The Untapped Power of UVM Resources and Why Engineers Should Use the uvm_resource_db API

Clifford E. Cummings
Paradigm Works, Inc.
cliff.cummings@paradigm-works.com

Heath Chambers
HMC Design Verification
hmcdvi@msn.com

Mark Glasser
Elastics.cloud
mark.glasser@elastics.cloud

***Abstract*- The resource database has been part of UVM since its first release. It was built to replace the cumbersome set_config/get_config API for configuring testbenches. The set_config_* functions could only store integers, strings and object handles in an inefficient manner distributed across components of an OVM testbench.**

The newly added UVM resource database greatly expanded the old set_config/get_config API capabilities. It could store values of any type in a centralized database that could be accessed anywhere in a testbench. It was built with two interfaces, a low-level interface and a convenience layer called "uvm_resource_db." This interface provides access to most of the functionality of the resource database through convenient one-line calls. The uvm_resource_db API is simple to use and allows storage and retrieval by any module and UVM testbench class, including transactions and sequences.

Later, a second API was added, uvm_config_db API, to provide backward compatibility with the OVM set_config_* API. This provided a way for users of set_config/get_config to transition to the resource database. The uvm_config_db API was never intended as the primary interface to the resource database, yet in practice, it has become so. The problem is that the uvm_config_db API imposes the ridiculous restriction that only UVM testbench *components* can set and retrieve items stored in the resource database. Using uvm_config_db as the primary means to access the resource database has led to continued usage of awkward constructs such as p_sequencer and so-called virtual sequencers.

This paper will explain how UVM resources work and how to use the simple and powerful uvm_resource_db API to take full advantage of the UVM resources. This paper will also outline the shortcomings and misconceptions related to the uvm_config_db API and why engineers should quit using this *very flawed* API.

I. Introduction

OVM had **`set_config_int`**, **`set_config_string`**, and **`set_config_object`** APIs (collectively referred to as **`set_config_*`**) that served the purpose of configuring components in OVM testbenches but were relatively inefficient. UVM introduced a more efficient facility that includes a centralized UVM resource database to replace the older OVM **`set_config_*`** facility. Access to the new UVM resource database was accomplished using the **`uvm_resource_db`** Application Programming Interface (API).

To ease the transition from OVM to UVM, a "convenience" layer was added to UVM using an API that more closely mimicked the semantics of the older OVM **`set_config_*`** facility. The **`set_config_*`** API was rewritten in UVM in terms of **`uvm_config_db`**. This made the transition from OVM to UVM much smoother, as users could use the **`uvm_resource_db`** and **`set_config_*`** in the same testbench. The **`uvm_config_db`** API was intended to be a "transition" layer rather than a "convenience" layer and only included a subset of the capabilities available to users of the **`uvm_resource_db`** API.

Unfortunately, as of DVCon 2023, we estimate that more than 90% of UVM Verification Engineers are using the **`uvm_config_db`** API, ***which is the wrong API***. Engineers broadly use the wrong API because early UVM books and examples gave the flawed recommendation to use the inferior **`uvm_config_db`** API.

This paper will show the numerous limitations and complexities surrounding the **`uvm_config_db`** API and illustrate the simpler syntax and more powerful capabilities available using the **`uvm_resource_db`** API. UVM

Verification Engineers should plan to abandon the **uvm_config_db** API and embrace the more straightforward and powerful **uvm_resource_db** API.

A. The UVM Resources Database Intent - Summarized

The resources database was designed with several goals in mind:

- Enable virtual interfaces to be treated like other configuration items.
- Remove the restrictions on what types can be stored.
- Detach configuration from the component hierarchy and enable objects other than components to access the resource database.
- Provide a general-purpose mechanism for sharing data between entities.

II. get_full_name() -vs- this

Three commonly used constructs when accessing UVM database resources are the SystemVerilog keyword **this**, and the UVM function calls **get_full_name()** and **get_name()**.

The **this** keyword is a class handle to the class object that uses the **this** keyword. In other words, **this** is a handle called by a class object to access itself without regard to where the class is in a testbench hierarchy. It is important to remember that **this** is a class handle and not a string.

The **get_full_name()** method is a method that returns the full-path string to the calling object for objects derived from **uvm_object**. The returned value is a string-based hierarchical path and is not a class handle. The **get_full_name()** method is used by the **uvm_config_db** command to return the string that corresponds to the **this** class handle.

The **get_name()** method is a method that returns the string name of just the calling object and not the full path name to the object. It is a string name that points to the current object and is not a class handle.

get_full_name(), **get_name()** and **this** are often used together as UVM database command arguments, but they are not interchangeable. To summarize:

- **get_full_name()** - returns a *full-path string name* to the current object.
- **get_name()** - returns the *string name* of the current object but not the full path.
- **this** - returns a *full path class handle* to the current object.

III. The Resource Database

UVM does not have two resource databases, only one. **uvm_resource_db#()** and **uvm_config_db#()** are two different Application Programming Interfaces (APIs) for the same resources database. **uvm_config_db#()** is a wrapper around **uvm_resource_db#()** -- that is, **uvm_config_db#()** is derived from **uvm_resource_db#()**, and the **uvm_resource_db#()** is a layer on top of the low-level resources database (**uvm_resource_pool**). It is possible to dispense with both **uvm_config_db#()** and **uvm_resource_db#()** APIs and use the low-level **uvm_resource_pool** access methods. However, doing so is more verbose than using either of the interfaces, so we generally do not recommend working with the low-level resource-pool database API directly.

IV. Introduction to uvm_resource_db & uvm_config_db APIs

The **uvm_resource_db** class provides a simplified interface for UVM resources as described in the previous section and in the UVM Class Reference [6]. The **uvm_resource_db** interface has a simple set of commands that can replace multiple commands required for equivalent operations using the **uvm_resource_base** and **uvm_resource#(T)** classes.

All of the functions in the **uvm_resource_db** (and the **uvm_config_db**) are static and must be called using the **::** operator. All of the **uvm_resource_db#()** (and **uvm_config_db#()**) commands are parameterized with the default **#(type T uvm_object),** and the user replaces the **#(...)** type with the actual type to be stored or retrieved.

Because the **uvm_resource_db** and **uvm_config_db** APIs are both interfaces to the same database, any item put into the resource database using the **uvm_config_db#()** commands can be retrieved using the **uvm_resource_db#()** commands.

Important Note #1: The **uvm_resource_db** commands can retrieve any resource stored using *either* the **uvm_config_db** or **uvm_resource_db** commands.

This also means you can use **uvm_resource_db#()** commands to put an item into the database using string scope values, based on component hierarchies, and retrieve the same item using **uvm_config_db#()** commands.

Users should understand that **uvm_resource_db#()** commands can also store items in non-component referenced locations, such as in UVM sequences, and those items can only be retrieved using **uvm_resource_db#()** commands. This offers many **uvm_resource_db#()** command advantages explained in this paper.

Important Note #2: Any resource stored with the **uvm_resource_db** commands that use a non-component scope cannot be retrieved using **uvm_config_db** commands. The **uvm_config_db** API is a subset of the **uvm_resource_db** API.

These notes are essential to understand because using the **uvm_resource_db#()** commands may be desirable to retrieve an item that another engineer stored using **uvm_config_db#()** commands. If you mix the **uvm_db** APIs, you must pay attention to the context and regular expression scope arguments described later in this paper.

The bottom line is that anything stored using **uvm_config_db#()** commands can be retrieved using **uvm_resource_db#()** commands, but not all items stored with **uvm_resource_db#()** commands can be retrieved using **uvm_config_db#()** commands. As will be shown in this paper, the **uvm_resource_db#()** API is more powerful and has a simpler syntax.

V. Storing UVM Resources using the uvm_resource_db API

UVM resources are typed extensions of the **uvm_resource_base** class. This section details the storing and retrieving of resources.

A. uvm_resource_pool & uvm_queue#(uvm_resource_base)

Each typed resource handle is stored in a pair of **uvm_queues** of **uvm_resource_base** class handles. One **uvm_queue** handle is stored in a **string**-indexed associative array called the Name Table, and another **uvm_queue** handle is stored in a **type**-handle-indexed associative array called the Type Table.

The Name Table and Type Table associative arrays are declared and maintained inside a singleton **uvm_resource_pool**, which is automatically created at the beginning of a UVM test.

The block diagram for the singleton **uvm_resource_pool** with both Name Table and Type Table is shown in Figure 1, and it should be noted that:

- The tables do *NOT* store resources directly; the tables are associative arrays that store handles to queues.
- Each **uvm_queue** entry stores **uvm_resource_base**-type class handles.

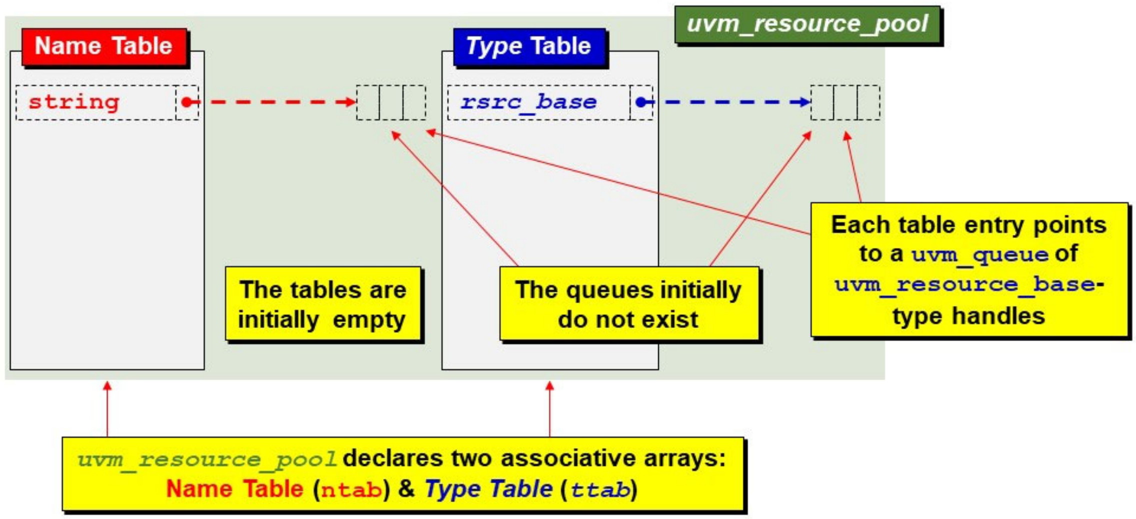

Figure 1 - uvm_resource_pool Block Diagram

Each **uvm_resource** is a type-specialized derivative of the non-typed **uvm_resource_base** class. Since each resource handle is an extension of the **uvm_base_class** type, they can be copied into the **uvm_queues** in the associative arrays. Assigning a typed resource into a queue containing base class handles is an *upcast* operation. When each resource object is retrieved, UVM does a **$cast** (*downcast*) operation to convert it back to the correct type-specialized **uvm_resource** class handle.

B. uvm_resource_db#()::set Details

Consider the following **uvm_resource_db** command, with type **virtual dut_if**, string-name **"vif"**, wild-card scope string **"*agnt*"** and the value is the **dif** dut interface handle.

```
uvm_resource_db#(virtual dut_if)::set("*agnt*", "vif", dif);
```

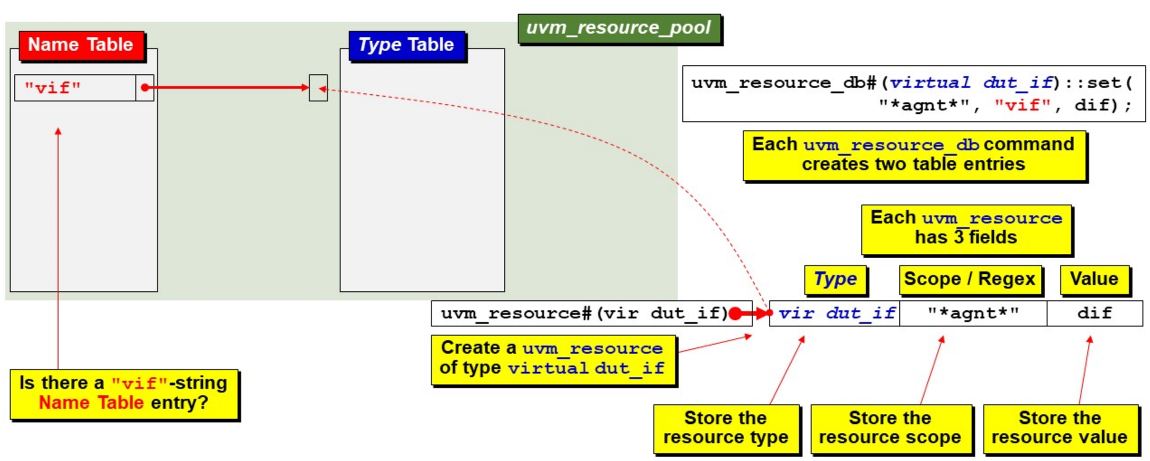

Figure 2 - uvm_resource_db Name Table storage action

UVM first creates the new typed **uvm_resource#(virtual dut_if)**. Then UVM checks to see if there is already a **"vif"** string entry in the string-indexed Name Table. When UVM recognizes that there is no **"vif"** entry, it then creates a new **uvm_queue** and pushes the **uvm_resource#(virtual dut_if)** handle onto the queue and stores the queue handle in the Name Table associative array at the string location, **"vif"**.

Whenever a **`uvm_resource`** is created, UVM stores three items in the resource: (1) the resource type, (2) the *resource scope* (which is a regular expression that can contain wildcards), and (3) the resource's value.

The *resource scope* is a somewhat misleading term. The *scope* is just a string. Using the **`uvm_resource_db`** API, the *scope* does *NOT* have to match an actual testbench component scope. The *scope* is just a string with wildcards that must be matched when retrieving a resource value using **`get`** or **`read_by_*`** commands.

Each **`uvm_resource_db::set()`** command creates both a Name Table entry, as described above, and a Type Table entry, as described **below**.

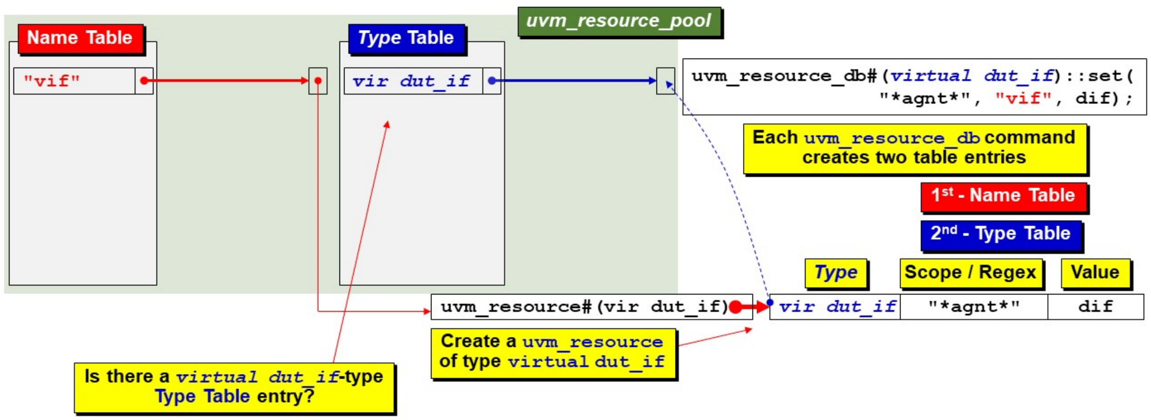

Figure 3 - uvm_resource_db Type Table storage action

After inserting the resource into the Name Table, UVM checks to see if there is already a **`virtual dut_if`** type entry in the type-handle-indexed Type Table. When UVM recognizes no **`virtual dut_if`** entry, it creates another new **`uvm_queue`** and pushes the **`uvm_resource#(virtual dut_if)`** handle onto the new queue. It then stores the queue handle in the Type Table associative array at the type-index location, **`virtual dut_if`**, as shown in Figure 3.

NOTE: There was only one new resource created, but its handle was made accessible from both the Name Table and the Type Table.

Now assume that the following **`uvm_resource_db`** commands have been executed:

This command was executed in Figure 2 and Figure 3.

```
uvm_resource_db#(virtual dut_if)::set("*agnt*", "vif",     dif);
```

The next two commands have been executed to add two new entries to the Name and Type Tables.

```
uvm_resource_db#(env_cfg)::set       ("*.e*",    "env_cfg", cfg,  this);
uvm_resource_db#(agnt_cfg)::set      ("*agnt1", "cfg",     cfg1, this);
```

The two preceding **`uvm_resource_db`** commands require that a pair of new **`uvm_queues`** be created to store the unique **`string`**-index names and **`type`**-index values.

Now let's add another **`uvm_resource_db`** command that reuses the existing Name Table **`"cfg"`** string-index (shown in Figure 4) and Type Table **`agnt_cfg`** type-index (shown in Figure 5). That is, we will create a new resource with the same string and type names as an existing resource.

```
uvm_resource_db#(agnt_cfg)::set      ("*agnt2", "cfg",     cfg2, this);
```

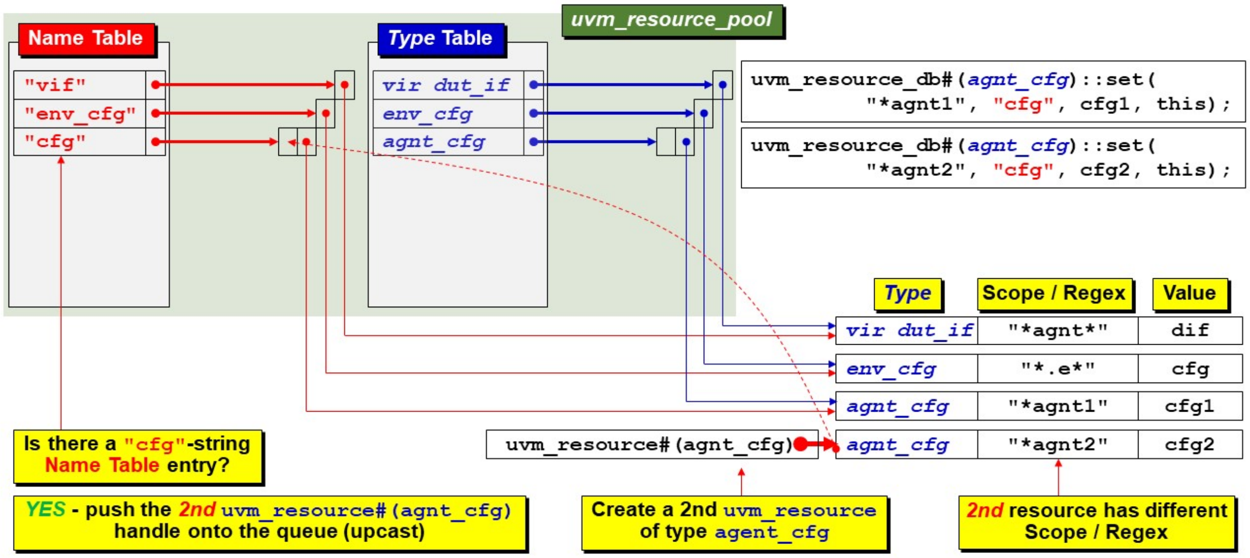

Figure 4 - uvm_resource_db Name Table new queue entry action

Since there was already a **"cfg"** string index in the Name Table, the new **uvm_resource#(agnt_cfg)** handle was pushed onto the existing queue pointed to by the **"cfg"** string index (shown in Figure 4).

And since there was already an **agnt_cfg** type index in the Type Table, the **uvm_resource#(agnt_cfg)** handle was pushed onto the existing queue pointed to by the **agnt_cfg** type index (shown in Figure 5).

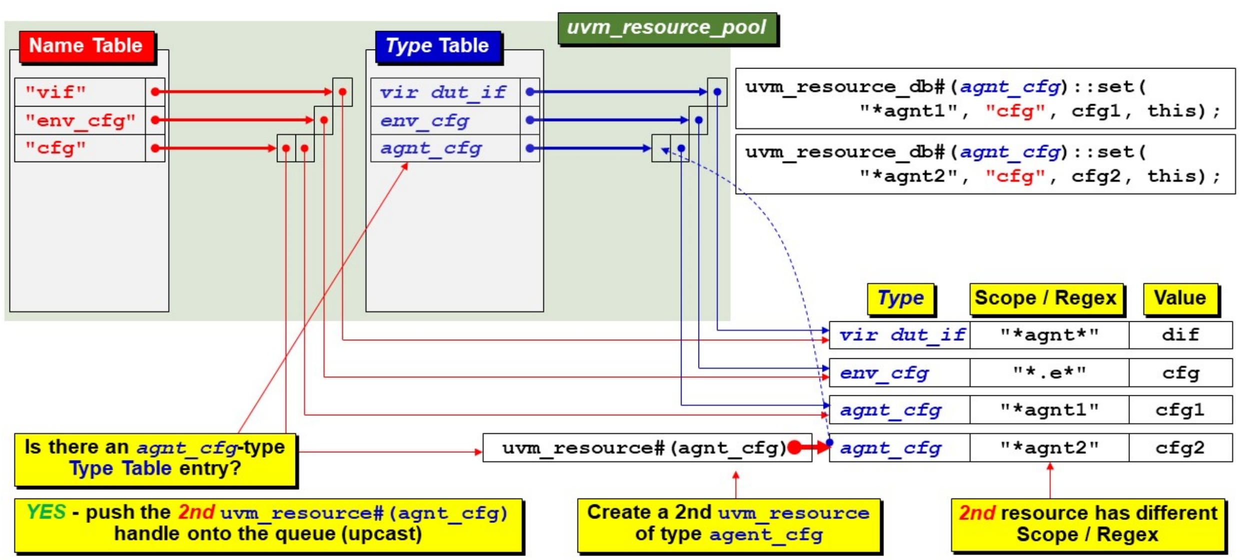

Figure 5 - uvm_resource_db Name Table new queue entry action

To continue this example, assume that four additional **uvm_resource_db** commands have been executed. These commands will create four new string-indexed queues for the Name Table and two new type-indexed queues for the Type Table.

```
uvm_resource_db#(int)::set    ("*",        "cnt",     4,       this);
uvm_resource_db#(int)::set    ("*.e*",     "has_cov", 1,       this);
uvm_resource_db#(string)::set("*agnt1",   "msg1",    "Warn1", this);
uvm_resource_db#(string)::set("*agnt2",   "msg2",    "Err2",  this);
```

C. uvm_resource_db Using a Pseudo Scope

Finally, let's execute a **uvm_resource_db** command to store a resource with a pseudo-scope (non-**uvm_component** scope) at the new string-index location **"LCNT"** of the Name Table and push the resource handle onto the existing **int**-type-index queue of the Type Table (shown in Figure 6).

```
uvm_resource_db#(int)::   set("LCNT::*", "LCNT",    10);
```

This last **uvm_resource_db** command would not be legal using a similar **uvm_config_db** command because **uvm_config_db** scopes must be a legal path to a **uvm_component**.

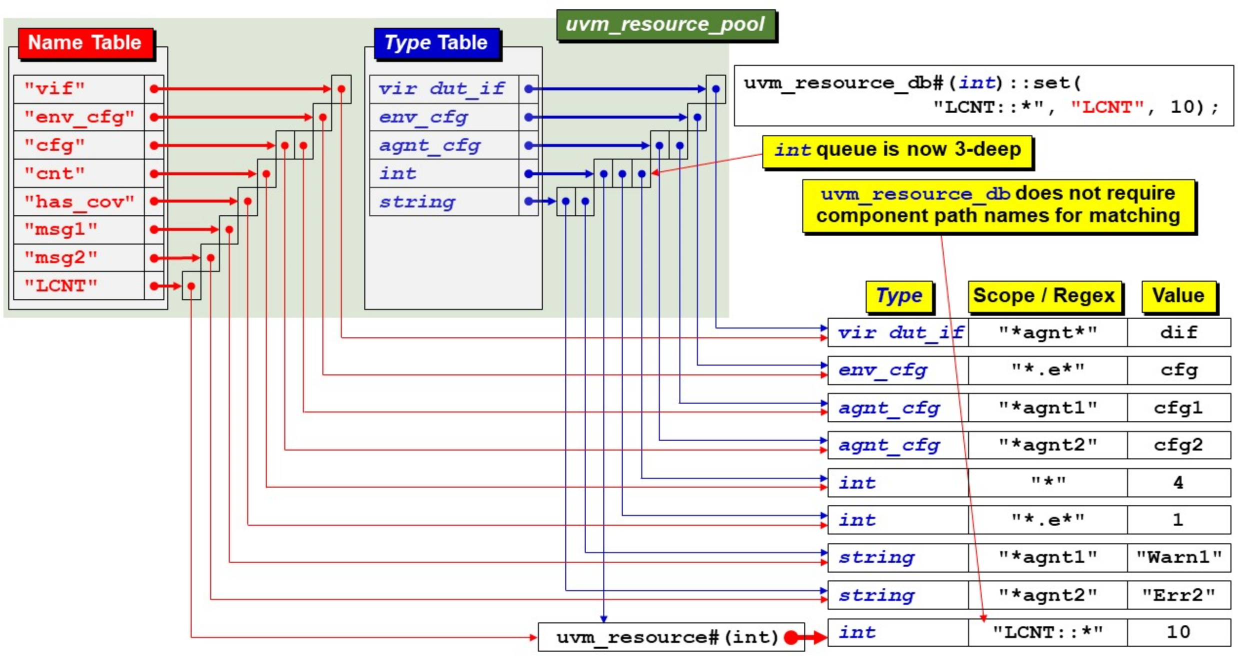

Figure 6 - uvm_resource_db Name Table new "LCNT" queue & Type Table push int-type queue entry

D. Pseudo Scopes

As previously mentioned, all of the *"scopes"* used by all UVM resource commands are strings. Since the *scope* arguments used by **uvm_resource_db** commands are just strings, they do not have to match an actual *scope path* to a real **uvm_component** in the component hierarchy. The only requirement for retrieving resources is that the regular expression used to **set** the resource must match the regular expression of the **uvm_resource_db read_by_name** or **read_by_type** commands.

This is both valuable and extremely useful. Since the *scope* is just a string that must be matched when accessed using **uvm_resource_db#()::read_by_name** or **uvm_resource_db#()::read_by_type** commands, items can be stored as resources and accessed directly by entities other than **uvm_components**, such as sequences and modules.

In his 2014 DVCon-India paper [5], Mark Glasser made the following observations and recommendations regarding pseudo-scope creation and naming conventions:

- Since non-hierarchical scopes do not have a natural naming scheme, we are free to invent one.
- Since *scopes* are not tied to the component hierarchy, any naming convention can be used for pseudo-scopes.
- It is essential to use a consistent naming convention amongst target scopes so that reasonable regular expressions can be used to identify them.
- Mark recommended using a common prefix and a separator unlikely to appear elsewhere in the target scope name.
- Mark recommended using the double colons (`::`) as the separator. Note that in this context the double colons do not have any special meaning. It is just a string that can easily be matched with a regular expression and is easily identifiable visually.

The **`uvm_resource_db`** command shown in Figure 6 used the pseudo scope **`"LCNT::*"`**. Any **`uvm_resource_db read`** command with a *scope* field that starts with the prefix **`"LCNT::"`** can match this pseudo scope.

E. Summarizing uvm_resource_db Storage Operations

The resource database, known as the *resource pool*, is organized as a pair of associative arrays: the Name Table, which stores resources by a string-name index, and the Type Table, which stores resources by a type-handle index. Each resource is always added to the Name Table and the Type Table such that either name-index or type-index **`uvm_resource_db`** commands can access the resource.

Adding a new entry to the database proceeds as follows:

Name Table
1) Look up the name index in the name table.
2) Get a handle to the queue for that name if it exists.
3) Else, create a new queue for that name and insert it in the name table.
4) Put the resource handle into the existing or new queue.

Type Table
5) Look up the type handle in the type table.
6) If it exists, get a handle to the queue.
7) Else, create a new queue for that type.
8) Put the resource handle into the existing or new queue.

Each resource with the same name-index or type-index is differentiated by its regular expression *scope* field.

VI. Name Table, Type Table & UVM Resources

After executing the nine **uvm_resource_db** commands shown previously, there are nine typed-**uvm_resources**, eight entries in the Name Table that point to the resources, and five entries in the Type Table that point to the same resources, as shown in Figure 7.

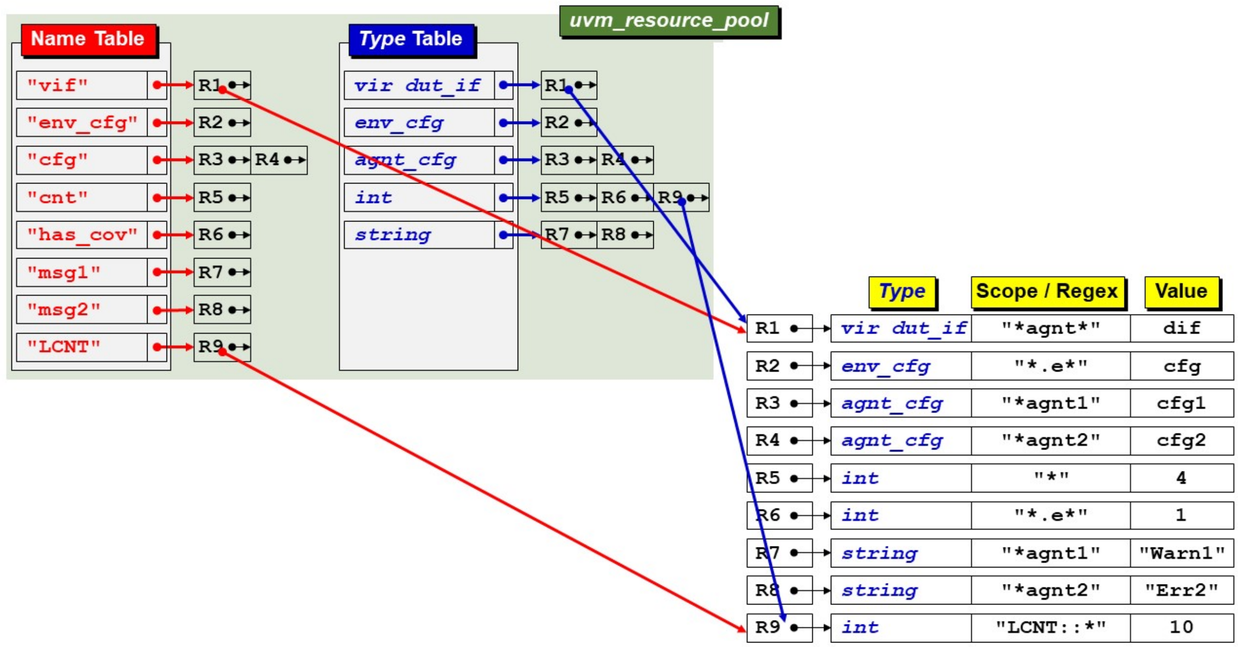

Figure 7 - Name Table, Type Table & UVM Resources

Each typed-**uvm_resource** has a handle that points to it by a Name Table entry and a separate Type Table entry. Using **uvm_resource_db** commands, each resource can be retrieved either from the Name Table, Type Table, or both.

VII. Retrieving UVM Resources using the uvm_resource_db API

Once items have been stored as resources, then components, sequences, sequence_items, and modules are able to access the resources and retrieve the stored values. These resources can be retrieved by name or by type.

Both **uvm_resource_db::set** and **uvm_config_db::set** commands store typed-**uvm_resource** handles into queues whose queue handles are stored in the Name Table and the Type Table.

The **uvm_config_db#()::get** command can only access the Name Table string-index values; it cannot access the Type Table type-handle-index values.

The **uvm_resource_db** can access string-index resource handles in the Name Table using **uvm_resource_db#()::read_by_name** commands, and can access type-handle-index values in the Type Table using **uvm_resource_db#()::read_by_type** commands.

A. uvm_resource_db#()::read_by_name Details

The first resource-retrieval technique is demonstrated using the **uvm_resource_db read_by_name** command.

It is common practice for an agent to retrieve a **virtual dut_if** handle from the resource database and store it locally. The agent frequently copies the retrieved **virtual dut_if** handle to its subordinate driver and monitor.

Figure 8 shows the essential steps to retrieve the **dut_if** handle. The agent would declare a **virtual dut_if** handle; in this example, the handle has been named **vif**. When first declared, the **vif** handle points to **null**, so later, the agent code calls the **uvm_resource_db#(virtual dut_if)::read_by_name** command to retrieve the stored **virtual dut_if** handle.

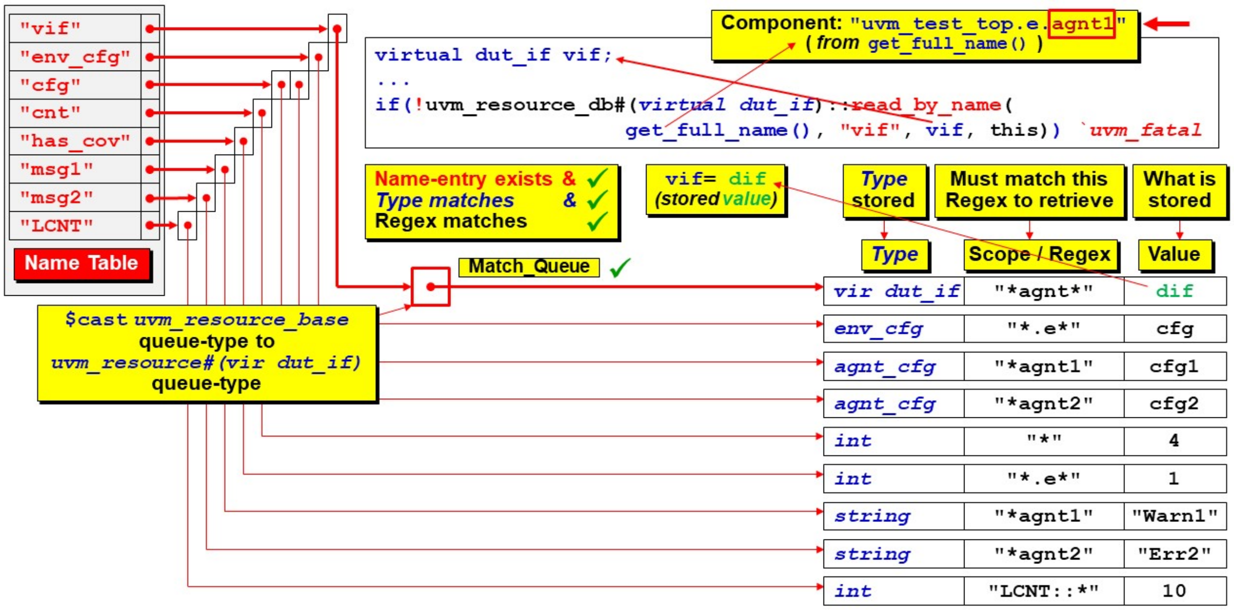

Figure 8 - uvm_resource_db#(virtual dut_if)::read_by_name example & Pseudo Scope Regex matching

The **read_by_name** command attempts to access a Name Table entry with the string-index **"vif"**. If the **"vif"** string entry exists, the **uvm_queue** for the entry is **$cast** to a **uvm_queue#(virtual dut_if)**. This **$cast** is a *downcast* operation. The queue is then traversed to extract all queue handles that match both the **virtual dut_if** type and that can match the **"*agnt*"** regular expression. The **uvm_resource_db** command creates a matching scope by calling UVM's built-in **get_full_name()** method that returns a full-path string of the calling component, which in this example is **"uvm_test_top.e.agnt1"**. This name will wild-card match **"*agnt*"**.

Each matching entry is placed into a *Match_Queue*. The stored value from the top entry in the *Match_Queue* is returned and stored in the **vif** handle.

Note: The UVM resources facilities provide a way to add priority weighting and a way to push matching queue entries to the top of the *Match_Queue*, but those mechanisms are rarely used and not described in this paper. The user can refer to the UVM Reference manual if such mechanisms are required. Engineers generally control what is placed in the *Match_Queue* by using uniquely crafted matching scopes.

The **if**-test ensured that a valid **virtual dut_if** handle was returned. For proper testbench implementation, any accessed resource must have already been stored as a typed-**uvm_resource**. The **if**-test traps missing resource errors that could otherwise be **null**-pointer references, which can be exceptionally difficult to debug. Every **uvm_resource_db#()::read_by_name** or **uvm_resource_db#()::read_by_type** command returns status to indicate if the command was successful (**1**) or not (**0** or **null**), and each resource access should be checked with an **if**-test. If not successful, it is common practice to issue a **`uvm_fatal** command, especially if cascading, catastrophic failures would happen in the test if the resource was missing. The **if**-test can save hours of debugging time.

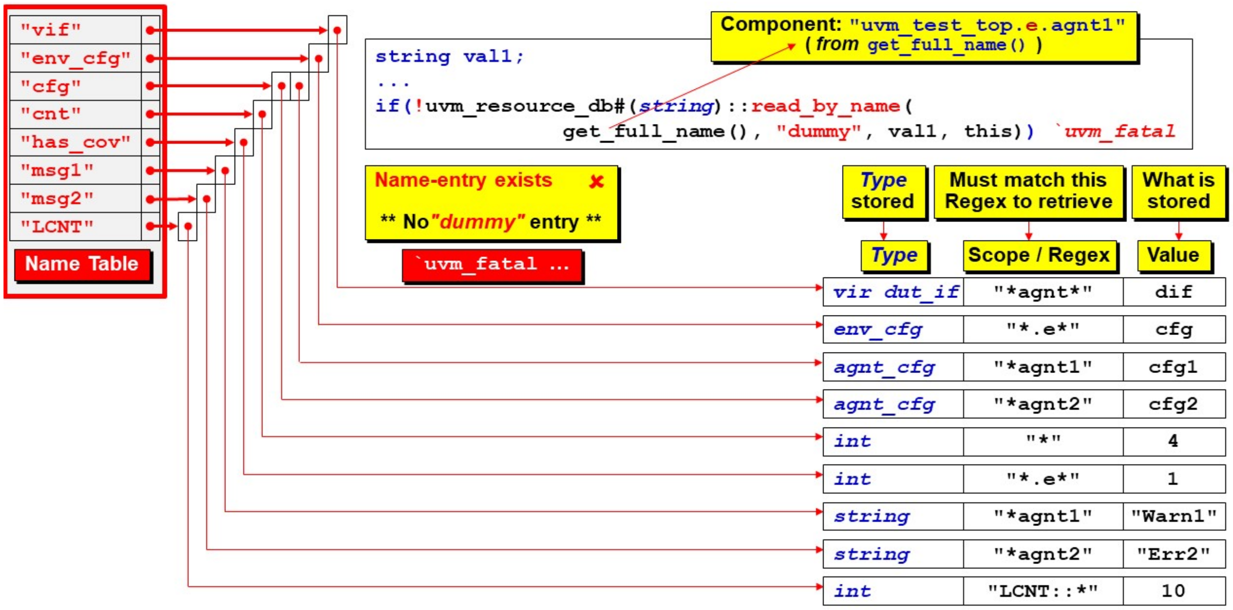

Figure 9 - uvm_resource_db#(string)::read_by_name - No existing Name Table entry

Consider what happens when one tries to access a non-existent Name Table entry, as shown in Figure 9. The **`uvm_resource_db#()::read_by_name`** command attempts to retrieve a resource handle that presumably was stored at the string-index **`"dummy"`**. The **`read_by_name`** command will fail, the **`if`**-test will detect the failure and execute a **`` `uvm_fatal ``** macro to print a failure message and abort the simulation.

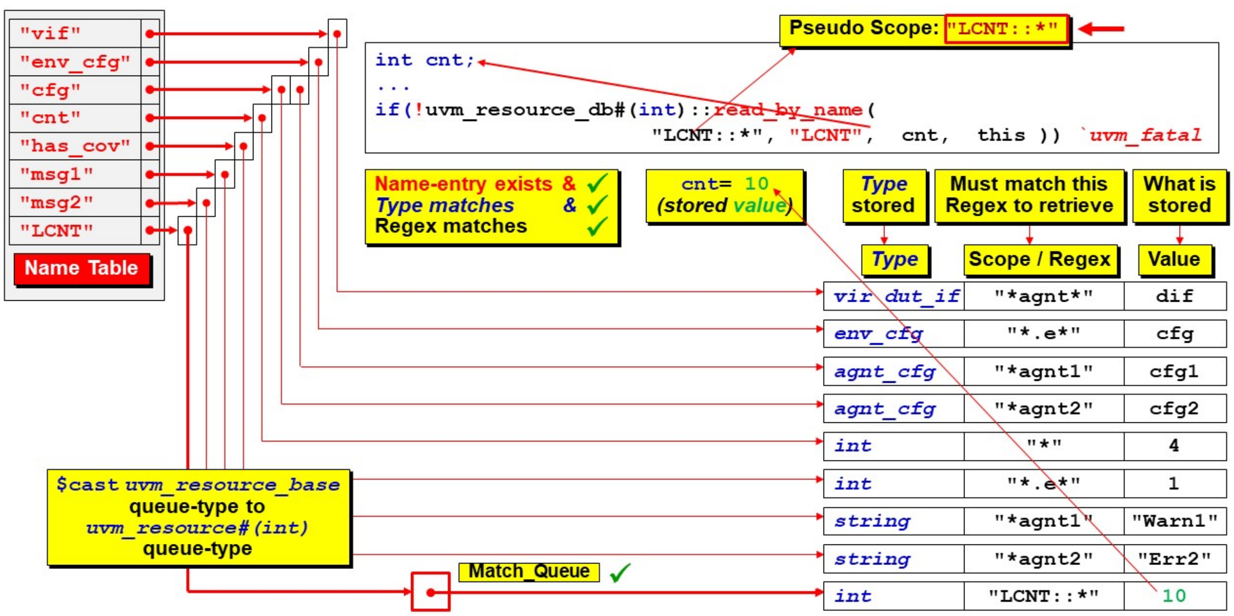

Figure 10 - uvm_resource_db#(int)::read_by_name example & Pseudo Scope Regex matching

In Figure 10, a **`uvm_resource_db`** command is used to access a resource stored at the **`"LCNT"`** string-index location in the Name Table. This entry includes the pseudo scope string **`"LCNT::*"`** that does not point to any component in the UVM testbench. Perhaps a sequence base class needs to retrieve a Loop Count set by the **`top`** module or top-level environment. Passing information from the **`top`** module or one of the testbench components is very simple when using **`uvm_resource_db`** commands with pseudo scopes. This is one of the outstanding advantages that **`uvm_resource_db`** commands have over **`uvm_config_db`** commands.

B. uvm_resource_db#()::read_by_type Details

The second resource-retrieval technique to be demonstrated uses the **`uvm_resource_db read_by_type`** command.

The **`virtual dut_if`** described in the previous section can also be retrieved by its type without knowing where it is stored in the Name Table.

Figure 11 shows the essential steps to retrieve the **`dut_if`** handle. Once again, the agent would declare a **`virtual dut_if`** handle; in this example, the handle has been named **`vif`**. When first declared, the **`vif`** handle points to **`null`**, so later, the agent code calls the **`uvm_resource_db#(virtual dut_if)::read_by_type`** command to retrieve the stored **`virtual dut_if`** handle, this time from the Type Table.

The **`read_by_type`** command attempts to access a Type Table entry with type-handle-index **`virtual dut_if`**. If an entry for the **`virtual dut_if`** type-handle exists, the **`uvm_queue`** for the entry is **`$cast`** to a **`uvm_queue#(virtual dut_if)`**. This **`$cast`** is a *downcast* operation. The queue is then traversed to extract all queue handles that match both the **`virtual dut_if`** type and that can match the **`"*agnt*"`** regular expression. The **`uvm_resource_db`** command created a matching scope by calling UVM's built-in **`get_full_name()`** method that returns a full-path string of the calling component, which in this example is **`"uvm_test_top.e.agnt1"`**. This name will wild-card match **`"*agnt*"`**.

Each matching entry is placed into a *Match_Queue*. The stored value from the top entry in the *Match_Queue* is returned and stored in the **`vif`** handle.

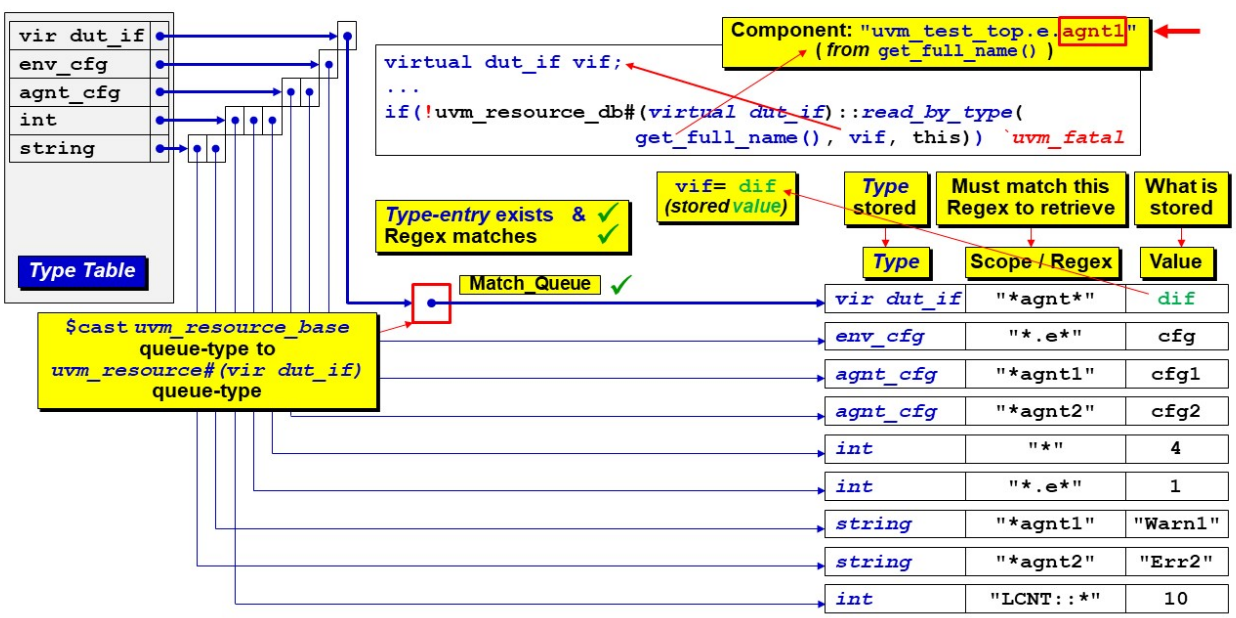

Figure 11 - uvm_resource_db#(virtual dut_if)::read_by_type example & Regex matching

The last argument, **`this`**, is used for audit tracing when debugging the creation and access of resources. The default for this argument is **`null`**, which does no audit tracing. The recommended usage for this audit flag is to use **`null`** when retrieving a resource into a non-class, such as a **`module`**, and use **`this`** inside all classes to enable class-based audit tracing.

Now consider a **uvm_resource_db** command that references a Type Table entry that has a queue with multiple entries, as shown in Figure 12.

```
string msg1;
…
if (!uvm_resource_db#(string)::read_by_type(get_full_name(), msg1, this))
        `uvm_fatal …
```

The **read_by_type** command accesses the Type Table entry with **string** type-index. The **uvm_queue** for the entry is **$cast** to a **uvm_queue#(string)**. This **$cast** is a *downcast* operation. The queue is then traversed to extract all queue handles that match both the **string** type and that can match the **"*agnt2*"** regular expression. The **uvm_resource_db** command created a matching scope by calling UVM's built-in **get_full_name()** method that returns a full-path string of the calling component, which in this example is **"uvm_test_top.e.agnt2"**. This name will not match the **"*agnt1*"** scope of the first **string**-type resource but will wild-card match **"*agnt2*"** of the second **string**-type resource.

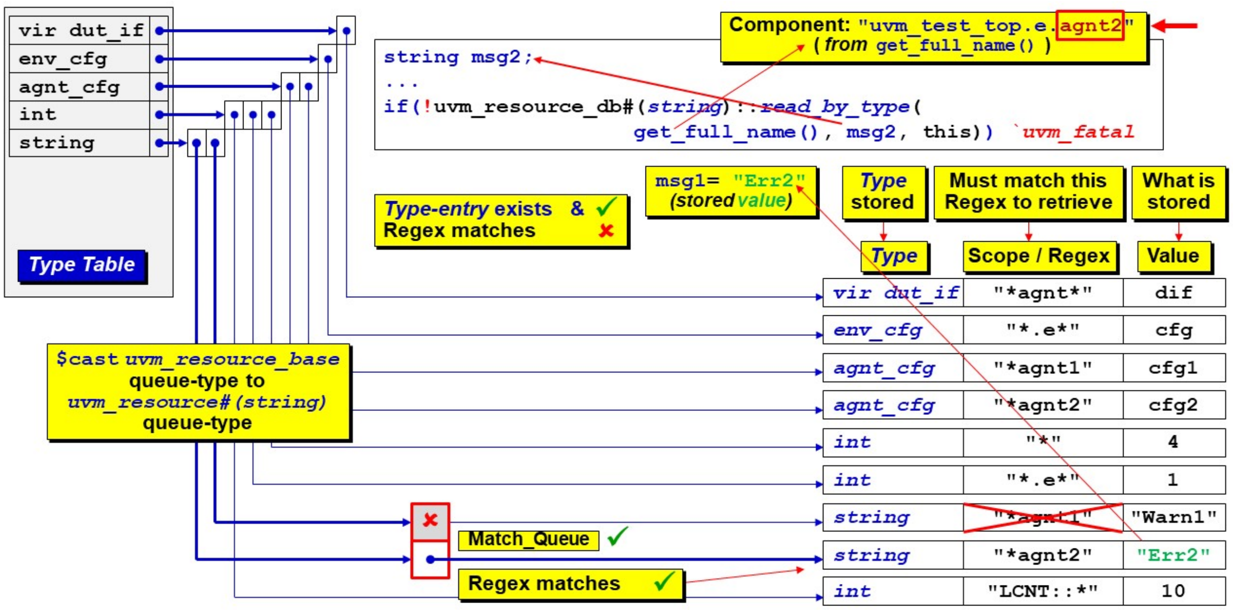

Figure 12 - uvm_resource_db#(string)::read_by_type example & Regex matching

If an engineer knows the type of the stored resource, and if the resource is easily distinguished with a unique match-scope, the **uvm_resource_db::read_by_type** command is a straightforward syntax that can be used to retrieve a value from a resource. There is no equivalent **read_by_type** capability using the **uvm_config_db** API.

VIII. Storing UVM Resources using the uvm_config_db API

There is a second resources API that most UVM engineers frequently use and partially understand, called the **uvm_config_db** API. The **uvm_config_db#()** class definition is included the UVM Base Class Library (BCL), in the file **src/base/uvm_config_db.svh**. The **uvm_config_db** class is a derivative of the **uvm_resource_db** class as shown below:

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

The **uvm_config_db** API imposes additional **set**-method scoping requirements and has a limited subset of the **uvm_resource_db** read capabilities.

Using the **uvm_resource_db** API, the **set** and **read_by_*** commands simply stored a scope-string and test for a matching scope-string when retrieving a resource value. As will be seen in this section, the **uvm_config_db** API requires additional testing and setting of scope-strings to make sure they correspond to the full-path string of an existing component. This means that the **uvm_config_db** API is slightly less simulation efficient than the **uvm_resource_db** API, plus it means that sequences and modules cannot set or access resources using the **uvm_config_db** API.

A. UVM uvm_config_db Command & Usage

Defined in this same **uvm_config_db.svh** source file is the definition for the static **set** method shown in Figure 13.

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

Figure 13 - Prototype of the uvm_config_db static set method

Also defined in this same source file is the definition for the static **get** method, shown in Figure 14.

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

Figure 14 - Prototype of the uvm_config_db static get method

For both the static **set** and **get** methods, the last two arguments (**string field_name, input** or **inout T value**) are reasonably well understood by most UVM users. The **string field_name** is the string address indicating where the variable will be stored in a string-based Name Table associative array (it is just the storage address for **set** and **get** commands). The **T value** is the typed value of the stored variable or the name of a declared properly typed variable that the **get** command will declare to hold the retrieved value.

The first two arguments (**uvm_component cntxt, string inst_name**) can confuse many new and experienced UVM users.

The first two arguments form a path to one of the extended **uvm_component** classes in the user's UVM testbench. The first argument (**cntxt**) must be a **uvm_component**-derivative handle, not a **string**. The second argument (**inst_name**) must be a **string**, not a handle. The UVM source code does a **cntxt.get_full_name()** to return the full-path-handle-string name to the referenced **cntxt**-handle, then generally concatenates the full-path-handle-string to the inst_name string to form a full-path string to the referenced component. Since the inst_name string can contain wildcard characters, the full-path string frequently contains wildcarded paths.

The full-path string is the scope-string that is stored in a UVM resource. The full-path string does not indicate where the resource is stored. The full-path string is literally *just a string* that must be matched when a uvm_config_db get command attempts to retrieve the stored value in a resource.

B. UVM uvm_config_db Class Set/Get Definitions

To fully understand the uvm_config_db ::set and ::get methods, one also needs to realize that there is a uvm_root class (in the uvm_root.svh file) extended from the uvm_component class that declares the following singleton uvm_root handle:

```
const uvm_root uvm_top = uvm_root::get();
```

Figure 15 - The const uvm_root uvm_top declaration

In the same uvm_root.svh file is this snippet of uvm_root constructor code that will call the uvm_component new() constructor with the unique string "__top__" and parent null.

```
function uvm_root::new();
  super.new("__top__", null);
  …
endfunction
```

Figure 16 - uvm_root new() constructor code

Included in the uvm_component.svh file is the snippet of uvm_component new() constructor code, shown in Figure 17.

```
 1 function uvm_component::new (string name, uvm_component parent);
  …
 2   uvm_root top;
 3   uvm_coreservice_t cs;

 4   super.new(name);

 5   // If uvm_top, reset name to "" so it doesn't show in full paths then return
 6   if (parent==null && name == "__top__") begin
 7     set_name(""); // *** VIRTUAL
 8     return;
 9   end

10   cs = uvm_coreservice_t::get();
11   top = cs.get_root();
  …
12 endfunction
```

Figure 17 - uvm_component new() constructor code

In Figure 17, lines 2-3 & 10-11 retrieve the one and only (singleton) handle to the uvm_root class object, which has the full handle name top. After line 11 is executed, top and uvm_top are equivalent handles in the UVM testbench that point to the singleton const uvm_root uvm_top object, shown in Figure 15.

Lines 4-9 define the uvm_component new() constructor and this constructor checks the exception condition that is present when uvm_root calls super.new("__top__", null); After this new() constructor completes, there will be a singleton uvm_root object with handle name top (and uvm_top) and the top object has had its get_full_name() return value set to an empty string "", which happened on line 7.

C. uvm_config_db::set source code details

Now moving to the `uvm_config_db.svh` source file, the top-module will typically call the `uvm_config_db::set()` method with the first two arguments, `null` (`cntxt`) and `"*agnt"` (`inst_name` -or- some other path-string). So by the time the `::set` method is called, the following values exist:

- `cntxt=null`
- `inst_name="*agnt"`

The `set` method includes the following snippet of implementation code:

```
1 uvm_root top;
…
2 uvm_coreservice_t cs = uvm_coreservice_t::get();
…
3 top = cs.get_root();
…
4 if(cntxt == null)     cntxt     =  top;
5
6 if(inst_name == "")   inst_name =  cntxt.get_full_name();
7 else if(cntxt.get_full_name() != "")
8                       inst_name = {cntxt.get_full_name(), ".", inst_name};
```

Figure 18 - uvm_config_db set() method code

Before walking through the description of this code, remember for our top-module example, the `::set` `inst_name="*agnt"` input argument is not an empty string.

Lines 1-3 retrieve the one and only (singleton) handle to the `uvm_root` class object, which has the full handle name `top`. After line 3, `top` and `uvm_top` are equivalent handles that point to the singleton `uvm_root` object.

Line 4 shows that if the `set` command argument is `cntxt=null`, `cntxt` will be set to `top` / `uvm_top`.

Line 6 checks to see if the `set` command argument `inst_name=""`, and `inst_name` will be set to either the `cntxt` argument string-name (if not `null`) or will be set to the full string name of `uvm_top`, which is `""` (if `cntxt=null`). For our top-module example, the `inst_name` is not an empty string, so this line of code will not execute.

Lines 7 & 8 are executed if the retrieved `cntxt` string value is not `""` and sets the final `inst_name` to the full-path string name starting at the specified non-null `cntxt` component followed by `.inst_name` (a string). For our top-module example, the `cntxt` string is an empty string, so lines 7-8 will not execute.

For our top-module example, the original `"*agnt"` passed as the `inst_name` input argument will remain unmodified and is the final `inst_name` argument.

D. uvm_config_db::get source code details

We now move on to the `uvm_config_db::get` method to see how items are retrieved. Let's consider a typical `tb_agent` action that retrieves the virtual interface from the `uvm_config_db` and then stores the retrieved handle into a `virtual dut_if vif` handle.

The ubiquitous command used to retrieve the `vif` handle is the following:

```
if(!uvm_config_db#(virtual dut_if)::get(this,"","vif", vif)) <… call `uvm_fatal…>
```

For this **get** command:

- **cntxt=this**
- **inst_name=""**

Examining the **uvm_config_db get()** method code shown in Figure 19 and using the previous argument values (**this**, **""**), line 3 will not execute, and line 4 will execute and get the final **inst_name** to the full-path-name of **this** component. Lines 5-6 will not execute. For our simple example, the **vif** handle declared in **this** component will be set to point to the **vif** handle set by the **uvm_config_db::set** command used in the top-module example.

```
1 uvm_coreservice_t cs = uvm_coreservice_t::get();
2
3 if(cntxt == null)     cntxt     =  cs.get_root();
4 if(inst_name == "")   inst_name =  cntxt.get_full_name();
5 else if(cntxt.get_full_name() != "")
6                       inst_name = {cntxt.get_full_name(), ".", inst_name};
```

Figure 19 - uvm_config_db get() method code

E. Preferred Usage Observations

It is worth making a few preferred-usage observations:

(1) In the UVM testbench **top** module, a **uvm_config_db** command often stores a **virtual** interface handle before even calling the UVM **run_test()** command. At this point in the simulation, there is no UVM testbench hierarchy. This command is often called with arguments **null** (**cntxt**) and a wildcard path (**inst_name**), which frequently specifies any path to an agent-handle (**agnt**) component. The **null** keyword argument is recognized by UVM and converted into the **uvm_top** handle. In general, **uvm_config_db** commands called from a module scope (not a class scope) will use the **cntxt** handle **null,** and the **inst_name string** will be a wildcard to one of the components that will be factory-constructed during the UVM **build_phase()**. The **inst_name** will not be the empty string **""**, but must be a string path even if it is just the wildcard string **"*"**.

(2) When using the **uvm_config_db** commands from inside a class, the first argument is typically the keyword **this** (a handle to this class object no matter where the component object is located inside the UVM testbench). If the variable type is **set** or **get**-retrieved in this class scope, the **inst_name** is frequently the empty string **""** because the full-path string references something inside **this** constructed component.

(3) When a component attempts to set or retrieve a variable in a subcomponent or a config object, the component typically still sets the **cntxt=this** to reference itself as the starting point of the full-path string and then uses the subcomponent handle string instance name or config object handle name to complete the full-path string where the required variable will be **set** or **get**-retrieved.

IX. Example uvm_config_db Commands and their uvm_resource_db Replacements

This section serves as a quick-tip-sheet to show how UVM verification engineers can replace common **uvm_config_db** commands with simple and efficient **uvm_resource_db** commands.

*A. Top-module **set** commands:*

- Existing **uvm_config_db::set** command #1:
  `uvm_config_db#(virtual dut_if)::set(null, "*", "dut_if", dif);`
- Replace with **uvm_resource_db::set** command #1:
  `uvm_resource_db#(virtual dut_if)::set("*", "dut_if", dif);`

- Existing **uvm_config_db::set** command #2:

```
uvm_config_db#(virtual dut_if)::set(null, "*agnt", "dut_if", dif);
```

- Replace with **uvm_resource_db::set** command #2:

```
uvm_resource_db#(virtual dut_if)::set("*agnt", "dut_if", dif);
```

B. *Agent component **set** commands:*

- Existing **uvm_config_db** command #3:

```
uvm_config_db#(agnt_config)::set(this, "", "cfg", cfg);
```

- Replace with **uvm_resource_db** command #3a:

```
uvm_resource_db#(agnt_config)::set(get_full_name(), "cfg", cfg, this);
```

- *OR* -

- Replace with **uvm_resource_db** command #3b:

```
string scope = get_full_name();
uvm_resource_db#(agnt_config)::set(scope, "cfg", cfg, this);
```

- Existing **uvm_config_db** command #4:

```
uvm_config_db#(agnt_config)::set(this, "drv", "cfg", cfg);
```

- Replace with **uvm_resource_db** command #4a:

```
uvm_resource_db#(agnt_config)::set({get_full_name(), ".drv"}, "cfg", cfg, this);
```

- *OR* -

- Replace with **uvm_resource_db** command #4b:

```
string drv_scope = {get_full_name(), ".drv"};
uvm_resource_db#(agnt_config)::set(drv_scope, "cfg", cfg, this);
```

C. *Agent component **read_by_*** command (Part 1):*

If top module **set** command was one of the following:

```
uvm_config_db#(virtual dut_if)::set(null, "*", "dut_if", dif);
```

- *OR* -

```
uvm_resource_db#(virtual dut_if)::set("*", "dut_if", dif);
```

Agent component **read_by_*** command(s) should be:

- **uvm_resource_db::read_by_name** command #1:

```
uvm_resource_db#(virtual dut_if)::read_by_name(get_full_name(),"dut_if",vif, this);
```

- **uvm_resource_db::read_by_type** command #2:

```
uvm_resource_db#(virtual dut_if)::read_by_type(get_full_name(), vif, this);
```

D. *Agent component **read_by_*** command (Part 2):*

If agent component **set** command was one of the following:

```
uvm_config_db#(agnt_config)::set(this, "", "cfg", cfg);
```

- *OR* -

```
uvm_resource_db#(agnt_config)::set(get_full_name(), "dut_if", dif);
```

- *OR* -

```
string scope = get_full_name();
uvm_resource_db#(agnt_config)::set(scope, "cfg", cfg, this);
```

Agent component **read_by_name** command should be:

- **uvm_resource_db read_by_name** command #1:

```
uvm_resource_db#(agnt_config)::read_by_name(get_full_name(), "cfg", cfg, this);
```

- **uvm_resource_db read_by_type** command #2:

```
string scope = get_full_name();
uvm_resource_db#(agnt_config)::read_by_name(scope, "cfg", cfg, this);
```

X. Avoiding p_sequencer by Using the uvm_resource_db API

Sequences cannot easily access resources using the `uvm_config_db` API because the `uvm_config_db` API was really designed to only work with components.

When engineers need to pass testbench information to a sequence, one common technique is to use the `` `uvm_declare_p_sequencer() `` macro to create a `p_sequencer` handle. Since sequences are started on a sequencer, sequences have a handle to the sequencer where they are running, and anything stored in that sequencer is now accessible to the sequence. The sequence can retrieve any stored value that may have been declared and stored in that sequencer.

If you trust that you have not made a mistake, you can access the sequencer handle using the built-in `m_sequencer` handle that is set every time a sequence is started on a sequencer. Using the `p_sequencer` handle, created using the `` `uvm_declare_p_sequencer() `` macro, is fully vetted and therefore, a safer alternative. This is one of the primary ideas behind using virtual sequencers [2].

Engineers who use the more advanced `uvm_resource_db` API to store and retrieve resource information can completely bypass `` `uvm_declare_p_sequencer() `` macro and `p_sequencer` handle usage altogether. Using the `uvm_resource_db` API with pseudo-scopes (non-component-path strings) a verification engineer can store any information required by a sequence into the resource database from modules and UVM testbench components and retrieve it directly into the sequence. With the `uvm_resource_db` API, there is no need to pass information through a sequencer so the `` `uvm_declare_p_sequencer() `` macro and `p_sequencer` handle are unnecessary.

XI. OVM set_config_* / get_config_* Commands

Although useful in OVM, the `set_config_*` and `get_config_*` commands were deprecated from the UVM standard but are still supported by vendors for backward compatibility. This interface has two key restrictions that severely reduce its utility. One is that it supports only three data types (integers, strings, and class handles). The other is that it only works within components.

A. set_config_* / get_config_* Examples

Figure 20 shows examples of `set_config_*` commands:

- The `set_config_int("*",…)` command sets a `cnt` integer to the value of `2` for every component in the OVM testbench.
- The `set_config_string("*e",…)` command sets an `sqr1` string to the value `"agnt.sqr"` for just the `env` component in the OVM testbench.
- The `set_config_object("*agnt",…)` command sets a `dif_w` object handle to point to the `dif_w` handle defined in the `top` module and does so for just the `tb_agent` component in the OVM testbench. The last `0` argument specifies that this is just a handle to the existing `dif_w` class object and not a handle to a cloned copy of the class object.

```
set_config_int    ("*",     "cnt",   2);
set_config_string ("*e",    "sqr1",  "agnt.sqr");
set_config_object ("*agnt", "dif_w", dif_w, 0);
```

Figure 20 - OVM - set_config* examples

All set values are shown in the block diagram of Figure 21.

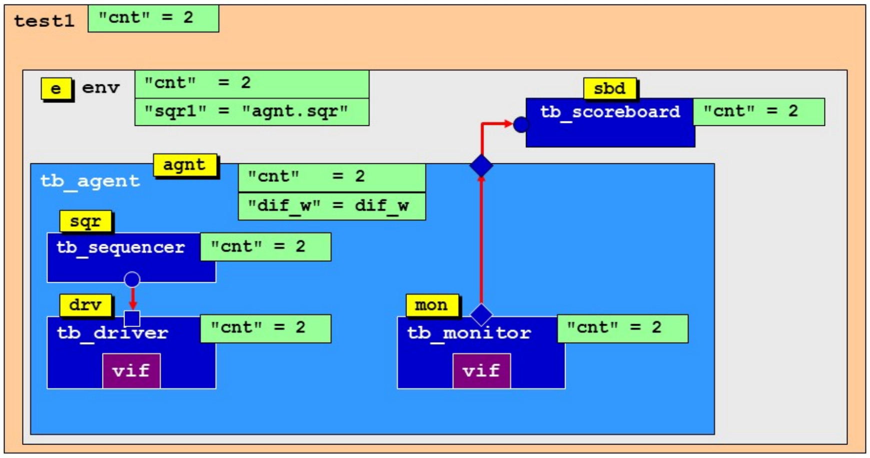

Figure 21 - OVM Block Diagram - shows variable assignments to components in the OVM testbench.

Figure 22 shows examples of `get_config_*` commands, and since each `get_config_*` command is a function that returns a status indicating if the `get`-operation was successful (non-`0` value) or unsuccessful (`0`-value), each should be `if`-tested and, if unsuccessful, execute one of the following message macros, `` `ovm_info ``, `` `ovm_error ``, or `` `ovm_fatal ``, with corresponding behaviors:

- The `get_config_int(…)` command will retrieve the `cnt` integer value from any component in the OVM testbench.
- The `get_config_string(…)` command retrieves the `sqr1` string value but only does so if the command is executed from the `env` component. No such string value is available from any other component in the OVM testbench.
- The `get_config_object(…)` command retrieves the `dif_w` object handle but only does so if the command is executed from the `tb_agent` component. No such object handle value is available from any other component in the OVM testbench. The last `0` argument specifies that this is just a handle to the stored `dif_w` class object and not a handle to a cloned copy of the class object.

```
if (!get_config_int("cnt", cnt))
    `ovm_info ("NOINT", "NO cnt value",          OVM_HIGH)

if (!get_config_string("sqr1", sqr1))
    `ovm_error("NOSTR", "NO sqr1 string value",  OVM_HIGH)

if (!get_config_object("dif_w", obj, 0))
    `ovm_fatal("NOVIF", "NO dif_w handle found", OVM_HIGH)
```

Figure 22 - OVM - get_config* examples

Due to their inefficient storage model and limited capabilities, we recommend transitioning away from the older OVM-style `set_config_*` commands and adopting the newer UVM Resources Database commands. The `set_config*` / `get_config*` interface has been deprecated and has been removed from the IEEE UVM 1800.2 Standard [3].

B. set_config_* / get_config_* is the Reason for the uvm_config_db API

If the `uvm_config_db` API was never intended to be the primary resources API, why does it even exist? Was it a mistake to add the `uvm_config_db` API to UVM?

It was not a mistake to add the **uvm_config_db** API to UVM. It was necessary to provide backward compatibility with earlier OVM **set_config_*** / **get_config_*** commands. The alternative was to support two entirely incompatible means of configuring testbenches, the resource database AND the **set_config**/**get_config** facility. Had UVM developers left it that way most people would never have switched from **set_config**/**get_config**. The mistake was using and promoting the **uvm_config_db** API as the primary interface to the resource database.

XII. Resource database read-functions and testing

All of the **uvm_resource_db** and **uvm_config_db read_by_***/**get** commands are functions that return a status bit indicating if the **read**/**get** operation was successful. This status bit should ALWAYS be tested because an unsuccessful **read**/**get** command almost always causes failures, which are hard to detect and difficult to debug.

The following are guidelines regarding handling the returned **read**/**get** status bit:

- Never **void'()** cast the return bit. Doing a **void'()** cast is a legal way to discard the returned status bit, but that status bit should never be discarded.

- Do not use an **assert** statement to test the returned status bit. There are SystemVerilog Assertion (SVA) commands to disable assertions, and a disabled assertion coded as part of a resource database command disables the resource database command and the retrieval of the resource database variable. Disabling assertions can turn a passing test into a failing test, which can be difficult to debug.

- When a **read**/**get** command fails, the returned status bit is **0**. Use an **if (! uvm_resource_db#()…)** command to detect a failing **uvm_resource_db** (or **uvm_config_db**) command and report a **`uvm_fatal** message or a **`uvm_error** message to help rapidly debug the problem. If retrieving a resource database variable would cause a catastrophic and obscure test failure where the test could not do any subsequent productive testing, use the **`uvm_fatal** message.

- It is an unfortunate common practice to code the id-string of **`uvm_fatal**/**`uvm_error** messages as either **get_full_name()** or **get_type_name()**. This practice can make it more difficult to debug huge verification environments, especially if there are multiple resource database **read**/**get** commands in the same component. The printed output can be very verbose when using **get_full_name()** and non-intuitive when using **get_type_name()**, especially if there are multiple print commands in the same class and multiple copies of that class type used in a huge test environment. Adding short, unique names (perhaps even the same short name) is recommended.

XIII. POSIX Regular Expressions and Globs

The low-level interface to the resources database supports both regular expressions and globs. By extension, **uvm_resource_db** supports both. However, **uvm_config_db** only supports globs.

The low-level interface to the resource database assumes that the scope argument is a glob unless you surround it with slashes. For example, **top.*** is a glob, **/top.*/** is a semantically equivalent proper regular expression. If the slashes are present, the underlying UVM library will strip the slashes and return the string. Otherwise, it will do a conversion. Table 1 shows a short comparison of glob meta-characters versus equivalent regular expression meta-characters.

glob	regex
.	\.
?	.
*	.*

Table 1 - Meta-Character Conversion from Globs to Regular Expressions

The globs used with **`uvm_config_db`** commands are a reasonable subset of regular expressions. Still, there are times when the true regular expressions offer enhanced wildcard access to pieces of the **`uvm_resource_db`** referenced resources.

For an expanded description and additional examples using regular expression access and glob access in a UVM testbench, see Mark's DVCon 2014 India paper [5].

XIV. Debugging uvm_resource_db operations

Because all resource database operations are global, it is often difficult to trace buggy operations back to the offending resource database command. The following debug facilities are available to aid in debugging UVM resource database operations.

A. uvm_resource_db Tracing Facility

When debugging **`uvm_resource_db`** operations, there is a very convenient runtime **`+UVM_RESOURCE_database_TRACE`** option that will report all resource database write and read operations. The output from this command can be rather verbose, but it is easily runtime-enabled and disabled. Sometimes this tracing capability is the easiest way to find resource database access problems. There is an equivalent **`uvm_config_db`** runtime tracing option: **`+UVM_CONFIG_database_TRACE`**.

B. uvm_resource_db Dumping Facility

Dumping a database so you can see what it contains is the most obvious debugging tool for any database. The resource pool class provides a **`dump()`** function to do just that. The function is made accessible in the **`uvm_resource_db`** interface as **`uvm_resource_db#(T)::dump()`**. Each resource in the database is printed along with its scope regular expression and all its access records.

C. uvm_resource_db Auditing Facility

The term *auditing*, as used with the **`uvm_resource_db,`** refers to tracking the different variety of *set* and *get* operations on portions of the **`uvm_resource_db`**. Auditing is possible when an **`accessor`** field is used in the **`uvm_resource_db`** commands. Setting the **`accessor`** field to **`this`** allows the auditing capabilities to report which class objects called the **`uvm_resource_db`** commands. If the **`accessor`** field is left blank, then the accessor handle keeps the default value of **`null,`** and tracking information for that command is practically useless.

Typical practice is to add **`this`** as the accessor field of **`uvm_resource_db`** commands used in classes to allow tracking when enabled for debugging purposes. Omitting the accessor field of the **`uvm_resource_db`** command is perfectly legal, but it eliminates useful debugging information for that command if tracking is turned on (enabled).

XV. UVM Resource Efficiency & Usage Recommendations

As was described in the previous sections, the storage of general-purpose resources is a compute-intensive operation. The **`uvm_pool`** singleton includes two associative arrays. The associative arrays have pointers to multiple **`uvm_queues`** of **`uvm_resource_base`** class handles that are dynamically created during the simulation as needed. Each resource is created as a type-specialized **`uvm_resource`**.

Also, as described in previous sections, retrieval of general-purpose resources is another compute-intensive operation. Each **`read_by_name`** or **`read_by_type`** command must do a lookup from the corresponding Name Table or Type Table associative array; then they must walk through all of the queued **`uvm_resource_base`** class handles for the index value (frequently, there is only one class handle, but each queue could have multiple class handles), then try to match the resource **`type`** field, and try to match the resource **`scope`** field (using wild-card DPI-C function calls). Each matched queue item is pushed onto a dynamically created match-queue, another queue of the `uvm_queue#(uvm_resource_base)` type, and either selects and returns and stores in a separate variable (the accessed resource value from the match-queue if matches exist) or returns a fail status that should be `if`-tested when the `uvm_resource_db` command is called. If no match exists, the `if`-test should often report a `` `uvm_fatal `` or `` `uvm_error `` message.

Since storage and retrieval are compute-intensive operations, usage of the resources database should largely be restricted to storing one-time setup and configuration information. Using the resources database for frequent run-time variable storage and retrieval is very simulation inefficient and not recommended.

XVI. uvm_resource_db & uvm_config_db Capabilities Summarized

The resource database `set` commands always create a single typed UVM resource and then stores the resource handle into both the Name Table and Type Table queues.

Resources handles in the Type Table queues are only accessible using the `uvm_resource_db read_by_type` command. There is no equivalent `uvm_config_db get`/`read`/`read_by_type` command. These capabilities and restrictions are shown graphically below.

A. *uvm_resource_db Capabilities*

Figure 23 shows that the `uvm_resource_db` API is used to store (`set`) resource references in both the Name Table and Type Table queues. The `uvm_resource_db` API can be used to set both component-based scopes and non-component-based scopes in the resources.

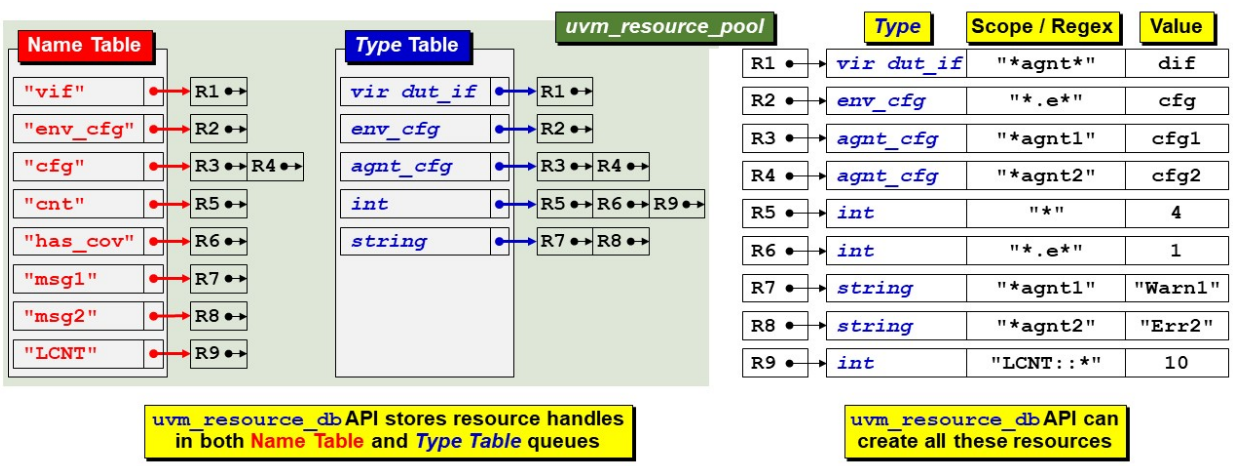

Figure 23 - uvm_resource_db can set component-scope and pseudo-scope resources that can be referenced from both Tables

Figure 24 shows that the **uvm_resource_db::read_by_name** command can be used to retrieve resource references from the Name Table, while the **uvm_resource_db::read_by_type** command can be used to retrieve resource references from the Type Table. The **uvm_resource_db** API can be used to match both component-based scopes and non-component-based scopes in the resources.

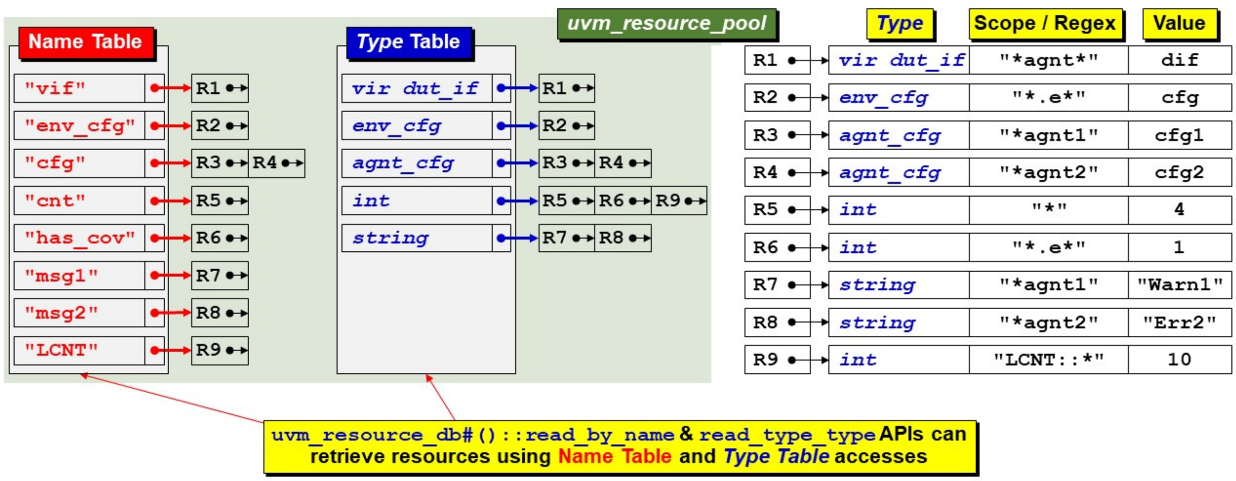

Figure 24 - uvm_resource_db can do both read_by_name & read_by_type

B. uvm_config_db Capabilities

Figure 25 shows that the **uvm_config_db** API is used to store (**set**) resource references in both the Name Table and Type Table. The **uvm_config_db** API is required to use component-based scopes. Non-component-based scopes are not permitted when using the **uvm_config_db** API.

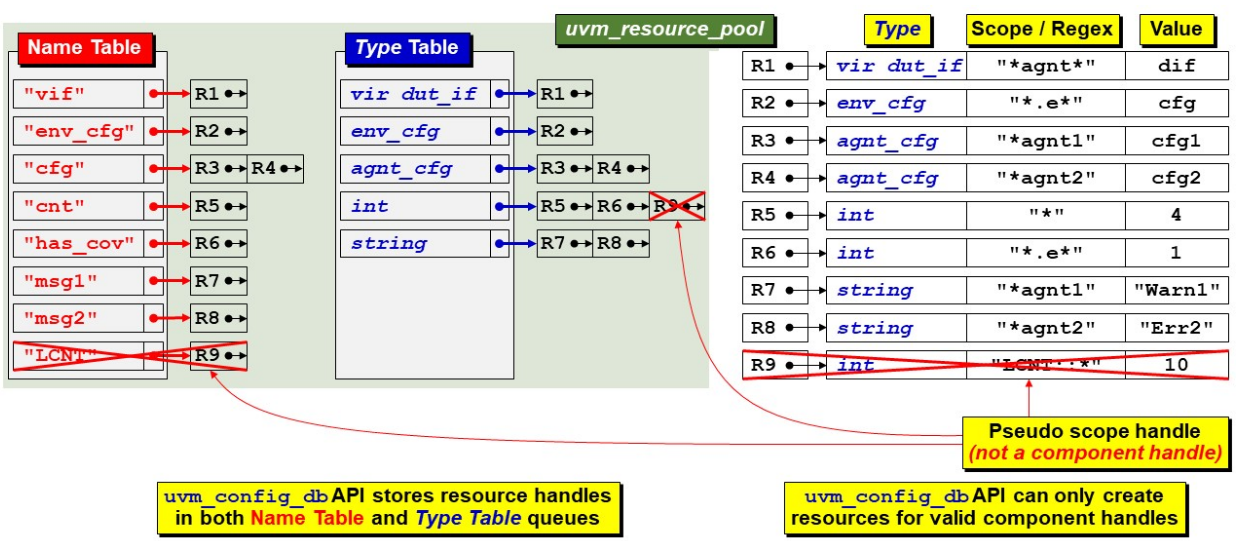

Figure 25 - uvm_config_db pseudo scope storage limitations

Figure 26 shows that the **uvm_config_db::get** command can only be used to retrieve resource references from the Name Table queues. There is no equivalent **uvm_config_db** command to retrieve resource references from the Type Table. The **uvm_config_db** API is required to use component-based scopes when retrieving a resource. Non-component-based scopes are not permitted when using the **uvm_config_db** API.

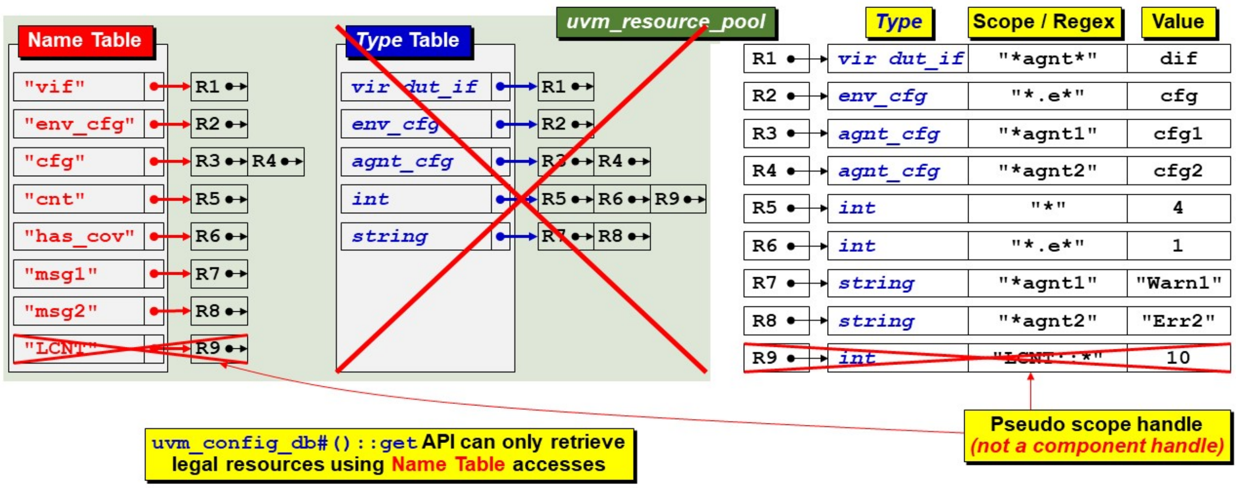

Figure 26 - uvm_config_db get limitations

XVII. User Experiences

The authors worked together on a large verification project. Mark Glasser is also one of the primary inventors of the **uvm_resource** and **uvm_resource_db** classes and methods, so Mark decided that our project would focus on using the **uvm_resource_db** API. Heath and Cliff were more familiar with the **uvm_config_db** API, so we were new to and skeptical about using the **uvm_resource_db** API.

Heath summed up his experience using the **uvm_resource_db**, coming from a **uvm_config_db** perspective.

There were several positive surprises regarding the use of the **uvm_resource_db** directly:

- It was much easier to use than many papers and other materials lead people to believe.
- The flexibility of not being tied to the component hierarchy.
- The ability to use it outside UVM classes (e.g., modules).

Things that took time to get used to:

- Changing from using **::get** to **::read_by_name** and **::read_by_type**.
- Remembering to leave off the **uvm_component cntxt** first argument of the method calls (including **::set**).

Things to watch out for or plan for:

- Set up a good "naming convention" for the scope argument of the method calls to avoid conflict of the name arguments between various calls to set and read different values.
  - If needing to use legacy code with **uvm_config_db** calls along with new code using **uvm_resource_db**, particular attention will need to be paid to setting up the scope argument for all **uvm_resource_db** methods that access items that **uvm_config_db** calls have either **::set** or **::get**. If the item was **uvm_config_db::set**, wildcarding could be used in the scope argument of the **uvm_resource_db::read_*** calls. The other way around is a much more difficult problem to **uvm_resource_db::set** an item for use by a **uvm_config_db::get** call in legacy code.

XVIII. Summary of Capabilities

To summarize the capabilities described in this paper, consider the following table of capabilities using OVM & UVM config commands.

	set config_*	uvm_config_db	uvm_resource_db
Used in OVM testbenches	✓	✗	✗
Used in UVM testbenches	✗[1]	✓	✓
Stores int / string / object data types	✓	✓	✓
Stores any data type	✗	✓	✓
Allows use of glob regular expressions	✓	✓	✓
Allows use of POSIX regular expressions	✗	✗	✓
Distributes stored information across components	✓	✗	✗
Stores information in a common resource database	✗	✓	✓
Requires complex component handle & string scoping	✗	✓	✗
Allows simple string scoping	✗	✗	✓
Can store & retrieve information by name	✗	✓	✓
Can store & retrieve information by type	✗	✗	✓
Can store & retrieve information into components	✗	✓	✓
Can store & retrieve information into sequences	✗	✗	✓
Can store & retrieve information into modules	✗	✗	✓

The `uvm_resource_db` commands have three primary capabilities not available using the `uvm_config_db` commands:

1. The ability to use the more expressive POSIX regular expression capability provides a fine-grained means for specifying the visibility of resources – i.e., which components, sequences, etc. have access to a resource.

2. The ability to store and retrieve information not only `by_name` but also `by_type` can simplify the retrieval process. This can be very useful in a large UVM testbench environment.

3. The ability to store information that can be directly accessed by sequences is one of the most compelling reasons to prefer `uvm_resource_db` commands over the continued use of `uvm_config_db` commands.

XIX. Conclusions

Quit using `set_config_*` / `get_config_*` commands - These commands were deprecated in UVM because they used a very inefficient storage model.

Quit using the `uvm_config_db` API - The `uvm_config_db` API lacks important features that simplify UVM testbench development, features that are available when using the `uvm_resource_db` API. The `uvm_config_db` commands also require the `cntxt`(component-handle)-`inst_name`(string) pair to specify the matching scope, which has proven to be confusing to many verification engineers.

The good news is that `uvm_config_db` code does not have to be removed from existing UVM testbenches. `uvm_resource_db` commands are fully backward compatible with `uvm_config_db` code so `uvm_resource_db` commands can work with all existing UVM testbenches.

USE the `uvm_resource_db` API - the `uvm_resource_db` syntax is easier than the `uvm_config_db` syntax and uses a simple-string scoping mechanism.

[1] Although deprecated from UVM, the `set_config_*` facility is still used in some UVM testbenches.

Using the **`uvm_resource_db`** API also simplifies the development of advanced UVM testbench techniques, such as:

- Virtual sequences [2] - **`uvm_resource_db`** makes subsequencer handles directly available to the virtual sequence base class [1].

- Parameterized MAX_IF techniques [4] - again, **`uvm_resource_db`** allows passing of DUT parameters from the top module to the UVM testbench without passing the parameters through a sequencer.

The **`uvm_resource_db`** API is by far the simplest, most powerful and preferred API to interact with UVM resources.

References

[1] Clifford E. Cummings, Heath Chambers, Mark Glasser, *"UVM Virtual Sequences The Easy Way (not the Hard Way ... or the other Hard Way!),"* SNUG-SV 2023, in press.

[2] Clifford E. Cummings, Janick Bergeron, *"Using UVM Virtual Sequencers & Virtual Sequences,"* DVCon 2016 Proceedings, also available at www.sunburst-design.com/papers/CummingsDVCon2016_Vsequencers.pdf

[3] "IEEE Standard For Universal Verification Methodology Language Reference Manual," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800.2™-2017

[4] Jeff Montesano, Paul Marriott, *"Parameterize Like a Pro - Handling Parameterized RTL in you UVM Testbench,"* DVCon 2020 Tutorial, also available at www.verilab.com/files/parameterize_like_a_pro_web_final.pdf

[5] Mark Glasser, *"UVM Resources Database: The Missing Manual,"* DVCon India 2014 silo.tips/download/configuration-in-uvm-the-missing-manual#

[6] Universal Verification Methodology (UVM) 1.2 Class Reference - June 2014