this, "", "in_intf", vif))
     `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
                          get_full_name( ), ".vif"})
  `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
 endfunction
```

The first level of debugging, as shown above, is to always do a test to see if the call to the configuration database's `get()` function succeeds or not. If it fails, we issue a fatal error via the `` `uvm_fatal `` macro and bring simulation to a halt. This is because if we haven't successfully retrieved a virtual interface handle, then the testbench cannot drive the DUT and so there's no point in continuing.

If we run a test with `+UVM_CONFIG_DB_TRACE` and look for the calls to `set()` we see the following in the transcript file:

```
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/SET] Configuration
'uvm_test_top.env.penv_in.agent.is_active' (type int) set by uvm_test_top.env
= (int) 1
UVM_INFO @ 0: reporter [CFGDB/SET] Configuration
'uvm_test_top.env.penv_out.agent.is_active' (type int) set by
uvm_test_top.env = (int) 0
```

The first line above shows that we set a virtual interface of type pipe_if with the string entry `"*.agent.*.in_intf"`. Since we set this from an initial block, the context is just shown as `"(virtual pipe_if) ? "`. We can also see that the `is_active` fields of our agents are set to active (int value 1) and passive (int value 0) by observing the final two lines above. Note that in these cases, we do see the context where the calls to `set()` took place as being `uvm_test_top.env`.

Now if we look at the corresponding debug messages from calls to `get()`, we see this:

```
UVM_INFO @ 0: reporter [CFGDB/GET] Configuration
'uvm_test_top.env.penv_in.agent.driver.in_intf' (type
virtual interface pipe_if) read by
uvm_test_top.env.penv_in.agent.driver = (virtual interface
pipe_if) ?
```

This shows that the driver instantiated in our agent was able to retrieve the virtual interface of type `pipe_if` and instance name `in_intf`.

Now let's suppose a typo was made in the string name of the interface instance we're trying to retrieve in the driver:

```
class pipe_driver extends uvm_driver #(data_packet);
…
 function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   if(!uvm_config_db#(virtual pipe_if)::get(this, "", "i_intf", vif))
     `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
                          get_full_name( ), ".vif"})
  `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
 endfunction
```

Instead of `"in_intf"`, we have accidentally typed `"i_intf"`. If we now run a simulation without debugging turned on, then we see this:

```
UVM_FATAL @ 0: uvm_test_top.env.penv_in.agent.driver [NOVIF]
virtual interface must be set for:
uvm_test_top.env.penv_in.agent.driver.vif
```

Of course, this error message, whilst somewhat useful, doesn't really give much debugging information. If we now run with `+UVM_CONFIG_DB_TRACE` and then look in the logfile for messages associated with the pipe_if (since we know from the fatal error message that this is the area to look for), we get this:

```
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)
```

We can now see the `set()` calls took place properly, but the `get()` call in the driver of the penv_in agent failed. Furthermore, we can see the error in the string we were searching for:

`'uvm_test_top.env.penv_in.agent.driver.i_intf'`

versus

`'*.agent.*.in_intf'`.

Thus with judicious coding in our environment to throw errors where a failed lookup from the database would give an unusable environment coupled with using `UVM_CONFIG_DB_TRACE`, we can fairly readily pin-point any areas where we have to debug the contents of the database as well as locate where the calls to `set()` and `get()` took place.

## VII. Known Usage Problems With the UVM Config Database

There are a few known problems with the configuration database the first of which is with the automation of class properties that are of an enumerated type. Once pernicious example of this is with the `is_active` field of agents derived from `uvm_agent` base class.

Consider this code:

```
class pipe_agent extends uvm_agent;

  pipe_sequencer sequencer;
  pipe_driver    driver;
  pipe_monitor   monitor;

  `uvm_component_utils(pipe_agent)

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(is_active == UVM_ACTIVE) begin
      sequencer = pipe_sequencer::type_id::create("sequencer", this);
         driver = pipe_driver::type_id::create("driver", this);
    end

    monitor = pipe_monitor::type_id::create("monitor", this);

    `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    if(is_active == UVM_ACTIVE)
       driver.seq_item_port.connect(sequencer.seq_item_export);
      `uvm_info(get_full_name( ), "Connect stage complete.", UVM_LOW)
  endfunction: connect_phase
endclass: pipe_agent
```

If an agent is needed that is passive, (i.e., “`is_active`” should be `UVM_PASSIVE`), the user must first call the `set()` function in the environment class, and add “`is_active`” to the database with the value of `UVM_PASSIVE`. Once the entry is in the database, the agent can be created using the factory. When the `build_phase()` method of that agent is called, “`is_active`” will be updated to `UVM_PASSIVE`.

In the code above, “`is_active`” is a member of the base class, `uvm_agent`. However, the automatic `get()` of its state from the configuration database is done like this:

```
 virtual class uvm_agent extends uvm_component;
  uvm_active_passive_enum is_active = UVM_ACTIVE;

function void build_phase(uvm_phase phase);
    int active;
    super.build_phase(phase);
    if(get_config_int("is_active", active)) is_active =
uvm_active_passive_enum'(active);
  endfunction
```

Note that the `get()` is performed as parameterized by an `int`, not `uvm_active_passive_enum`. One expected use pattern to configure the pipe_agent in the example would be as follows:

```
uvm_config_db#(uvm_active_passive_enum)::set(this, "env.agent",
"is_active", UVM_PASSIVE)
```

However, this does not work due to the `get_config_int` call in the base class. Digging deeper into the UVM source code reveals that `get_config_int` is a function that looks like this (as defined in uvm_component.svh):

```
typedef uvm_config_db#(uvm_bitstream_t) uvm_config_int;
function bit uvm_component::get_config_int (string field_name,
                                            inout uvm_bitstream_t
value);


  return uvm_config_int::get(this, "", field_name, value);
endfunction
```

What this shows is that the configuration database access is parameterized by the `uvm_bitstream_t` type, not `uvm_active_passive_enum`. Now to add to the confusion, one of the three major simulators works correctly if the `set()` is performed using `uvm_active_passive_enum` as the type parameter, but the other two do not. However, all three work as correctly if the set is done as follows:

```
uvm_config_db#(int)::set(this, "env.agent", "is_active", UVM_PASSIVE)
```

This issue with using enumerated types in the configuration database is further manifested in cases where such class properties are registered with the `` `uvm_field_enum `` automation macro. Again, this works correctly in only one of the three major simulators. However, if one does a manual `set()` and `get()` using the correct enumerated type parameter, all three simulators work correctly. As to be expected, the type parameters for both the `set()` and `get()` must be identical.

In the previous section, we demonstrated the use of the `+UVM_CONFIG_DB_TRACE` option which had the effect of printing messages for each call to `set()` and `get()`. This works as expected for all cases where the `set()` and `get()` calls are performed manually. However, in the case where the `get()` is automated through the field macros, a `get()` of a member which succeeds in retrieving an entry from the configuration database is ***not*** printed out in the logfile. A print ***is*** present for the case where no matching entry is found! Given that there are many class properties that automatically try to get their value from the configuration database, this is possibly a good thing, but users may be surprised by just how many database lookups are performed. In a simultation of the test environment from [2], approximately 140 lookups report that they failed to find a matching entry in the database. Judicious use of grep and the name of the field that one is trying to `set()/get()` can still be useful in debugging problems, particularly as the biggest source of errors is a typo in the string name used.

## VIII. 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. The resource database can be thought of as a pool of global variables whereas the configuration database is structured hierarchically and is more suited to data that is related to the structure of the testbench itself.

We provided an overview of the API of both databases as well as example code of their usage. We then presented some debugging techniques were presented to allow the reader to understand what to look for when the databases fail to return the data a user is expecting. Finally, some known issues with using the configuration database with enumerated types were presented.

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.

## IX. ACKNOWLEDGEMENTS

The authors would particularly like to acknowledge the invaluable contribution of their colleague, Jonathan Bromley, for his insight, research and knowledge of the inner workings of the UVM class library, particularly for the section on debugging.

The versions of the simulators used were current at the time of writing of this paper and made use of the pre-compiled libraries of the UVM base class library version 1.1d.

Finally we would like to thank all our colleagues at Verilab who provided their time to review this paper and make suggestions to improve it.

## REFERENCES

[1] Universal Verification Methodology (UVM), www.accellera.org

[2] Vanessa Cooper, Getting Started with UVM: A Beginner’s Guide, 1st ed., Verilab Publishing, 2013