Configuration in UVM: The Missing Manual

Mark Glasser
NVIDIA Corporation
Santa Clara, CA 95050
mglasser@nvidia.com

***Abstract*—The UVM resources database facility is a very powerful tool for configuring testbenches. Its flexibility enables a wide range of use models. As a result, some confusion has arisen within the UVM community on configuration best practices. With apologies to David Pogue[1] we present details on how the configuration facility in UVM works and some use models for common configuration problems.**

Resources and the resources database were first introduced to the UVM community in [1]. That paper was written before *uvm_config_db* was written and added to UVM which is why that paper does not address the *uvm_config_db*. In this paper we will also look at the two primary interfaces to the resources database *uvm_resources_db* and *uvm_config_db* and compare and contrast them. We will demonstrate that the *uvm_config_db* is not necessary except as a backward compatibility layer for the now deprecated *set_config/get_config* configuration facility. Along the way we will dispel some myths regarding configuration that have been circulating within the UVM community for some time.

***Keywords*—testbench, UVM, configuration, resources.**

I. Introduction

Configuring a testbench is the act of supplying parameter values that control its operation and topology. A *configurable* testbench is imbued with degrees of freedom that can be controlled externally. A particular setting of the parameters that control the degrees of freedom is a *configuration*. The concept of configuration is closely related to reuse. Reusable elements are configurable. Their structure and behavior can be modified, within constraints, from *outside*, from an external entity. Think of the configuration parameters as a series of knobs, sliders, and switches which change how the devices operates. The knobs, sliders, and switches do not change themselves, they are changed by some other element in the testbench.

As an example, the thermostat in your house allows you to *configure* the ambient temperature. The house makes its internal ambient temperature a degree of freedom that can be controlled by the thermostat. The house doesn't control its own temperature. It relies on an external entity, in this case a person, to operate the thermostat and set the temperature configuration. The thermostat operator does not need to be aware of what happens to cause the temperature to change, only that the change occurs. When he changes the setting on the thermostat he does not have to be aware of the heating unit which modulates the temperature, nor that the heater runs on electricity, has an electric heating coil, a fan, and so forth. Those are all hidden within the *implementation* of the house. The thermostat control *isolates* the person who wants the air temperature to be warmer with the mechanics of how that change occurs. The implementation could change without the user being aware. The electric heater could be replaced with a more efficient unit or one that runs on a different fuel source. The thermostat can stay the same. It will work the same way and its operation will produce the same result.

In this paper we are concerned with run-time configuration, not compile-time configuration. *Run-time configuration* is supplying and responding to parameter changes after the point that the testbench starts running. *Compile-time configuration* involves changing things like class parameters, array sizes, data types, and other things that must be known at compile-time and cannot change at run-time. A discussion of compile-time configuration is worthy of another paper.

II. Essential Concepts and Definitions

In this section we will review some concepts and definitions that are essential to using the configuration facilities in UVM.

Resource Database: The mechanism that UVM uses for run-time configuration is the *resources databases*. This is an in-memory database that contains objects of different types. Configuration information is stored in and retrieved from the resources database.

Resource: In the context of UVM a *resource* is a parameterized container that holds an object. The resource class parameter defines the type of the object it contains. The data type can be legal SystemVerilog type, scalar or aggregate. More about how resources are structured is in section IV.

The reason that these container objects are called resources and not simply configuration items, is that they are general purpose. While configuring testbenches is motivation for the facility there are other legitimate use models for resources and the resources database. See section VII for more discussion on use models.

Scopes and Entities: A SystemVerilog class forms a *namespace* or *scope*. Identifiers declared within a class do not conflict with identifiers declared in other classes (namespaces). An instance of a class is an *object*. For the purposes of talking about configuration in UVM we will refer to objects as *scopes*. Sometimes we refer to instances of UVM objects — components, sequences, sequence items, etc. — generically

[1]David Pogue is a well known technology writer who, amongst other things, writes the *Ask Pogue* blog for Yahoo. He is the author of the *Missing Manual* series of books, published by O'Reilly (O'Reilly Media, 10 Fawcett St., Cambridge, MA 02138).

TABLE I. THE SET_CONFIG/GET_CONFIG INTERFACE

set_config_int()	store an integer
get_config_int()	retrieve an integer
set_config_string()	store a string
get_config_string()	retrieve a string
set_config_object()	store an object
get_config_object()	retrieve an object

as *entities*. In common usage these terms are often used interchangeably.

Just as all UVM objects (objects derived from `uvm_object`) have a name, so do scopes. UVM components each have a unique name by virtue of the way the component hierarchy is constructed. Non-hierarchical entities that serve as target entities are not necessarily required to have unique names. However, the names should be easily distinguished. More about entity names in section V.

Scope Spaces: A *scope space* is a collection of one or more scopes. Scope spaces are discussed in section V.

Configuring Entity: The resources database is populated by calls made to one of its interfaces to add resources into the database. The entity that makes those calls to populate the database is the *configuring entity*. The configuring entity is responsible for supplying values for the resources it adds into the resources database.

Target Entity: The *target entity* is an entity, usually one that is different than the configuring entity, which retrieves resources from the resources database. The target entity is typically a configurable object such as a protocol agent or a sequence that uses the resource information to modify its behavior or topology.

Override: When the object contained in a resource is replaced by a new object, or the entire resource is replaced with a new one, the resource is said to have been *overridden*. In a testbench a configuring entity may *override* a configuration value established by another configuring entity. The *overriding entity* replaces the value of a resource with a new value. Some or all of the target entities retrieving this resource will see the new value. For example, the top-level testbench environment may establish a default value for a resource which can be changed by the test.

III. HISTORY

OVM, the predecessor to UVM, did not have the resources facility. Configuration was done with the so-called "set_config/get_config" interface, a collection of functions, all methods of `ovm_component`, for setting and retrieving *configuration items* in the *configuration database*. The interface consists of six functions, listed in table I.

The configuration database is distributed across components, with each component containing a piece of it. When you call a `set_config_*()` function the configuration item is put into the portion of the database owned locally by the component from which the call originated. When you call `get_config_*()` a search is made up the component hierarchy starting with the component from which the call originated, traversing from child to parent until either the item is located or the root component is reached.

Each pair of set/get functions manipulates one type of object, either integers, strings, or objects, where in this case an object is anything derived from `ovm_object`. This interface has two key restrictions which severely reduces its functionality. One is that it supports only three data types. The other is that it only works within components.

The first restriction was initially put in place out of necessity. In 2007 and 2008 when OVM was developed and first released, SystemVerilog was still in the early stages of commercial deployment. Not all simulators available on the commercial market at that time properly supported class parameterization. It was not feasible to create a vendor-independent interface that relied on class parameterization.

One of the most important consequence of this restriction is that there was no convenient way to pass virtual interfaces to components. People have used a variety of techniques to get around this, including passing them through constructor parameter lists, putting them in *configuration objects* (See section VII-G for more about configuration objects), and assigning them directly. While all these technically worked, they lacked any elegance and were not consistent with how the rest of the testbench was configured. Also, some of these techniques can limit the reusability.

The second restriction was born from a hardware view of testbenches. In that view testbench components are much like hardware modules. They are instantiated and remain static for the duration of the simulation. Only components require configuration because sequences can be transient. Sequences get external information only from components. That would be fine if UVM testbenches were *only* constructed from components. There are other elements not part of the component hierarchy, such as sequences and sequence items, that can benefit from access to configuration information.

During the development of UVM it was recognized removing these restrictions yielded better and more flexible configuration use models. The resources database was designed with several goals in mind.

- Enable virtual interfaces to be treated like other configuration items.
- Remove the type restrictions.
- Detach configuration from the component hierarchy and enable objects other than components, as well as components, to access the configuration database.
- Provide a general purpose mechanism for sharing data between entities.

Resources and the resources database were designed to meet these objectives. As an ease-of-use feature the `uvm_resource_db#(T)` interface was created. The class parameter `T` represents the type of the object contained in the resource. This interface isolates users from the low-level details of how the database is organized. Users of the interface do not have to deal with resource containers directly. With this API most operations can be performed using only one line of code.

The VIP-TSC, as the Accellera UVM Working Group was known during the time UVM was initially under development, had concerns about backward compatibility and support for the set_config/get_config interface. Should the configuration database be left in place and the resources database serve as a parallel facility? Or, should the set_config/get_config API be removed completely?

After some deliberation, the committee wisely decided to keep the set_config/get_config API and re-implement it in terms of the resources database. Simply removing the set_config/get_config API was unreasonable in light of the fact that every OVM testbench used it. Removing the API would put an excessive burden on users converting from OVM to UVM. It might slow down UVM adoption. Having two separate facilities is also undesirable. One can easily imagine a number of scenarios where a provider of UVM testbench components uses one facility and the integrator using those component is expecting the other; or where testbench integrator is expected to mix the use of two independent facilities. The mind reels at the problems that could result.

Implementing set_config/get_config in terms of the resources database was fairly straightforward, with one exception. The search semantics of the resources database is not the same as the search semantics for the set_config/get_config interface. In the set_config/get_config facility, when calling any of the get_config_*() functions a search is initiated beginning in the component from which the call was made. The search continues upward in the hierarchy until either the desired configuration item is located or the top of the hierarchy was reached. The resources database does a lookup by name or type and searches a queue of resources associated with the name or type used in the lookup. The queues in the resources database may be assembled in a different order than the order of the search order of get_config_*(). In that case the resources database search would yield different results than get_config_*().

The `uvm_config_db` convenience layer was created to smooth over the semantic differences. It provides a means of storing and retrieving resources that is semantically identical to the set_config/get_config interface. Some of the underlying resources database code was also modified to accomodate `uvm_config_db`. Details of the differences will be discussed later in this paper.

To mimic the set_config/get_config semantics the `uvm_config_db` interface had to thwart one of the resources database design goals. Since set_config/get_config database is tied to components, so must `uvm_config_db`. That was deemed to be acceptable because *its only purpose for existence is backward compatibility*. The idea was that users would generally use the `uvm_resource_db` interface for most configuration needs. Users of set_config/get_config would not need to know that `uvm_config_db` interface is "under the hood." It would be straightforward to begin replacing set_config/get_config calls with calls to the `uvm_config_db` interface.

Direct usage of `uvm_config_db` appeared in some examples and it became accepted as the primary interface to the resources database. As will be shown later in this paper, using `uvm_config_db` is a suboptimal approach for most applications.

IV. Resource Database Architecture

This section describes implementation details of resources and the resource database. It is important to understand how the database is organized in order to understand its use models.

A. Resources

The foundation of the resources database facility is the *resource*. A resource is a container object. The type of the object that resides in the container is a parameter of the resource class. Here is the essential structure of a resource.

```
class uvm_resource#(type T=int)
  extends uvm_resource_base;
  protected T val;
endclass
```

The object `val` whose type is `T` is contained in the parameterized resource container. `Uvm_resource_base` is a non-parameterized base class that enables resources to be managed polymorphically. Most resources database operations, such as insertions and lookups, are done polymorphically using the `uvm_resource_base` class. Only operations that operate on the typed `val` require the parameterized subclass.

In addition to the contained object, the resource container has a *type handle*. Type handles are used to store resources by type. You cannot print type handles or perform operations on them other than compare for equality.

A type handle is a static reference to an empty specialized resource container. All of the instances of a resource with the same type parameter will refer to the same static type handle. Thus, the type handle serves as a proxy for the parameter type. Below is the resource container with the type handle functionality included.

```
class uvm_resource#(type T=int)
  extends uvm_resource_base;

  protected T val;
  typedef uvm_resource#(T) this_type;
  static this_type my_type = get_type();

  static function this_type get_type();
    if(my_type == null)
      my_type = new();
    return my_type;
  endfunction

endclass
```

The static member `my_type` is the actual type handle. Users access it through the `get_type()` function and not directly. This member should be declared as `local static`. `Get_type()` ensures that the static type handle is a singleton and is assigned only once. It first checks to see if `my_type` already has a value. If it does, then return it. Otherwise, create a new empty container for the type handle. Only the very first access will create a new type handle; subsequent accesses will retrieve the same one.

Accessing the data in the resource container is done through *read* and *write* operations. A resource is modified by

writing a value to the resource container. Reading a resource returns the object in the container. If the resource type is a non-scalar type, such as a class, only a handle to the object is returned. No deep copying is done during reads.

B. Resources Database

The resources database, known as the *resource pool*, is organized as a pair of associative arrays: the name table, which stores resources by name; and the type table, which stores them by type handle. Each resource is added to both the name table and the type table so it can be looked by either means. The exception is when you add a resource anonymously. In that case the resource is added *only* by type; there is no name entry added to the name table for that resource.

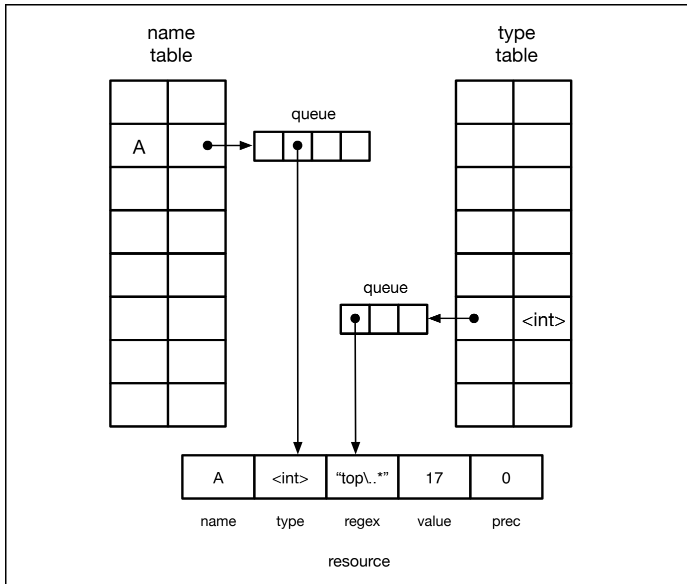

Fig. 1: Resources database data structure

As illustrated in figure 1, each entry in both the type and name tables is a queue. The queue enables the database to contain more than one resource of any name or type. In the name table the queue stores all the resources with the same name; in the type table the queue stores all the resources with the same type. Each resource with the same name or type is differentiated by its regular expression. The regular expression identifies the set of scopes in which the resource is visible. This comes into play during the search, which is explained in the next section. Adding a new entry to the database proceeds as follows:

1) Look up the name in the name table.
2) If it exists, get a handle to the queue for that name.
3) Else create a new queue for that name and insert it in the name table.
4) Put the resource handle into the queue.
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 queue.

A resource can be added to the database anonymously. In that case only the type table is updated, not the name table. For each queue, the handle of the new resource can be inserted either at the front or the back of the queue. The default is to insert it in the back. If the override flag is supplied then the resource is set to the front of the queue. The position of each resource in the queue affects the search.

Section II of [2] states "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." This implies that there is a check for duplicate resources and an error condition signalled if an attempt is made to add a duplicate. A close examination of the methods of the resource pool will reveal that this statement is not correct. There is no such check in the resource pool or any of the database interfaces. Each entry in the name and type table is a queue. Whenever a new resource is added to the database it is added to the queue (steps 4 and 8 noted above). There is no limit to the number of resources you can put in the database with the same name, type, or scope. Not all of the duplicates are visible. The exact one retrieved is dependent on the way searches are done. See the following section (section IV-C) for an explanation of the searching algorithm.

Note that there is no means for deleting resources from the database. An explicit decision was made to omit this feature. One reason is that there is no way to tell what entities have looked up and are holding handles to resources. Deleting a resource while various entities still hold handles can leave dangling resource handles. Another, possibly more important reason, is that the intent of the resources database is to provide configuration to testbenches, not to serve as a general purpose database. When doing any sort of verification it is important to be able to trace the origins of any and all activity in the testbench. A deleted resource could break the trail from a cause to an effect. Deleting resources could impugn the integrity of the verification.

C. Searching

To locate a resource in the resources database by name three pieces of information are required — the resource name, its type (a type handle), and a string representing the scope from which the search initiated. The type handle passed in to the search function is called the *search type*, and the string passed in that represents the scope in which the search was initiated is known as the *search scope*. The latter is an ordinary string representing an individual scope, not a regular expression. The name is used to locate the queue in the name table that contains the set of resources filed under that name. If the name is not located in the name table then the search function terminates with an indication that the resource was not found.

Once the queue is located it is traversed from front to back. As each resource is visited a comparison is made by performing a regular expression match between each resource's regular expression and the search scope and a comparison is made between the resource's type and the search type. All resources whose regular expression matches the search scope and whose type matches the search type are stored in a temporary match queue. If the match queue is empty then the search function terminates with an indication that the resource was not found. Even though there are resources filed under the lookup name, no regular expression matches and no type

match means that none of the resources filed under that name are visible in the search scope.

Finally, the temporary queue of matching resources is traversed to locate the one with the highest *precedence*. If there are multiple resource with the same highest precedence, the first one is returned. The `uvm_resource_db` interface does not use the precedence, defaulting it to zero. `Uvm_resource_db` lookups will return the first resource. How the `uvm_config_db` interface uses resource precedence will be explained below.

A search by type operates in exactly the same manner, except using the type queue located in the type table. Obviously, for a type search there is no name supplied to the search function, only the type handle and the search scope string are used.

V. Resources and Hierarchy

In [3], page 30, the author states "The uvm_resource_db is better suited for cases where a hierarchical context is not needed." In [2] the authors restate this same premise in section II: "The *uvm_resource_db* is a data sharing mechanism where hierarchy is not important." With all due respect to the authors, these statements are simply not correct. The `uvm_resource_db` interface was designed as a *convenience layer* for the resources database. The term convenience layer implies that the layer is not adding any new functionality, it is only providing simplified access to a lower-level layer. The resources database and the `uvm_resource_db` convenience layer were designed to support a full range of hierarchical and non-hierarchical use models. Use models will be discussed in depth in section VII.

Since UVM testbenches are constructed as hierarchies of components, hierarchy is *always* important and was considered carefully in the design of the resources database and the `uvm_resource_db` convenience layer. As noted in [1], "The most basic usage of resources is to supply configuration information to a component." And, of course, components in UVM are arranged hierarchically.

To be fair to the authors of the cited publications the `uvm_resource_db` and the underlying database each view hierarchy a bit different than `uvm_config_db` does. In the following section we will look at the component hierarchy in terms of *scope spaces*, which is a bit different than the traditional hardware view. The notion of scope spaces is key to understanding how to use the resources database.

A. Regular Expressions and Hierarchy

A scope space is a collection of one or more scopes. Scope spaces are represented in terms of regular expressions. There is a direct relationship between regular expressions, hierarchy, and scope spaces.

To understand this relationship let's first review regular expressions. A regular expression is a string that represents a set of other strings. A regular expression string contains *meta-characters* which form patterns. A (non-regular expression) string matches a regular expression if it fits the pattern. There are many syntaxes for regular expressions. The regular expression facility in UVM is based on POSIX regular expressions, which we will use in this paper.

The most basic meta-character is the dot (.). It represents a single character that's not a newline. Meta-characters must be preceded with a backslash to represent the character literally. For example, to represent a dot literally, which appears frequently as a hierarchical separator character in UVM and SystemVerilog, you specify `\.`. *Character classes* are a shorthand for a choice amongst a group of characters. The syntax for a character class is a set of characters or character ranges embedded in square brackets. For example `[ab]` represents either an a or a b. The character class `[a-z]` represents the lower case alphabet and `[a-bA-Z0-9]` represents any number or upper or lower case letter.

A group of meta-characters called *quantifiers* represent some number of occurrences of the token preceding the quantifier. These include `*`, `+`, and `?`. The `*` represents zero or more occurrences, the `+` represents one or more occurrences and `?` represents exactly zero or one occurrences. The regular expression `[a-bA-Z0-9]+` represents a string of one or more alphanumeric characters. Consider the following regular expression as an example:

```
top\.u[0-7]\.[a-zA-Z]+
```

This regular expression represents strings that begin with "top", followed by a dot, followed by a two character element that has a "u" followed by a single digit in the range 0-7. Then another dot follows, and the last part is a string of any length greater than 1 that consists of upper and lower case letters. The string `top.u3.abc` matches the regular expression; the string `top.u92` does not. For more information about POSIX regular expressions see [4].

A *glob*[2] is a kind of regular expression. Globs have a simplified syntax that offers less expressiveness than full regular expression syntax, but often its just the right thing for the job at hand. In glob syntax the meta-characters `?` means one character and `*` means zero or more characters. A dot is a literal dot. Character classes are also supported. Here is an example of a glob:

```
top.u?.*
```

This glob represents all strings that begin with the five characters "top.u" followed by any single character, followed by another dot, and finally, a string of zero or more characters. "top.ux.abc" matches the glob, while "top.u47" does not. Note that in a glob a dot is a literal dot, whereas in a proper regular expression a dot represent a single non-newline character. The interfaces to the resources database supports globs. They are converted to regular expressions internally. The meta-character conversion is shown in table II. Our example glob above would be converted to the following regular expression:

```
top\.u.\..*
```

The glob and regular expression representations are semantically equivalent, but the glob is easier to read and to write.

[2]The term glob is probably a shorthand for global, as in global regular expression. This is speculation, we do not have a citation for it.

TABLE II. META-CHARACTER CONVERSION FROM GLOBS TO REGULAR EXPRESSIONS

glob	regex
.	\.
?	.
*	.*

TABLE III. EXAMPLES OF HIERARCHICAL SPACES REPRESENTED BY REGULAR EXPRESSIONS

TOP	the single node TOP
TOP\.U[0-9]	The four children of TOP
TOP\.U2\..*	The subtree below TOP.U2

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` is set up to support only globs.

Now let's look at hierarchy. Consider that a hierarchy of components is a tree. A tree is a collection of connected nodes with one identified as a root. Each node, with the exception of the root node, has a single parent and zero or more children. Nodes that have children are *internal nodes* and nodes that do not have children are *leaf nodes*[3]. There is a unique path from the root to every node in the tree. To identify the path each node is labelled with a name. Node names do not have to be unique except within each grouping of children for a parent. That is, all the children of a single parent have unique names. Using the node names we can now form *path names*. A path name is formed by stringing together the names of the nodes in the path in the order they appear from the root to the node in question. To ensure all the strings are unique, when forming path name strings we insert a special character between each node name in the string. That special character is called the *hierarchy separator*. In Verilog and SystemVerilog a dot is traditionally used as the hierarchy separator, and we will do the same here.

Figure 2 shows a small example of a tree. The root of the tree is the node labelled `TOP`. The tree has 11 nodes, each of which has a unique path name.

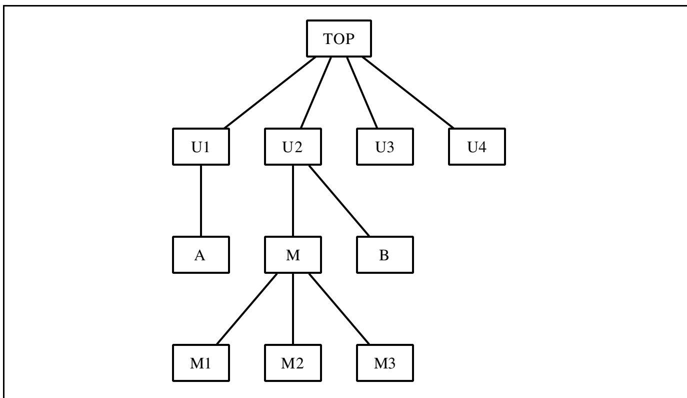

Fig. 2: Example of a tree

As long as our condition holds that the children of each parent have a unique name within that grouping of children, all paths in the tree are unique. (We leave the proof of that statement as a exercise for the reader.) By enumerating every path in the tree we will have covered the entire hierarchy space. Thus, our hierarchy space can be described by a set of path names. We can easily enumerate all the paths in this small tree.

```
1. TOP          5. TOP.U2.M          9.  TOP.U2.B
2. TOP.U1       6. TOP.U2.M.M1       10. TOP.U3
3. TOP.U1.A     7.  TOP.U2.M.M2      11. TOP.U4
4. TOP.U2       8.  TOP.U2.M.M3
```

Since path names are strings we can identify proper subsets of those strings using regular expressions. Some examples are in table III.

Nodes whose paths are identified by a regular expression do not have to be siblings, they do not have to have a common ancestor (except for the root node), they do not have to be structurally related in any way, except that they are in the same tree. To be a valid descriptor of a hierarchy space a regular expression only need to match the path names of at least one node in the tree. For example:

```
.*\.A|.*\.M2
```

This is a slightly more exotic regular expression which uses the alternation operator. The regular expressions matches either of the alternatives, `.*\.A` or `.*\.M2`. The former represents the node `TOP.U1.A` and the latter represents the node `TOP.U2.M.M2`. Thus, using regular expressions you can represent any single node or any contiguous or discontiguous collection of nodes in the hierarchy tree.

B. Pseudo-spaces

UVM components reside in a hierarchy and naturally form a hierarchy space all or parts of which can be represented by regular expressions. How do we deal with non-hierarchical target objects such as sequences or sequence items? The only criteria for matching resources between configuring entities and target entities is that the regular expression established by the configuring entity matches the name of the scope supplied by the target entity. Since non-hierarchical scopes do not have a natural naming scheme we are free to invent one. These invented names are called *pseudo-spaces*.

Since they are not tied to the component hierarchy, any naming convention can be used for pseudo-spaces. It's important to maintain some consistency amongst target scopes so that reasonable regular expressions can be used to identify them. One way to do this is to use a common prefix and a separator that is unlikely to appear elsewhere in the target scope name. Using double colons (`::`) is often used in this context. While this appears to have the syntax of the SystemVerilog scope dereference operator it is not such an operator. It's just two colons. However the association with the scope dereference operator is a convenient way to define pseudo-spaces.

As an example, let's say you want to supply configuration information to a set of sequences that drive an AXI transactor. We'll use `AXI::` as the prefix for our AXI pseudo-space. Then

[3]There are many excellent computer science texts that discuss more about trees. A classic is [5].

we can put some things in the resources database that apply only to that space.

```
uvm_resource_db#(int unsigned)::set("AXI::\.*",
                  "iterations", 1000, this);
uvm_resource_db#(int unsigned)::set("AXI::write",
                  "burst_size", 8, this);
```

The first resource in our example above applies to all members of the AXI pseudo-space. The scope argument of the second example does not contain meta-characters so it applies to exactly one member of the pseudo-space, `AXI::write`.

There are a couple of points to make about pseudos-spaces:

- You are not constrained to represent pseudo-spaces as dot-separated strings as with hierarchical spaces.
- Since you are less constrained we recommend you put conventions in place in order to maintain consistency of pseudo-space names.

VI. Convenience Layers

The resources database has a low-level interface that provides access to all features of the database. To put a resource in the database you have to first create the resource (the typed container), populate it with the object, and then insert it. This takes three separate calls — one to create a new resource container, one to put the object into the container, and one to put the container in the database.

```
uvm_resource#(T) rsrc = new(name, scope);
rsrc.write(val, accessor);
rsrc.set();
```

Getting something from the database also requires multiple calls — one to locate the resource container and one to retrieve the object from the container.

```
uvm_resource#(T) rsrc;
rsrc = uvm_resource#(T)::get_by_name(scope,
                                      name,
                                      rpterr);
val = rsrc.read(accessor);
```

While these calls are not particularluy complicated, coding all those low-level calls can be tedious and error prone. To make life a bit easier UVM has two convenience layers that abstract away most of the details of the low-level resources database interface. Specifically, they abstract away the resource container. Using the interfaces, the user only has to be concerned with setting and getting values in the database. Each of the interfaces will be discussed in this section.

A. *uvm_resource_db*

The *uvm_resource_db* interface to the resources database is shown in table IV. Note that all of the functions are static. The `uvm_resource_db` class is never instantiated. The functions are invoked using the scope dereference operator ::. For example the `set()` function is invoked by calling `uvm_resource_db#(sometype)::set()`.

The functions in the interface are organized in groups. The `set*` functions modify things in the database. `Set()` and `set_override()` both create a new resource and adds it into the database. `Set()` puts adds the new resource to the back of the queue, while `set_override()` adds it to the front, overriding all the resources that are already in the queue. The `read*` functions retrieve objects from the database; the `write*` functions lookup resources and then modify their contents. Finally, the two functions `get_by_type()` and `get_by_name()` retrieve resources and not objects contained in resources. These are mainly used by the other functions in the interface, but they are exposed to the user in case there is a need to manipulate resources as opposed to just objects contained in resources.

The arguments to these functions are consistent across all members of the interface. In the functions that write to the database `scope` is a regular expression string; in the functions that read from the database `scope` is a string that represents the search scope – i.e. the scope from which the search is initiated. `Name` is the name of a resource, and `val` is the value of the resource. `Val` can be either an input or an inout argument. For functions that modify the database `val` is an `input` argument, for query function `val` is an `inout` argument. The reason that `inout` is used instead of `output` is that if the desired resource is not located in the database the value of that argument remains unchanged. `Accessor` is a reference to the object that is accessing the database. This is only used to track which object controls which resources for debugging purposes.

The most commonly used functions are `set()` and `read_by_name()`. `Set()` creates a new resource whose name is supplied by the `name` argument and whose value is supplied by the `val` argument. The new resource is visible in the set of scopes identified by the `scope` argument. `Read_by_name` attempts to locate a resource in the database. The scope argument is a string representing the search scope. If the resource whose name matches the `name` argument, is visible within `scope`, and its type matches the type of the class parameter, then its value is returned through the `val` argument and the return value of the function is 1. If such a resource is not located the value of `val` is not changed (note that it is an inout argument) and the function returns a 0.

B. *uvm_config_db*

There is a common misconception that `uvm_resource_db` and `uvm_config_db` are two separate things. In fact, they are not. They are both interfaces to the resources database. `Uvm_config_db` is derived from `uvm_resource_db`, and most of the functions in the latter are available in the former, but not all. `Uvm_config_db::set()` masks `uvm_resource_db::set()`. To get the semantics of the latter you have to call it through the `uvm_resource_db` interface.

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

Despite the fact that `uvm_config_db` is derived from `uvm_resource_db` the former does not use much from its parent class. The only thing it uses is the function `m_show_message()` as part of the trace facility. Tracing

TABLE IV. THE UVM_RESOURCE_DB INTERFACE

```
static function rsrc_t get_by_type(string scope);

static function rsrc_t get_by_name(string scope,
                                    string name,
                                    bit rpterr=1);

static function rsrc_t set_default(string scope,
                                    string name);

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

static function void set_anonymous(string scope,
                                   T val,
                                   uvm_object accessor = null);

static function void set_override(string scope,
                                  string name,
                                  T val,
                                  uvm_object accessor = null);

static function void set_override_type(string scope,
                                       string name,
                                       T val,
                                       uvm_object accessor = null);

static function void set_override_name(string scope,
                                       string name,
                                       T val,
                                       uvm_object accessor = null);

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

static function bit read_by_type(string scope,
                                 inout T val,
                                 uvm_object accessor = null);

static function bit write_by_name(string scope, string name,
                                  T val,
                                  uvm_object accessor = null);

static function bit write_by_type(string scope,
                                  T val,
                                  uvm_object accessor = null);

static function void dump();
```

is discussed in section VIII-F. For the most part it is its own interface to the resources database.

Also, like its parent, all of the functions in `uvm_config_db` are static. It is never instantiated, all of the functions are accessed statically using the scope dereference operator. The `uvm_config_db` interface to the resources database is shown in table V.

Because `uvm_config_db` is intended to serve as a backward compatibility layer for the set_config/get_config facility, the argument names of the interface functions are compatible with those from the set_config/get_config function. `Inst_name` refers to the instance name of the component; `field_name` refers to the name of the resource; `value` is the resource value. The `cntxt` argument is a reference to the component where the search is to begin. Typically, when calling from a component `this` is passed in as the value of the `cntxt`argument. However, there is no requirement that `this` is used.

The `uvm_config_db` interface has three functions, `set()`, `get()`, and `exists()`. It also has a task `wait_modified()` which provides a way to trigger an event when a resource managed by `uvm_config_db`.

TABLE V. THE UVM_RESOURCE_DB INTERFACE

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

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

static function bit exists(uvm_component cntxt,
                           string inst_name,
                           string field_name,
                           bit spell_chk=0);
```

To support backward compatibility with the set_config/get_config facility the `uvm_config_db` layer emulates the hierarchical search used by set_config/get_config. It does this using resource *precedence*.

Resource precedence is a notion that was introduced into the implementation of the resources database specifically for `uvm_config_db`. Each resource is assigned a precedence, with the default being a large number, 1000. When a search locates multiple resources, the one with the highest precedence is returned. If there are multiple resources that have the same highest precedence then the first one located is returned. The `uvm_config_db` layer uses the distance from each component to the root as the precedence. The precedence is defined as $default_precedence - depth$, where depth is defined as the distance the current component is from the root. Items deeper in the hierarchy will have lower precedence. Configuring entities higher in the component hierarchy will take precedence over those lower in the hierarchy. Things fall apart with more than 1000 levels of hierarchy. The designers of the `uvm_config_db` interface have taken a bet that there will never be a need to configure testbenches with more than 1000 levels of hierarchy.

This is a more complex means of providing the same semantics as `uvm_resource_db` provides by relying on the top-down order of the build phase function calls. The result is identical. The `uvm_config_db::set()` function contains the following code fragment:

```
if(curr_phase != null &&
   curr_phase.get_name() == "build")
  r.precedence =
    uvm_resource_base::default_precedence -
    (cntxt.get_depth());
else
  r.precedence =
    uvm_resource_base::default_precedence;
```

The variable `r` is a handle to the resource being added to the database. Note that the precedence is used only during the build phase. In other phases the precedence is set to the default which is 1000. The net effect is that a resource added to database after the build phase will have precedence over all resources added during the build phase. Since `cntxt.get_depth()` is subtracted from the default in the build phase precedence calculation, the build phase precedences will always be numerically smaller than the default precedence. This is similar, but not identical, to the semantic of `uvm_resource_db::set_override()`.

Since the build phase override semantics could be managed by the top-down `build_phase()` execution order when using `uvm_config_db` as it is when using `uvm_resource_db`, and `uvm_config_db::set()` could use `set_override()` instead of using the default precedence, the whole precedence facility could be removed from the resources database implementation while still maintaing the desired semantics. This would have the benefit of avoiding the need to create and traverse the match queue used in resource searches. Since precedence implementation is part of the low-level interface to the resources database the performance of *all* lookups from any interface to the resources database would be improved.

In OVM the set_config/get_config functions are methods of `ovm_component` and only components can use the configuration database. The configuration database is distributed throughout the component hierarchy. When you call `set_config_*` the database in the local component, the one from which the call was made, is updated. When you want to retrive an object using `get_config_*` a hierarchical search is initiated. The search begins with the parent of the current component and continues upward in the hierarchy until either the item is located or the top of the hierarchy is reached.

The hierarchical search is emulated in the resources database. The `uvm_config_db` layer maintains a cache of resource handles using pools. Pools are singletons and can be accessed statically. Pools map a key to an object. The types of the key and the mapped object are defined using class parameters. The `uvm_config_db` uses the following pool definition:

```
uvm_pool#(string,uvm_resource#(T)) pool;
```

The key is a string and the mapped object is a resource whose type is the parameter T used in the specialization of the `uvm_config_db#(T)`. The key value is manufactured by concatenating the instance name of the component in which the `set()` call is made with the field name of the object (which is synonymous with the resource name). The concatenation is done in a clever manner to avoid problems with dots in names or other pathological cases.

The hierarchical search is emulated when `get()` is called. Oddly, the `get()` function does not use the handles stored in the pools. Instead it accesses the low-level interface of the resources database directly, bypassing all the convenience layers. The `uvm_resource_pool` function `lookup_regex_names()` is used to locate all of the resources in the resources database whose name matches `inst_name`. A queue of all the resources that match is returned. Then the function `uvm_resource#(T)::get_highest_precedence()` is used to find the resource in the queue that has the highest precedence.

The `uvm_config_db` interface does not support the use of full regular expressions, only globs. The low level interface to the resource database assumes that the scope argument is a glob unless you surround it with slashes. E.g. `top.*` is a glob, `/top.*/` is a semantically equivalent proper regular expression. If the slashes are present then the underlying library will just strip the slashes and return the string. Otherwise it will do a conversion.

TABLE VI. THE SET_CONFIG/GET_CONFIG INTERFACE

	Adding resource	Retrieving Resource
A.	`uvm_config_db#(T)::set`	`uvm_resource_db#(T)::get_by_name`
B.	`uvm_resource_db#(T)::set`	`uvm_config_db#(T)::get`

To maintain the connection with the component hierarchy, `uvm_config_db` prefixes all `inst_names` with the context name. `Inst_name` is how the `uvm_config_db` refers to scope. `Uvm_config_db::set()` has this bit of code in it:

```
if(inst_name == "")
  inst_name = cntxt.get_full_name();
else if(cntxt.get_full_name() != "")
  inst_name = {cntxt.get_full_name(), ".", inst_name};
```

`Inst_name` is prefixed with the context name plus a dot. This implies that `inst_name` is always a glob. A regular expression (surrounded with slashes) passed in will work in the sense that no error will be produced, but it will not do what is intended. For example, lets say you pass in `/u1\..*/`, a proper regular expression. That string will be changed to `top.env.my_context./u1\..*/`, assuming you are operating from context top.env.my_context. The resources database will not see a leading slash, will assume its a glob, and convert to a regular expression. Furthermore, `uvm_config_db` cannot easily be modified to use proper regular expressions, it can only use globs. It would have to parse the regular expression and insert the context name at the proper point in the regex. Certainly that is not easy, and probably cannot be done in any reliable way. This is not just a matter of moving the slash to the front of the string since regular expressions can be arbitrarily complex.

C. Intermingling the Two Interfaces

There may come a time when you need to use both `uvm_resource_db` and `uvm_config` in the same testbench. This can work just fine in many circumstances. There are two scenarios of concern noted in table VI.

In scenario A we use `uvm_config_db` to add a resource to the database, and `uvm_resource_db` to retrieve it. In `uvm_resource_db` both of the functions `read_by_name()` and `write_by_name()` use `get_name()` to locate a resource. Although `uvm_config_db` does not expressly support adding resources by type, the type table is updated anyway when new resources are added by `uvm_config_db`. Therefore it's possible to also use `get_by_type()` to retrieve resources originally added through the `uvm_config_db` interface.

To make effective use of scenario A you must understand that the `uvm_config_db` prefixes the scope name with the context name using the `get_full_name()` method. To retrieve it using `uvm_resource_db` you will have to ensure that you supply the correct scope name.

Supplying configuration information on the command line using `+UVM_SET_CONFIG_INT` or `+UVM_SET_CONFIG_STRING` and retrieving them with `uvm_resource_db` is an instance of scenario A. The

underlying command line interpreter uses uvm_config_db to store the parameters in resources database.

Scenario B, using uvm_resource_db to add a resource and uvm_config_db to locate it, is a bit more difficult because uvm_config_db changes the scope name by prefixing it with the context name. This limits the scope names you can use when adding a resource with uvm_resource_db if you plan to retrieve it using uvm_config_db. Also, uvm_config_db does not provide a means for type lookup, so anonymous additions cannot be located using uvm_config_db.

D. Comparing the Two Interfaces

The two interfaces to the resources database, uvm_resource_db and uvm_config_db are closely related. Which should you use? Here is a run down of the pros and cons of each.

context dependence: The functions in uvm_resource_db are *context independent*, while those in uvm_config_db are dependent on the context in which the calls are made. The same uvm_config_db call moved to a different context has a different meaning. The functions in uvm_resource_db have the same meaning no matter from which context they are called.

lookup by type: The uvm_config_db only supports storing and retrieving resources by name. Lookups by type are not available. The underlying resources database will store all resources by type whether or not that portion of the facility is used by uvm_config_db. You need to use uvm_resource_db to do type lookups.

component hierarchy: The uvm_config_db does not fully utilize the power of regular expressions to enable configuration of arbitrary collections of objects. It is restricted by the component hierarchy.

configuring non-components: Uvm_resource_db uses pseudo-spaces defined by regular expressions to configure any object. Uvm_config_db is not well suited to configuring objects that are not components, such as sequences or sequence items.

performance: Interestingly, the performance of the two interfaces is about the same. The uvm_config_db has the extra overhead of maintaining the resource handle caches. However, it does not keep *get records*, trace information that is kept about resource access.

expressiveness: Uvm_config_db does not support full POSIX regular expressions, only globs. For many applications this is sufficient. To describe complex scope spaces full POSIX regular expressions are required. This feature is available in uvm_resource_db.

debugging: Uvm_config_db::get() uses a function called uvm_resource_pool::lookup_reg_names() to locate resources in the database, whereas uvm_resource_db uses uvm_rsource_pool::get_by_name(). Both of them ultimately call lookup_name(), so the search mechanism is the same for both. The difference is that get_by_name() also stores *get records* when auditing is turned on. Using uvm_config_db get records are not stored, bypassing a valuable debugging aid.

Bottom line: There really is no reason to use uvm_config_db. It exists as a backward compatibility layer for the old set_config/get_config interface, and for that purpose it does a fine job. For the general problem of configuring testbenches it does not add anything and instead re-introduces old restrictions that were removed when the resources database and the uvm_resource_db convenience layer were introduced to UVM.

VII. Use Models

In this section we will look at use models for solving some common configuration problems in UVM. These are akin to patterns which represent generic solutions to a variety of common configuration problem. If you can map your problem to one of these you can apply the solution to your problem.

A. Basic Configuration

The most straightforward use of the configuration facilities is to direct resource from a configuring entity to a one or more target entities. The configuring entity, typically a test, supplies a value, and one or more target entities retrieve it.

To supply a value of type int to the entire testbench the following call goes in the configuring entity:

```
uvm_resource_db#(int)::set("*", "A", 42, this);
```

To retrieve the integer put the following in the target entity or entities.

```
int val;
if(!uvm_resource_db#(int)::read_by_name(
    get_full_name(), "A", val, this))
  `uvm_error("NO_RSRC", "resource not located");
```

B. Configuring Multiple Spaces

A very common practice is to supply different values for the same configuration item in different parts of the hierarchy. To illustrate this use model we'll use an example. Take the case where an agent retrieves a sequence to run through the resources database. In our testbench we have two instances of the same agent, each of which is to run a different sequence. The problem is to ensure that each instance executes the correct sequence. Here's the code that retrieves the sequence and runs it.

```
uvm_sequence_base seq;
function void build_phase(uvm_phase phase);
  if(!uvm_resource_db#(uvm_sequence_base)::
      read_by_name(get_full_name(),
                   "seq", seq, this))
    `uvm_error("NO_SEQ", ...)
endfunction
```

Every instance of the agent looks for resource named "seq" that is visible in its scope space. In the test we supply two separate sequences, one for each agent. We tailor the regular

expression so that each is only visible in the desired portion of the component hierarchy.

```
some_seq seq1 = new("seq1");
some_other_seq seq2 = new("seq2");
uvm_resource_db#(uvm_sequence_base)::set
    "*.master1", "seq", seq1, this);
uvm_resource_db#(uvm_sequence_base)::set
    "*.master2", "seq", seq2, this);
```

It would be prudent to use the factory to instantiate the sequences in order to allow other entities to supply different sequences. However, that detail is not germane to this use model. In this example we use globs to represent our scope spaces. There is a different one in each `uvm_resource_db::set()` call — a different regular expression to represent a different scope space. The first one will match any string that ends in ".master1", and the second one will match any string that ends i ".master2". Since any string that ends in ".master1" cannot end in ".master2" and vice versa, the sets of strings and therefore the scope spaces that these two globs represent are mutually exclusive. The globs are constructed so they can match a scope name no matter where it is in the hierarchy. This contributes to the reusability of the configuring entity (usually a test).

C. Virtual Interfaces

A virtual interface is just another type of object. We can supply a virtual interface type as a specialization for any type parameter. So the idiom for putting virtual interfaces into the resources database and retrieving them is the same as for basic configuration. The main difference is that interfaces are instantiated in modules and not classes. So the accessor argument must be left null. This has no effect how the database is manipulated, only a trace record of the call will not include a scope.

```
module top;

  axi_if my_if();

  initial begin
    uvm_resource_db#(virtual axi_if)::set("*",
                                "axi_if", my_if);
  end

endmodule
```

Note that the `set()` call has only three arguments. The fourth, the one that specifies the context, is omitted. It defaults to `null`. And then to retrieve the virtual interface:

```
virtual axi_if my_if;
if(!uvm_resource_db#(virtual axi_if)::read_by_name(
    get_full_name(), "axi_if", my_if, this))
  `uvm_error("NO_VIF", "no axi_if found");
```

It's quite common to store and retrieve virtual interface by type only. In the following example we store two instances of `axi_if` so they are each visible in separate parts of the hierarchy space.

```
module top;

  axi_if if1();
  axi_if if2();

  initial begin
    uvm_resource_db#(virtual axi_if)::set_anonymous
                                ("top.u1.*", if1);
    uvm_resource_db#(virtual axi_if)::set_anonymous
                                ("top.u2.*", if2);
  end
endmodule
```

Each target scope only needs to locate the AXI virtual interface that is intended for it.

```
virtual axi_if my_if;
if(!uvm_resource_db#(virtual axi_if)::read_by_type(
      get_full_name(), my_if, this))
  `uvm_error("NO_VIF", "no axi_if found");
```

This use model is best applied in situations where virtual interfaces can easily be identified by type and set of scopes over which they are visible.

D. Override Part I

Sometimes more than one configuring entity is responsible for supplying a configuration parameter. In that case it is necessary for one configuring entity to override the other(s). There must be a clear protocol for which entity's parameter value will take precedence so that there is no ambiguity as to where the value comes from.

Recall that resources in the database are stored in queues and lookups involve traversing a queue in a defined order. The particular resource that is located in a search is heavily dependent on the order in which the resources were inserted into the queue. Since by default new resources are pushed on to the back of the queue, and the traversal order is front to back, the first one inserted into the queue is the one that will be located — the first one in wins.

For most testbench applications configuration is done in the build phase. Build is a top-down phase, meaning that the order in which the `build_phase()` functions are called is in a top-down order (as opposed to a bottom-up order). "Top-down" is the informal term for a pre-order depth-first traversal. In this ordering each node is visited before its children are visited. "Bottom-up" refers to a post-order depth-first traversal. In a bottom-up ordering each node is visited after all of its children are visited. Each component can assume in its `build_phase()` function that its parent's `build_phase()` function has already been called. This is useful for pushing configuration information from the top of the hierarchy down to the bottom of the hierarchy.

Consider the simple component hierarchy shown in figure 3. During the build phase the `build_phase()` function in A will run before the same function runs in B and C.

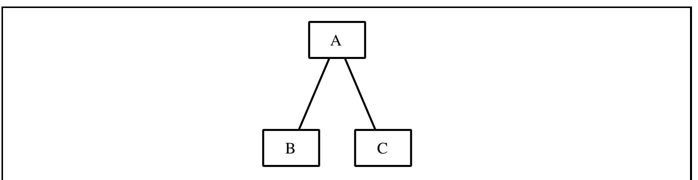

Fig. 3: Simple component hierarchy

Since `build_phase()` in A runs first it will put its items in the database first. Therefore, the resources supplied by A will be at the head of the queue and thus will be located first. The semantic is that the component higher up in the component hierarchy will take precedence over component lower in the hierarchy when configuring the testbench.

1) Override Part II: There are times when relying on the call ordering of the build phase is not the best thing to do. For example a configuring entity may provide configuration information in phases other than the build phase; or, maybe there is a requirement that configuration information provided by a component lower in the hierarchy takes precedence over that supplied higher in the hierarchy. In these cases you can use the `uvm_resource_db::set_override()` function to put the resource in front of the queue even if it is not called first.

The `uvm_resource_db::set()` function puts a new resource at the back of the queue; the `uvm_resource_db::set_override()` function puts the new resource at the front of the queue. Otherwise the two functions are identical. Both create new resources and insert them into both the type and name tables of the resources database.

E. Controlling from Below

This use model is concerned with the case where one part of the hierarchy is responsible for providing configuration information to another part of the hierarchy, not necessarily from the top. In the basic configuration use model we rely on the order that the `build_phase()` functions are called. We put things in the resource database at the top of the hierarchy knowing they can be retrieved from the lower parts of the hierarchy because of the top-down ordering of the build phase. Here we cannot solely rely on the order of the `build_phase()` functions because we cannot predict what order the children of each component are processed. If we want a component to be the configuring entity that is not necessarily an ancestor of the target component then we need to employ a different strategy. We can put the calls that put things into the database in the build phase and the calls that retrieve them from the database in a subsequent phase, such as end_of_elaboration.

The fact that the semantics of the `uvm_resource_db` are context independent makes this use model possible. The regular expression that defines the space over which a resource is visible is not relative to any particular element in the hierarchy. It refers to complete paths in some hierarchy or pseudo-space.

As an example consider our simple tree in figure 3. Let's say component B is to supply some configuration information to component C. We need to put the resource into the database in component B's build phase, and retrieve it in component C's end_of_elaboration phase.

In component B:

```
function void build_phase(uvm_phase phase);
  uvm_resource_db#(int)::set("A.C", "N", 200, this);
endfunction
```

In component C:

```
function void end_of_elaboration_phase
      (uvm_phase phase);
  int n;
  if(!uvm_resource_db#(int)::read_by_name
      (get_full_name(),"N", n, this))
    `uvm_error("MISSING_CONFIG", ...);
endfunction
```

F. Configuring Sequences

The key issue for using the resources database to configure sequences is to define names for the target sequences. Since they are not part of the component hierarchy you will need a way to group them and to distinguish amongst them. One way to do this is to create names using a space name, a space separator, and a member name. All resources that are in the same pseudo-space will have the same space name. The space separator is a syntactic element that doesn't have any specific meaning except to separate the space name from the member name. The member name is a name that is unique within the pseudo-space.

A common example of this is to create pseudo-spaces where the space name is in capital letters and the space separator is two colons. For example, `MEM::size` or `AHB::bus_width`.

You can call resource database functions anywhere, but we recommend that you use the `pre_body()` task to retrieve resources. This is simply for better performance as `pre_body()` is only called once for each sequence.

G. Configuration Objects

When you must supply many items to a single target entity it is often convenient to group all the items together in a *configuration object*, sometimes shortened to *config object*. A config object is simply a class that contains all the items necessary to configure a single target. Here is a fictitious example:

```
class some_config_object;
  rand int unsigned num_slaves;
  rand addr_t base_addr;
  rand bit [63:0] frequency; // in hertz
endclass
```

The config object is created and populated by the configuring entity and then passed to the target entity by the usual means.

In the configuring entity:

```
function void build_phase(uvm_phase phase);
  cfg_obj cfg = new();
  cfg.num_slaves = 4;
  cfg.base_addr = 'hFFFF0000;
  cfg.frequency = 27_000_000;
  uvm_resource_db#(int)::set("*", "cfg", cfg, this);
endfunction
```

In the target entity:

```
function void build_phase(uvm_phase phase);
  cfg_obj cfg;
```

```
  if(!uvm_resource_db#(cfg_obj)::read_by_name
      (get_full_name(),"cfg",cfg, this))
    `uvm_error("MISSING_CONFIG", ...);
endfunction
```

Configuration objects provide an opportunity to locally validate and constrain the items that go into them. You can put a validate() function or something similar into the configuration object class which can ensure that the various members of the class are in the correct ranges and are mutually consistent. Configuration objects are very useful for randomizing configurations. You can supply constraints in the configuration object and call `randomize()`. Those constraints can be overridden by deriving a new class and modifying the constraints.

H. Shared Objects

The resources database can be used to share objects between scopes. In this use model the configuring entity creates the object and supplies it to any number of target entities. The value in this is that the configuring entity, through the use of a regular expression, can specify precisely which targets can get the shared object. The scope space can be as broad or as narrow as desired.

Let's use the example of a shared memory.

```
  memory mem = new();
  uvm_resource_db#(memory)::set("*.axi_subsys.*",
                                 "mem", mem, this);
```

In this example we make a memory object available to the AXI-based network that is encapsulated in a component whose name is "axi_subsys". The regular expression that defines our target scope space begins with `"*."`. The root of the AXI subsystem can be relocated in the hierarchy and the space definition is still valid. We can augment this example by stipulating that different portions of the AXI subsystem each have access to different portions of the memory. To do this we'll create a configuration object that contains a handle to the shared memory, a base address, and a memory size.

```
class mem_cfg;
  memory mem;
  addr_t base_addr;
  addr_t mem_size;
endclass
```

Each instance of the configuration object will specify a non-overlapping portion of the memory for each target.

```
memory mem = new();
mem_cfg cfg;

cfg = new();
cfg.mem = mem;
cfg.base_addr = 'h00000000;
cfg.mem_size  = 'h0000ffff;
uvm_resource_db#(mem_cfg)::set("*.axi_subsys.io*",
                               "cfg", cfg, this);

cfg = new();
cfg.mem = mem;
cfg.base_addr = 'h00010000;
cfg.mem_size  = 'h0000ffff;
uvm_resource_db#(mem_cfg)::set(*.axi_subsys.jpg*
                               "cfg", cfg, this);
```

The I/O subsystem gets part of the address space and the JPEG encoder/decoder gets another part. Changes in the allocation can be controlled from the single location of the configuring entity. This does require knowledge of the construction of the hierarchy below immediate children. However, if you utilize a consistent naming scheme you can rely on it and not on hierarchical structure per se.

I. Change Notification

A class of advanced use models is based on blocking a process until a resource has changed. This is a way to notify an entity that a resource has changed. Resource *change notification* is a means to communicate changes in data that occur with very low frequency. This technique is useful for such things as communicating modal information to agents or other entities, changing constraints mid-stream, or responding to changes in shared data. Communicating information about clock operation is an example of a situation where resource change notification would be useful. However it is used, care must be taken that the technique is not abused. It should be used for changes that occur relatively infrequently and not for in-band data communications. It is not a substitute for the TLM interfaces.

There are two ways to implement resource change notification. The `uvm_config_db` interface provides a blocking task `wait_modified()`. The low-level interface to the resources database in `uvm_resource_base` provides the blocking task `wait_modified()`. Each works quite differently from the other. The `uvm_resource_db` interface does not provide a way to directly implement change notification.

Here is an example that modifies clock operation when the clock frequency changes. The component `clk_gen` is a clock generator. It's role is to supply streams of clock pulses.

```
class clk_gen extends uvm_component;

  uvm_resource#(clk_t) rclk;
  clk_t clk;

  function void build_phase(uvm_phase phase);
    rclk = uvm_resource_db#(clk_t)::get_by_name(
             get_full_name(), "clk", clk, this))
    clk = rclk.read(this);
  endfunction

  task run_phase(uvm_phase phase);
    forever run_clocks();
  endtask

  task run_clocks();
    process p;
    fork
      begin
        rclk.wait_modified();
      end
      begin
        p = process::self();
        clock_process(clk);
      end
    join_any
    p.kill();
  endtask

endclass
```

The run_phase() task continuously executes the run_clocks() task. Should run_clocks() ever terminate then it will be restarted. We will assume that the task clock_process() has at its heart a forever loop and will not terminate on its own. Run_clocks() forks two processes, one to run the clocks and one to wait to see if the resource that governs clock operation ever changes. If it does, then the locus of control will fall to the join_any statement. The clock process will be killed and the task will terminate. The forever loop in run_phase() will restart the clocks using the new information from the updated resource.

1) uvm_config_db::wait_modified(): The uvm_config_db version of wait_modified() is based on events. The interface keeps a cache of *waiters*, a list of information about resources that are begin waited upon (but not the actual resource handle). The resources are indexed by scope name. Whenever set() is called the cache of waiters is consulted to see if there are any resources waiting for the resource just changed. If so, the event stored in the waiter is triggered. Here is the pseudo-code for wait_modified():

```
 static task wait_modified(uvm_component cntxt,
                           string inst_name,
                           string field_name);

  // massage context
  waiter = new(inst_name, field_name);
  // put waiter in the cache
  @waiter.trigger;
  // Remove the waiter from the waiter list
endtask
```

The triggering of the event is dependent on a successful search for a resource when set() is called for the same cntxt, inst_name, and field_name. The exact resource that this represents is unknown at the time the trigger is armed.

2) uvm_resource_base::wait_modified(): The uvm_resource_base version of wait_modified() is based on waiting on changes to a specific resource. Here is the entire implementation of wait_modified():

```
task wait_modified();
  wait (modified == 1);
  modified = 0;
endtask
```

Each resource object contains a modified flag. The task unblocks when the flag changes value to 1. The flag is immediately reset to 0 so that it's possible to block again right away.

To use this facility you will have to get access to a specific resource via uvm_resource_db::get_by_name() or uvm_resource_db::get_by_type(). Then you can invoke wait_modified() on the retrieved resource.

J. Differences

The essential difference between the two means of change notification is that the uvm_config_db variant blocks on a change in a resource that matches cntxt, inst_name, and field_name, whereas the uvm_resource_base variant blocks on a change of a specific resource. The uvm_config_db variant is implemented in a convenience layer, the uvm_resource_base variant is part of the low-level resource database implementation.

VIII. Debugging

To make effective use of the resources database you need to be able to find out what happened when things go wrong. You need to be able to see what resources are in the database, in what scope spaces they are visible, and which entities have accessed which resources.

The resources facility provides some debugging features that make it possible to understand what where things may have gone awry.

A. Spell Checker

One of the most common errors is a simple spelling error. If the resource name supplied in a lookup is not exactly the same as the name used to store it then the resource will not be found. There errors can go unnoticed if a resource has a default. For example:

```
if(!uvm_resource_db#(int)::
  read_by_name(get_full_name(),
               "word_size", ws, this))
  begin
    ws = 8; // default
  end
```

If the resource was added to the database using the misspelled name "word_sizd" then the default will always be taken even if that was not the intent. The testbench will silently do the wrong thing.

UVM contains a simple spell checker that can be used to provide useful information about spelling mistakes. When a name lookup fails, the spelling checker is invoked. It will report on any names currently in the database that are *similar* to the one that failed the lookup. A message similar to the following will appear:

```
word_size not located, did you mean word_sizd
```

The notion of similarity is defined as names that have the smallest *Levenshtein distance* from the name in question. The Levenshtein distance is a count of the number of changes — insertions, deletions, and substitutions — required to transform one string into another. More about Levenshtein distance and an algorithm for computing it can be found at [6] and [7].

B. Dump

The most obvious debugging tool for any database is to dump it so you can see what it contains. 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 of its access records.

C. Auditing

The resources database can track a lot of its dynamic activity for review at a later time. This tracking is called an *audit trail* and is turned on using the *auditing facility*. The resources database has a static interface for controlling its behavior. Currently the only behavior under control of the static interface is the auditing facility. The interface has three functions:

```
class uvm_resource_options;
  static function void turn_on_auditing();
  static function void turn_off_auditing();
  static function bit is_auditing();
endclass
```

When auditing is turned on using the `turn_on_auditing()` collecting of *get records* and *access records* is enabled. These are discussed in subsequent sections. These three functions can be called at any time to turn auditing on or off, or interrogate the state of the auditing flag. Note that the auditing flag is *turn on by default*.

1) Get Records: The low-level resources database interface `uvm_resource_pool::get_by_name()` records each lookup access in what is called a *get record*. A get record tracks the following information for each access:

- Resource name
- Target scope — scope name of the entity performing the lookup
- Resource handle
- Time of the lookup

Failed accesses are also recorded. A `null` entry is provided for the resource handle when a lookup does not locate a resource. Each get record is pushed into a queue in the order they were generated. This will roughly be time order. The function `uvm_resource_pool::dump_get_records()` will print out the contents of the queue. This function is not available in the convenience interfaces.

A dump of the get records is useful to determine which target entity accessed which resource. In combination with a dump of the resources database you can compare regular expressions that define scope visibility with what was actually retrieved. This is a good way to find mistakes in regular expressions.

2) Access Records: The get records tracks accesses to the resources pool. Additionally, accesses to each resource can be tracked. Each resource holds a list of accesses. These track all the reads and writes for the resource. The access record contains four pieces of information:

- read time — the last time that the resource was read
- write time — the last time the resource was written
- read count
- write count

Each time a resource is read or written an access record is created and pushed onto the access queue for that resource. The function `uvm_resource_base::print_accessors()` prints all the accessors for a resource. This function is invoked for each resource when `uvm_resource_pool::dump()` is called.

D. Unused Resources

Usually, when resources are added to the resources database it is because there is an intent that the resource is used somewhere. The presence of unused resources — resources that have never been retrieved from the database — could indicate a problem. For example, the spelling error problem mentioned above could result in a resource not ever being accessed. The resource pool provides a function for locating unused resources.

```
function uvm_resource_types::
  rsrc_q_t find_unused_resources();
```

Note that the function returns a list of resources. This list may be empty if there are no unused resources. The function `uvm_resource_pool::print_resources()` is required to print the list of resources. Fortunately, a convenience function is available to locate and print unused resources. The function is called `check_config_usage()`. For historical reasons[4] the function is a member of `uvm_component`.

E. Lookup by Scope

There are times when it is instructive to see all of the resources that are visible in a particular scope. The resource pool provides the function `uvm_resource_pool::lookup_scope()` to do just that. It takes a single string argument representing a scope. It searches through the entire database to find all the resources that match the scope. It returns a queue of all the matches. Of course, if no resources match the scope then the queue will be empty. The queue is the same type returned by `find_unused_resources()`. As an example, the following code snippet will print all of the resources visible within a component where `get_full_name()` is defined.

```
uvm_resource_types::rsrc_q_t q;
q=uvm_resource_pool::lookup_scope(get_full_name());
uvm_resource_pool::print_resources(q);
```

The function `lookup_scope()` is quite expensive as it must visit every resource in the database and perform a regular expression match. So it must be used sparingly.

F. Tracing

The activity of the `uvm_resource_db` and the `uvm_config_db` can be *traced* during execution. Tracing means that messages are printed as each interface function is called. Each function in both interfaces will print a trace message when tracing is turned on.

The tracing functionality for each interface is managed separately. There are two classes that control

[4]"Historical reasons" means that there must have been a good reason for this at one time but no one can remember any longer what it is.

tracing — `uvm_resource_db_options` and `uvm_config_db_options`. These classes have static functions for turning tracing or off and for interrogating the tracing state of each of the respective interfaces.

```
class uvm_resource_db_options;
  static function void turn_on_tracing();
  static function void turn_off_tracing();
  static function bit is_tracing();
endclass

class uvm_config_db_options;
  static function void turn_on_tracing();
  static function void turn_off_tracing();
  static function bit is_tracing();
endclass
```

Thd tracing functions can be called at any time to turn on or off tracing for either interfaces. Command line options are available to turn on tracing for either interface. `+UVM_CONFIG_DB_TRACE` turns tracing on for the `uvm_config_db` interface; `+UVM_RESOURCE_DB_TRACE` turns tracing on for the `uvm_resource_db` interface. By default tracing is turned off.

IX. Conclusion

The resources database is a powerful tool for configuring UVM testbenches. There are several ways to access it – the `uvm_resource_db` convenience layer, the`uvm_config_db` convenience layer, and direct low-level interaction with the database. You should use the type of interaction that is most well suited to the application at hand. For most applications the `uvm_resource_db` layer provides the best combination of features, convenience, and performance.

Acknowledgment

We would like to thank **Justin Refice** of NVIDIA and **Kelly Larson** of Paradigm Works for reviewing drafts of this paper and providing valuable feedback. We also owe a debt of gratitude to **Kirin Sama** for providing encouragement and the opportunity to write the paper.

References

[1] M. Glasser, "Advanced testbench configuration with resources," in *DVCon 2011*. Accellera, 2011.

[2] V. R. Cooper and P. Marriott, "Demystifying the uvm configuration database," in *DVCon 2014*. Accellera, 2014.

[3] V. R. Cooper, *Getting Started with UVM: A Beginner's Guide*. Verilab Publishing, 2013.

[4] "Regular-expressions.info." [Online]. Available: www.regular-epressions.info

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, 1983.

[6] "Levenshtein distance." [Online]. Available: http://en.wikipedia.org/wiki/Levenshtein_distance

[7] "Fast, memory efficient levenshtein algorithm." [Online]. Available: http://www.codeproject.com/KB/recipes/Levenshtein.aspx

[8] *UVM 1.1 Class Reference*, Accellera. [Online]. Available: http://www.accellera.org/downloads/standards/uvm

[9] *UVM 1.2 Class Reference*, Accellera. [Online]. Available: http://www.accellera.org/downloads/standards/uvm