Advanced Testbench Configuration with Resources

Mark Glasser

Mentor Graphics Corporation
Fremont, CA
`mark_glasser@mentor.com`

***Abstract*—Building robust, reusable testbenches means the testbench elements must be *configurable*. At its essence, configuring a testbench is a matter of populating a database with name/value pairs and providing a means for testbench objects to access that database. Simply storing and retrieving name/value pairs does not tell the whole story. There are a number of architectural issues concerning the design of the database and how to effectively populate and use the items in the database to build highly configurable, reusable testbenches. UVM provides a facility called *resources* that provides the configuration infrastructure and API. We will discuss approaches to common configuration problems in term of resources. We will show how to use resources to implement sophisticated configuration use models.**

I. Introduction

An important requirement for constructing reusable testbenches is a means for providing information to various parts of the testbench from a central location. This is called *configuration*. Another requirement is enabling data sharing between testbench elements in a thread-safe and tractable manner. UVM contains a very generalized facility for meeting these requirements called *resources*. By “generalized” we mean that the facility supports a number of interesting use models and is suitable for specialization by creating layers on top of it.

Any object in a testbench could require information to be passed to it from an external source. That includes components, sequences, sequence items, modules, and interfaces, or any other object. We will refer to these generically as *testbench elements*. In this paper we will look at the issues related to supplying information to testbench elements and sharing information between them.

Central to the whole issue of configuring testbench elements is reusability. A reusable element is one that can be used in a variety of different circumstances. To be reusable, the element needs some knowledge of the circumstance at hand so that it can alter its topology or behavior accordingly. The element must have information given to it about how it should operate in the current environment. That information is called *configuration information* and can be passed to testbench elements in a variety of ways. The most common of these are:

- class parameters
- constructor arguments
- function calls

Information passed as class parameters must be supplied at compile time and cannot be changed at run time. This is fine for structural information that does not need to change or for information that is known at compile time and will not or cannot change. For example, bus width is a structural parameter that often must be fixed at compile time in order for the testbench to connect properly to the DUT.

Passing information in constructor arguments can lead to fragile systems. When you add a new argument to a constructor you have to make sure that all the constructor arguments are passed correctly as one object instantiates another. If this is not done carefully the wrong information may be passed along, or compile-time errors may occur if the argument orders and types are not correct. In UVM, objects are often instantiated through the factory where constructors are called for you and you don’t have the option of modifying constructor arguments. Modifying constructors to pass information to testbench elements is highly discouraged in UVM.

In the last case, functions can be called in testbench elements to set or get configuration information. Those functions put information into a central *configuration database* or retrieve information from it. This technique provides the most general way of accessing configuration information, so it is the one that we will focus on in this paper. The element that puts an item into the configuration database is the *setter* and the element that retrieves it is the *getter*.

II. Configuration Issues

There are a number of issues that arise when configuring a testbench. In this section we will explore some of them.

When verifying a DUT you must be clear on exactly what happened during simulation. Whether errors occur during execution or not, you need to know that the testbench operated exactly as you expected. If the testbench quietly does the wrong thing then there is a reasonable likelihood that the results are invalid. For that reason you must be able to precisely direct configuration information before execution, and you must be able to validate post facto what exactly happened.

When setting configuration information you must be able to easily identify the element or elements that will receive it. The information could be directed at one element, some subgroup of elements or all elements.

It may be that the setter of some piece of configuration information does not have the last word on the value supplied. There may be another setter that does. Neither of the setters may be aware of the other. The precedence of setters must be deterministic so that it is well understood exactly which information is retrieved by a getter. There must be a way to override information already put into the configuration database. The exact value retrieved from the database must be unambiguous in the presence of overrides. This implies that the getter does not need to know who the setter is. In that respect the database acts like a mailbox. Some entity puts data in the mailbox and another retrieves it without either having direct knowledge of the other.

Data communication through the database must be type-safe. That is, each object should be retrieved from the database using precisely the same data type as the setter. Any deviation from this could result in incorrect information being retrieved. There should be no artificial restrictions on the types of object that can be communicated through the configuration system.

III. Resources

The resources facility in UVM is comprised of polymorphic resource containers, a database for storing those resource containers, and a means for locating resources in the database. The polymorphic resource container, simply called a resource, is a parameterized container that holds arbitrary data whose type is defined by the parameter. It has methods for moving data into and out of it as well as other operations described later. Resources are collected together in a centralized database which is searchable by various means. The skeletal structure of the resource classes is as follows:

```
class uvm_resource_base extends uvm_object;
  string scope;
  int unsigned precedence;
endclass

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

Polymorphism refers to the notion of handling objects of different types uniformly. This is accomplished by creating a family of class types, all with a common base class. The base class contains methods that can be used to operate the objects in the family independent of the specific derived type of any family member. Some methods may be virtual methods and may be reimplemented in the derived class. Polymorphism, virtual functions, and other elements of object-oriented programming are discussed in many places in the literature, for example [2] and [4].

The class `uvm_resource_base` serves as the common base class for the family of resource containers. It provides an interface that applies to all resources, no matter what type it is. The term *interface* is often overused. Here, we mean a functional interface: a set of functions that operate a data structure. When dealing with resources polymorphically `uvm_resource_base` handles are passed around. The class `uvm_resource#(T)` provides an interface that is type specific, such as `read()` and `write()`, which makes operations on resources type safe.

Resources are scoped, meaning each resource is visible in one or more scopes within the testbench. A scope is a context, such as a component instantiated in the component hierarchy. As far as the resource facility is concerned, a scope is just a unique string. Typically the string contains dots to separate hierarchical elements, such as the string returned from `uvm_object::get_full_name()`, but there is no requirement that it does so. Since a scope is just a string it can represent any space, real or imaginary, that you like.

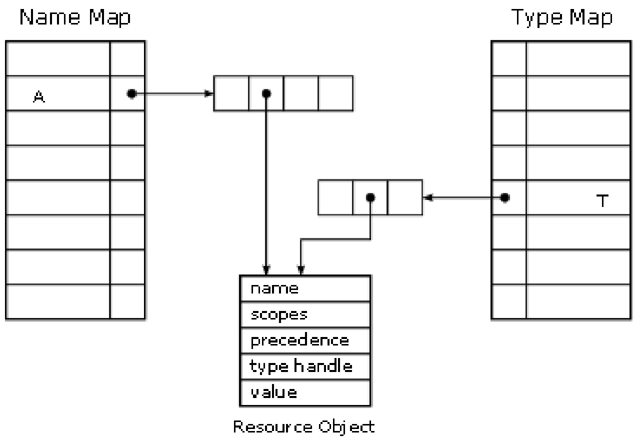

Fig. 1. Organization of the resource pool

A set of scopes is represented using a regular expression. A regular expression is a shorthand notation for a set of strings over some alphabet. For our purposes the set of strings is a set of scopes. For example, the regular expression `top\.env\.u.*`[1] represents all of the scopes that begin with the prefix `top.env.u`. For example, `top.env.u1` and `top.env.usb.mon` are strings that are in the set described by the regular expression. The set of strings (scopes) represented by our regular expression is infinite because the asterisk, also known as a *kleene star*, which terminates the expression means there can be zero or more instances of the previous element. There is no upper limit on "more".

Resource Pool. Resources are stored in a database, called the *resource pool*. The organization of the resource pool is designed to enable efficient insertions and lookups of resources. The primary organization of the pool is a pair of maps: a name map and a type map. When a new resource is inserted into the pool it is entered into both maps. Each map entry contains a queue of resources rather than a single resource. This enables the pool to store multiple resources with the same name or the same type.

Figure 1 illustrates the organization of the resource pool. The order of the entries are in each queue is significant, as it affects the order in which they are searched when looking up resources in the pool. The significanse will become clear as we look in detail at the algorithm for locating resources in the pool.

The essential algorithm for lookup up a resource by name in the resource pool is as follows:

[1]The regular expression syntax implemented in UVM is the extended regular expression syntax as defined in the Posix standard. In that standard a dot (.) matches any single character and an escaped dot (\.) matches a single dot.

1) Lookup the queue of resources associated with a name in the name map.
2) If the queue is empty then there are no resources with this name. The lookup has failed. Return null.
3) set `high_precedence` to 0. Set the search target to the first resource in the queue that is visible in the current scope. Visibility is determined by matching the regular expression associated with the resource, which represents the set of scopes over which the resource is visible, with the current scope. If there is a match then the resource is visible in the current scope.
4) Traverse the queue from front to back, visiting each resource.
5) For each resource, determine if it is visible in the current scope.
6) Upon visiting a resource, if it is indeed visible in the current scope, then check to see if its precedence is greater than the current value of `high_precedence`. If it is, then set `high_precedence` to the new value in the resource and save the resource as the new search target.
7) After all resources in the queue have been visited, return the current search target. If, after all resources in the queue have been visited, none are visible in the current scope then the lookup has failed and the search target will be null.

Looking up a resource by type works exactly the same way, except a queue of resources is located by type handle instead of by name. The result of this is that the search target returned is the resource that is visible in the current scope and has the highest precedence amongst all the resources with the same name. In the case where there are more than one resource with the same highest precedence, the one earliest in the queue (that is, the first one encountered in the search) is returned as the search target. The exact resource that is located by a search depends on three things: The name or type of the resource (depending on whether you are looking up a resource by name or by type), the precedence of the resource, and the order in which it was placed in the queue.

Auditing. To track activity during execution the UVM resources facility provides an auditing capability. Two types of information are collected. The first is a set of access records. Each resource is accessed – either read or written – by some object. The auditing facility can track the name of the object from which each resource was accessed. The accessor is supplied to the system by the user each time a resource access is made. Typically, the accessor is `this`. For example,

```
x = rsrc.read(this);
rsrc. write(x, this);
```

In both the `read()` and `write()` functions an optional argument identifies the resource accessor. Along with the name of the accessor, the time of the last read, the time of the last write, the number of reads, and the number of writes are stored. This information is stored for each resource

The second type of information collected is a *get record*. Each time a resource is looked up in the resource pool a record of the lookup is created. The record includes the name of the object being looked up, the scope supplied as the current scope, the resource handle, and the time of the lookup. If the lookup is unsuccessful the resource handle will be empty. Both the per-resource access records and the history of resource pool lookups can be dumped at any time. Typically, this information is dumped at the end of the simulation and serves as a way to determine if the testbench was configured properly.

All of the details of the structure and interfaces for resources and the resource pool can be found in [1].

IV. Configuration Use Models

In this section we will discuss some of the use models for configuring testbenches using UVM's resources facility.

Basic Usage. The most basic usage of resources is to supply configuration information to a component. The component retrieves a value from the resource pool and uses it to control its topology or behavior. For example, the test might issue the following call which creates a new resource, populates it, and inserts it into the resource pool.

```
uvm_resource_db#(int)::set("size",
                           "top\.u1\..*",
                           8, this);
```

This creates a new resource whose name is `"size"`, is visible in the scopes identified by `"top\.env\.u\..*"`, and has a value of 8. The last argument, `this`, provides accessor information for the auditing facility. The component that retrieves this resource would read the value from the database.

```
if(!uvm_resource_db#(int)::read_by_name(
                                "size",
                                get_full_name(),
                                val,
                                this))
  `uvm_error("configuration",
             "Resource A not found in this scope");
```

The call to `get_full_name()` identifies the current scope: the scope that is requesting the resource. `read_by_name()` returns a bit that indicates whether or not the lookup succeeded. The actual value of the resource is returned as an inout argument.

Notice that in the basic use model we do not have to deal with resource objects directly. The class `uvm_resource_db#(T)` is a convenience layer on top of the low-level database access classes and methods. It is not instantiated. Instead, it contains a collection of static functions that operate resources and the resource pool. The `set()` function creates a new resource object whose type is defined by T, writes data into it, and inserts it into the resource pool. The function `read_by_name()` looks up the resource whose name is `"size"` and is visible in the current scope. It then reads the data from the located resource and returns it via the inout function argument `val`. The success or failure of the operation is returned as the function return value. If a resource matching the search criteria is not located in the resource pool then the function returns a 0. If it succeeds then it returns a 1.

Components should not have to know where they are located in the hierarchy, nor should they have to know which element is providing resource values. A call to `read_by_name()` to retrieve resource values uses the current scope to locate the correct resource. Components can identify their current scope by calling `get_full_name()`. It may be the case that a component is instantiated multiple times, and the test wants each instance to have a different value for a particular configuration item. In that case the test can create separate resources with the same name, each of which has a different scope visibility. For example:

```
uvm_resource_db#(int)::set("A", "top.u1.*",
                           14, this);

uvm_resource_db#(int)::set("A", "top.u2.*",
                           1016, this);

uvm_resource_db#(int)::set("A", "top.u3.*",
                            82, this);
```

Here we create three resource, all named `"A"`, each with a different scope visibility, and each with a different value. Components in the `top.u1` sub-hierarchy will receive the value 14 when they look up resource texttA, components in the `top.u2` sub-hierarchy will receive the value 1016, and components in the sub-hierarchy `top.u3` will receive the value 82.

Overrides. Often it is the case that an agent will set a default value for a resourceS and the test or some other testbench element will want to override it. As we saw in section III, search order is determined by the order in which resources are added to the resource pool. The most straightforward way to override resources is to make sure that the override is entered in the database first. One way to do this is to create all your resources in the `build()` phase. Since build is a top-down phase, components higher in the component hierarchy will be processed before those lower in the hierarchy. Thus, those resources entered at the top of the hierarchy will be searched before those farther down.

If you are not creating all your resources in the build phase you run the risk of a race condition. If you create resources in the `run()` phase, for instance, you don't know which run task will execute first. You are relying on the order in which the run tasks execute to define the search order of resources. Since the order in which SystemVerilog processes execute is not necessarily deterministic, neither would be the resource search order. This not desirable. When inserting a resource into the resource pool you can use `set_override()` instead of `set()`. `set_override()` places the resource at the head of the queue, ensuring it will be searched before any other resource already in that same queue. Of course, if you have multiple processes, each calling `set_override()`, you still have a race condition. To get around this problem you can change the precedence value in the resource before inserting it in the pool.

Resources by Type. Each resource container has a *type handle*: a static member whose value uniquely represents the type of the container specialization. This is useful for storing and retrieving resources by type. One usage of this is to configure agents using a unique *configuration object* for each agent type. Consider `some_agent,` which is parameterized using bus width and the type of its configuration object.

```
class some_agent#(type CONFIG=int, int WIDTH=8)
  extends uvm_component;

  CONFIG cfg;

  function void build();
    if(!uvm_resource_db#(CONFIG)::read_by_type(
                                  get_full_name(),
                                  cfg,
                                  this))
      `uvm_error("build",
                 "configuration object not found");
  endfunction

  ...

endclass
```

In this use model, each agent type has a unique type for its configuration object, which is specified as a class parameter. `read_by_type()` looks up a resource by its type handle using the algorithm detailed in Section III. In this case the name of the resource is not important. Because the resource is a parameterized container, the methods used to store and retrieve the resource are similarly typed. Thus the `read_by_type()` call is type-safe, and no casting or other run-time type checking is required.

Virtual interfaces are another place where storing and retrieving resources by type can streamline your code. In [3] a technique for handling virtual interfaces is described using a customized container. The generalized resource container can take the place of the specialized virtual interface container. Virtual interfaces can be stored and retrieved either by name or by type. Here is a short example using types.

```
module top;
  ...
  bus_if() bif;  // instantiate interface

  initial begin
    uvm_resource_db#(bus_if)::set_anonymous(
                                  "top.*",
                                  bif,
                                  this);
  end
endmodule
```

In our hypothetical testbench there is only one instance of `bus_if`, and so there is no need to bother with a name. We can store the virtual interface anonymously and retrieve it by type. The function `write_and_set_anonymous()` works the same as `set()` except that the resource is entered only in the type map, not into the name map. There is no way to look up an anonymous resource by name.

Sequences. Sequences have been neglected in previous incarnations of the configuration system. Retrieving a configuration item from the resource pool is done the same way in a sequence as in a component. The difference is how the current scope is determined. Components have a natural scope which can be determined by calling `get_full_name()`. For sequences there are several choices for supplying a current scope. The most obvious choice is to use the scope of the sequencer on which the sequence is hosted. For example:

```
string scope = m_sequencer.get_full_name();
...
if(!uvm_resource_db#(int)::read_by_name("A",
                                        scope,
                                        val,
                                        this))
  ...
```

Another choice is that each sequence provides its own scope. The scope string can be anything, as long as it uniquely identifies the element which will be requesting resources from the resource pool. Any scope that is not part of the component hierarchy is called a *pseudo-scope*. Pseudo-scope strings can be anything. The only requirement is that the element setting resources agree with the element getting them. As an example, consider a collection of resources that drive an AHB interface. Each sequence could supply `"ahb"` as its scope.

```
if(!uvm_resource_db#(int)::read_by_name("A",
                                        "ahb",
                                        val,
                                        this))
```

Thus, resources, such as the one named `"A"`, in the `"ahb"` pseudo-space, will be retrieved by sequences that identify themselves as being in that same pseudo-space.

Virtual sequences, sequences that are invoked without an associated sequencer, can retrieve information about sequencers on which to initiate sub-sequences from the resource pool. This eliminates the need for so-called *virtual sequencers*: sequencers whose role is to provide a location to retrieve handles to other sequencers. Agents can put sequencer handles into the resource database in an agreed-upon pseudo-space. Virtual sequences can then retrieve them from this pseudo-space and use them to invoke sub-sequences.

V. Data Sharing Use Models

So far we've discussed use models that involve storing and retrieving data as resources. The data is supplied by the test or some other top-level element to be consumed elsewhere in the testbench. Another collection of use models involves storing and retrieving resources for the purpose of sharing data amongst various testbench elements.

One use model is for a process to wait on the value change of a resource. Consider an example where you want to run specific sequences when the DUT goes into certain states. You can capture that state as a resource. When the resource value changes the test can respond by executing a new sequence. The choice of sequence is based on the value of the resource. The monitor updates the shared resource. The test waits for the resource to change value and responds accordingly when it does. Here's a sketch of the relevant parts of the monitor that implements this use model.

```
class monitor extends uvm_component;

  uvm_resource#(int) rsrc;

  function void build();
    rsrc = uvm_resource_db#(int)::get_by_name(
              get_full_name(), "dut_state");
  endfunction

  task run();
    int state;

    // somewhere in the state machine...
    rsrc.write(state, this);
    ...
  endtask
endclass
```

The monitor retrieves the shared resource using `get_by_name()`. The value of the resource is updated appropriately as the state machine changes states. An agent can respond to changes the state of the DUT by using the `wait_modified()` task:

```
class agent extends uvm_component;

  uvm_resource#(int) rsrc;

  monitor m;
  driver d;
  uvm_sequencer sqr;

  function void build();
    m = new("monitor", this);
    d = new("driver", this);
    sqr = new("sequencer", this);
    rsrc = uvm_resource_db#(int)::get_by_name(
            get_full_name(),"dut_state");
  endfunction

  task run();
    fork
      sequence_kicker();
    join_none
  endtask

  task sequence_kicker();
    uvm_sequence seq;
    while(1) begin
      rsrc.wait_modified();
      seq = lookup_sequence(rsrc.read(this));
      if(seq != null)
        seq.start(sqr);
    end
  endtask
```

The agent retrieves the same resource as the monitor. The agent's `run()` task starts a free-running process called `sequence_kicker()`. This task waits until the shared resource changes value. When it does, it uses the new value to choose a new sequence to run. We've allowed for the case that not all state changes result in a new sequence being run. If `lookup_sequence()` returns `null`, then no new sequence is started for that state change.

Barriers are commonly shared to synchronize processes that otherwise do not have a clean way to share data. You can store and retrieve handles to barriers in the resource pool. Each process that is to be synchronized retrieves a handle to a shared barrier from the resource pool. Using the resource pool instead of global variables has several advantages. It's impossible to tell which elements have accessed a global variable. Such elements reduce reusability of any element that uses them by creating a compile-time dependency on the presence of a variable outside its own scope. The element can only be used when the global variable is present. The resource facility removes the requirement for a global variable. If there is a need to share multiple shared objects, they can easily be created and added to the resource pool without new global variables.

Besides lack of accountability, global variables suffer from lack of thread-safety. In the case where multiple processes are modifying a shared object, you need a way to lock and unlock a variable so that one process does not inadvertently overwrite a value or a way to resolve potential race conditions when updating a variable. Resources have a lock interface which lets you lock and unlock a resources so it can be updated atomically.

As an example, consider an integer resource that contains the serial number of the current transaction. The testbench has many sequences generating transactions, but only one at a time is in process. Here is an illustration of the code you would put in the sequence to update the serial number in a thread-safe manner.

```
int serial_number;
uvm_resource_db#(int) sn;
my_item item;
sn = uvm_resource_db#(int)::get_by_name("sn",
                                        my_scope);
...
sn.lock();
serial_number = sn.read();

// Create sequence item for
// our transaction. Put the serial
// number into the sequence item.
...
item.serial_number = serial_number;

// Finish populating the item
// and send it.
...

// update the serial number resource
// and unlock it.
sn.write(serial_number+1);
sn.unlock(();
```

You can also use a testbench resource to lock a hardware resource. A shared memory, for example, can only be accessed by one process at a time. To ensure reads and writes do not overlap you can use the locking interface on a resource. In this use model, we do not use the value of the resource, only the lock.

```
uvm_resource#(bit) r;
r = uvm_resource_db#(bit)::get_by_name("mem_lock", my_scope);

// lock when doing a read
r.lock();
data = mem.read(addr);
r.unlock();

// lock when doing a write
r.lock();
mem.wirte(addr, data);
r.unlock();
```

`lock()` is a task and will potentially block if another process currently has the resource locked. Because you are using a resource lock to protect the memory reads and writes you are ensured that the read and write operations are thread-safe.

VI. Conclusions

Configuring testbenches and sharing data amongst testbench elements requires careful planning in order to make sure that the right data lands in the right place. The UVM resources facility provides a mechanism that enables precise placement of configuration data within all kinds and types of testbench elements. To make sure that the right data is in the right place, an auditing facility shows all resources operations.

In this paper we have explained the structure and operation of resources, and how they can be stored and retrieved in the resource pool. We have also shown a variety of use models, starting from very basic storage and retrieval of resources to more sophisticated sharing of resources. The set of use models identified here is by no means comprehensive. The generality of the resources facility lends itself to creative application of new use models as part of a robust and reusable testbench architecture.

VII. Acknowledgments

The author wishes to acknowledge Mohamed Elmalaki of Intel Corporation for his contributions to this paper.

References

[1] Accellera. *UVM 1.0 Reference Manual*, 2011.
[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elememts of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[3] M. Glasser. *The OVM Cookbook*. Springer, 2009.
[4] J. Martin and J. Odell. *Object-Oriented Methods: A Foundation*. Prentice Hall, 1995.