A Unified and Simple Technique to Execute Both Sequences and Virtual Sequences

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

Mark Glasser
Paradigm Works, Inc.
mark.glasser@paradigm-works.com

***Abstract* - Virtual sequences are an essential element in UVM testbenches. They enable a test to exercise and manage multiple DUT interfaces. UVM, however, is silent on how virtual sequences obtain sequencer handles representing those multiple interfaces. A normal (non-virtual) sequence is bound to a sequencer when launched. A virtual sequence is not bound to any sequencer, and a way must be found to access sequencers. Traditionally, testbenches are constructed with so-called virtual sequencers, which are used not as sequencers per se but as containers for sequencer handles. p_sequencer handles are used to access the sequencer handles in virtual sequences.**

This paper will explain why virtual sequencers are not the best way to manage sequencer handles in a virtual sequence. We will also discuss an alternative testbench architecture that does not require virtual sequencers but instead uses sequencer containers and the resource database. We will share detailed descriptions of two implementations of sequencer containers: the sequencer pool (`sqr_pool`) and the sequencer aggregator (`sqr_aggregator`). We will demonstrate how these containers alleviate the issues associated with virtual sequencers through code examples.

Table of Contents

Table of Figures

I. Virtual Sequences and Virtual Sequencers

UVM sequences are launched with the **`start()`** task, which takes a single optional argument. When a valid sequencer handle is supplied as the argument, the sequence and sequencer are bound together. A sequence bound to a sequencer via the argument to **`start()`** is called a *normal sequence*. When the argument is omitted or **`null`** is provided, the launched sequence is called a *virtual sequence*. Each virtual sequence is responsible for obtaining sequencer handles to coordinate generated sequence items across multiple sequencers.

UVM is silent on how virtual sequences obtain sequencer handles to transmit sequence items. Traditionally, verification engineers use so-called virtual sequencers. A *virtual sequencer* is a sequencer that serves as a container for sequencer handles and does not function as a typical sequencer. It does not transmit sequence items from sequences to a driver. A virtual sequencer does none of the usual actions that sequencers are designed to do. Its only purpose is to hold sequencer handles.

Virtual sequencers are sequencers derived from the base class **`uvm_sequencer_base`**. The members of the virtual sequencer (derived) class include sequencer handles, which must be populated sometime between the construction of the virtual sequencer and the point where sequences access them.

The virtual sequencer was invented when OVM was widely used before UVM was developed. In the context of OVM, virtual sequencers made sense because the resource database did not exist.

II. Existing Virtual Sequence Techniques: Advantages & Disadvantages

Multiple commonly used virtual sequence techniques exist, each with its own advantages and disadvantages. Below are three of those techniques.

A. init_vseq Technique

The **`init_vseq`** method is the technique shown on Siemens Verification Academy and is probalby the least useful of the techniques described in this paper.

This technique places an **`init_vseq`** method in a **`test_base`** class, which requires full, hard-coded paths to the physical locations of the subsequencers in the testbench. As the testbench grows and changes shape, the paths must be properly updated for each new variation of the tests. The **`init_vseq`** technique requires the test to call the **`init_vseq()`** method with a virtual sequence argument to assign the subsequencer handles in each virtual sequence before the virtual sequences are started.

In general, hard-coded paths are very problematic in large and evolving UVM testbench environments, and a source of **`null`**-reference bugs when the paths are not correctly updated.

B. Virtual Sequencer Technique

Each virtual sequence needs access to the physical subsequencer handles. The virtual sequencer (**`vsequencer`**) method relies on a top-level environment to copy those handles into one or more **`vsequencer`** components, so each new environment is responsible for copying the appropriate subsequencer handles into the correct **`vsequencer`**. One advantage of this technique is that the tests are not required to maintain hard-coded paths to the subsequencers. Instead, each test must start virtual sequences on the appropriate **`vsequencer`**, and from there, the **`vseq_base`** can retrieve the handles from the **`vsequencer`** component for use by the virtual sequences.

Typically, one thinks of storing component handles in a config object. Still, since sequences must be started on a sequencer, which in this case is the **`vsequencer`**, the **`vsequencer`** acts as the config object to hold the subsequencer handles, and each environment is responsible for setting those handles for use by the tests.

After a virtual sequence is started on a **`vsequencer`**, the **`vseq_base`** class has access to the stored subsequencer handles and can retrieve the subsequencer handles in the **`vseq_base`**, and each extended virtual sequence is able to inherit the properly set subsequencer handles to coordinate sequence activity across the desired subsequencers.

Using the **vsequencer** method, the only hard-coded path is referenced when the test starts a virtual sequence on a path to one or more **vsequencers**. This can change in an evolving UVM testbench environment, but the changes are infrequent, and there would only be a few paths to a few different **vsequencers** at most. This technique is not nearly as problematic as the **init_vseq** technique described above in Section II-A.

When this paper was published, the **vsequencer** technique was probably the most commonly implemented virtual sequence method used by verification engineers.

C. uvm_resource_db Technique

The **vsequencer** method is the most commonly used virtual sequence technique because engineers have been operating under the false assumption that subsequencer handles could not be stored as resources and then directly retrieved inside a sequence. This false assumption results from engineers commonly attempting to use the **uvm_config_db** API to access resources. Generally speaking, sequences cannot use the **uvm_config_db** API because the **uvm_config_db** commands require retrieval to a component, and a sequence is not a component. Some engineers have figured out that if they hack the **uvm_config_db** command by passing a component handle of **null**, it will bypass the component-checking mechanism of the **uvm_config_db** commands. Then they can use an **inst_name string** to retrieve a resource, even into a sequence. But this is a needless hack!

The **uvm_resource_db** API can use simple strings to store and retrieve resources, and the strings are not required to be component paths, as required when using the **uvm_config_db** API. The strings are just passwords that are stored with resources that must be matched when retrieving the corresponding resources.

This means each agent can store a subsequencer handle at a string location in the resource name table, and then the **vseq_base** can retrieve the resource from the name table at that string location.

This technique does not require a separate **init_vseq** method to set subsequencer handles, nor does it require a **vsequencer** config-object-like component to store the subsequencer handles for later retrieval. The subsequencer handles are stored as resources and retrieved by the **vseq_base** class using the simple **uvm_resource_db** API. The disadvantage of this technique is that a large number of virtual sequences might use the **uvm_resource_db** commands to retireve subsequencer handles from the resource pool and the **uvm_resource_db** (and **uvm_config_db**) API commands are rather expensive simulation-performance commands to use.

III. Hierarchical Independence

A vital issue with virtual sequencers is that they are not hierarchically independent. *Hierarchical independence* is the idea that a component in the UVM component hierarchy does not have to “know” where it is instantiated. It may be instantiated anywhere in the hierarchy, and its function is independent of its hierarchical location. Virtual sequencers are locked into a particular location in the hierarchy. They are populated with sequencer handles by referring to their hierarchical location. Any component, such as a virtual sequencer, that requires a partial or complete component path is not hierarchically independent.

To use a virtual sequencer, a virtual sequence must have access to the full hierarchical path of the virtual sequencer. Attempting to relocate the virtual sequencer requires changing pathnames to the virtual sequencer called by virtual sequences. This becomes a maintenance headache. The problem becomes more acute with larger testbenches with many interfaces.

A related issue is the debugging of virtual sequences. The sequencer handles contained in a virtual sequencer must be populated somewhere outside the virtual sequencer. Often, but not always, the handles are assigned in the top-level test. Sometimes, they are populated by an agent or an environment. A debugging challenge is finding where a particular sequencer handle is populated. The virtual sequencer does not tell you how and where the sequencer handles obtained their values.

Although less noticeable, block-level sequences are typically started on a full hierarchical path to a single sequencer and are, therefore, not hierarchically independent. Again, any repositioning of the block-level sequencer in a UVM testbench might require the sequences to be started on an existing sequencer with a new hierarchical path.

IV. Sequence Fundamentals

Sequences must be started on a sequencer. Sequences cannot be started on a driver, nor can they be started on an agent, nor can they be started on a config object, nor can they be started on any other component. Any attempt to start a sequence on anything other than a sequencer will cause a run-time failure.

Sequences are started on a specified sequencer. Each sequence has a built-in **`start()`** method to initiate the communication between the sequence and the specified sequencer. When a sequence is started, the **`m_sequencer`** handle defined in the sequence, which was inherited from the **`uvm_sequence`** base class, is set to point to the sequencer with which the sequence is communicating. Sequences conduct handshaking actions with the sequencer where it is running, and those handshaking activities are only available on a sequencer.

When coordinating multiple sequences across multiple sequencers, each sequence must have a handle to the sequencer where it is running. The sequencers themselves have different handles or must have different names.

Traditional approaches to executing sequences have required the test (or a higher-level sequence) to call **`seq.start(`*`path_to_sequencer`*`)`**. If the testbench hierarchy changes, the test must also update the ***`path_to_sequencer`*** handles referenced in the **`start()`** method. Fixed path handles are a source of bugs as a testbench evolves or as a next-generation testbench is patterned after an earlier-generation testbench.

What if the sequencer handle could update itself as the testbench structure evolves? What if multiple uniquely named sequencer handles that are referenced by virtual sequences could similarly be automatically updated? These are the ideas that are satisfied by using sequencer containers described in this paper.

V. Sequencer Containers

The role of a virtual sequencer is to serve as a container for sequencer handles. Since sequences must be started on a sequencer, the sequencer container replaces a config object, frequently used to store many other pieces of information required by a UVM testbench. As a virtual sequencer, none of the sequencer machinery is used, so there is no need to use a sequencer as a container for sequencer handles. Instead, we can use a data structure designed as a container for sequencer handles.

Our alternative is to use a data structure called a *sequencer container*. At its essence, a sequencer container is an associative array that maps names to sequencer handles. Virtual sequences can obtain a handle from the sequencer container and then search for sequencer handles stored in the container.

Sequencer containers can be implemented in a variety of ways. Various implementations provide different means for accessing the container and other ways to store and retrieve sequencer handles. In this paper, we will detail two different implementations. The first, called the *sequencer pool*, is a singleton data structure derived from the **`uvm_pool`** class for storing and retrieving sequencer handles. The second, the *sequencer aggregator*, uses multiple associative arrays and provides various means to locate sequencer handles singly or in groups.

VI. Sequencer Pool

The sequencer pool (**`sqr_pool`**) is a sequencer container. The **`sqr_pool`** is a singleton class derived from **`uvm_pool`**. It maps string names to sequencer handles, much like UVM RAL uses register names to map to register addresses. The **`sqr_pool`** example used in this paper is shown in Figure 1. The **`sqr_pool`** has methods for adding new sequencer handles and retrieving them by name. As a singleton, it is available to all virtual sequences.

This section will look closely at the organization and implementation of the sequencer pool.

A. uvm_pool Class

The **sqr_pool** class is derived from **uvm_pool**. **Uvm_pool**, part of the UVM class library, provides much of the functionality required for **sqr_pool**. The **uvm_pool** base class defines an associative array with **type KEY int** and **type T uvm_void**.

```
class uvm_pool #(type KEY=int, T=uvm_void) extends uvm_object;
  const static string type_name = "uvm_pool";
  typedef uvm_pool #(KEY,T) this_type;

  static protected this_type m_global_pool;
  protected T pool[KEY];
```

Extending the **uvm_pool** to the user-defined **sqr_pool** with **type KEY string** and **type T uvm_sequencer_base** easily creates most of the functionality required by the **sqr_pool**.

```
class sqr_pool #(type T=uvm_sequencer_base) extends uvm_pool #(string,T);
```

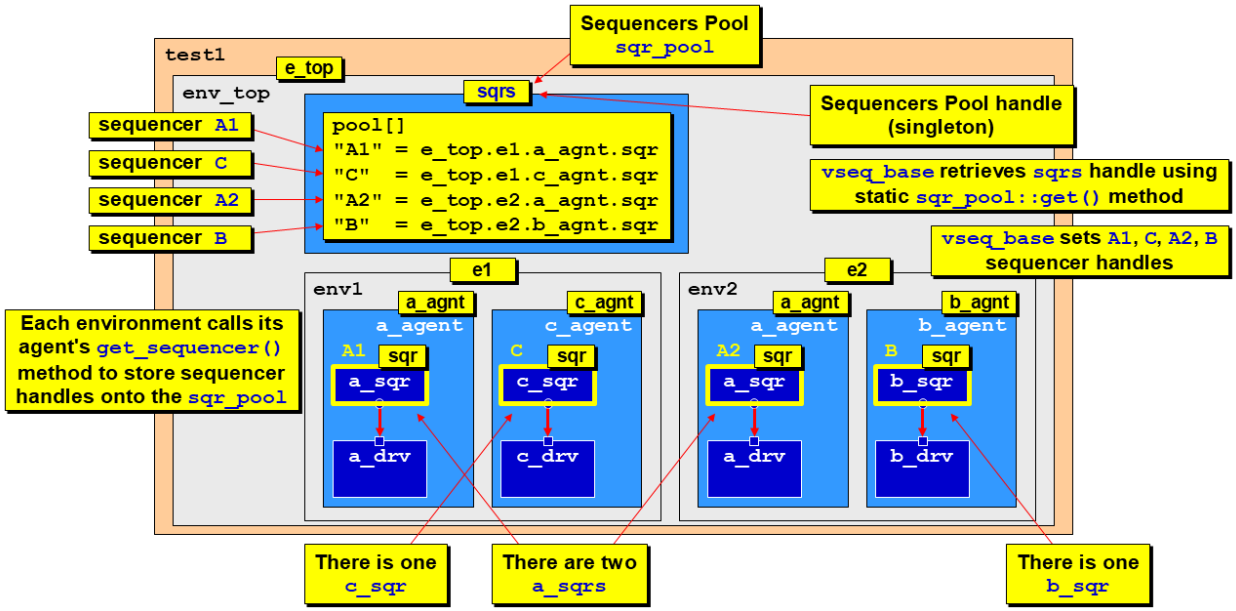

Figure 1 - Virtual sequence example - uses handles retrieved from the sequencer pool singleton.

B. sqr_pool Class Example

The virtual sequence example shown in Figure 1 is the same example shown on Siemens' Verification Academy [1], which allows the reader to compare the sequencer container techniques described in this paper to the **init_vseq** technique shown on Verification Academy. The **init_vseq** technique was briefly described in Section II-A..

This virtual sequence example has a top-level environment (**env_top**) with two sub-environments (**env1** and **env2**). Sub-environment **env1** has two agents (**a_agent** and **c_agent**) and sub-environment **env2** has two more agents (**b_agent** and a second copy of the **a_agent**).

Each agent has a **get_sequencer()** method that returns the **a** handle to the enclosed sequencer.

```
function uvm_sequencer_base get_sequencer();
  return sqr;
```

```
endfunction
```

Each UVM testbench typically has one environment, but more complex UVM testbenches might have a top-level environment and multiple lower-level environments often referred to as sub-environments. The example shown in Figure 1 has a top-level environment (**env_top**) and two sub-environments (**env1** and **env2**).

Each sub-environment has a **get_sequencers()** method that calls the **get_sequencer()** method from each enclosed agent and stores the returned sequencer handles into uniquely named locations in the **sqr_pool**, described in the next section. Note that **sqrs** is a handle to the singleton **sqr_pool**.

```
function void get_sequencers();
  sqrs.add("A1", a_agnt.get_sequencer());
  sqrs.add( "C", c_agnt.get_sequencer());
endfunction
```

Any sequencer, virtual and normal, can now locate sequencer handles in the **sqr_pool**.

The full sub-environment, agent, **sqr_pool**, **test_base** and virtual sequence classes are shown in Appendix I at the end of this paper.

C. sqr_pool Class

The sequencer pool stores all the sequencer handles used by a single or multi-level environment in an associative array named **pool**, which is accessed by **KEYs** of type **string**.

```
protected T pool[KEY];    // Inherited from uvm_pool - this is the sqr_pool
```

The **sqr_pool** overrides two virtual methods defined in the **uvm_pool** base class:

(1) **get()** – The **uvm_pool** implementation returns the item with the given key or creates a new item at that key location if one does not exist. The **sqr_pool** implementation returns the sequencer handle stored at the **key=string** location or issues a **`uvm_fatal** message. If no sequencer handle is stored at the referenced **KEY** location, nothing good is going to happen when a test tries to start a sequence on a **null** handle.

```
virtual function T get (string key);
  if (pool.exists(key)) return pool[key];
  else begin
    //print(); // Uncomment for verbose printing
    dump();
    `uvm_fatal("SQR_POOL",
    $sformatf("No pool entry exists for sqr name %s", key))
  end
endfunction
```

(2) **add()** – The **uvm_pool** version adds the given item to the associative array at the given **KEY** location and quietly overwrites the contents if there is already an item at that location. The **sqr_pool** version adds the sequencer handle to the associative array at the given **KEY** location or issues a **`uvm_fatal** message that a "Duplicate name_table entry: name %s" exists if there is a sequencer handle is already stored at that location, thus avoiding a previous handle being silently overridden. The **sqr_pool** should not allow a new assignment to overwrite the existing contents of the same location since that will delete the existing sequencer handle, which is probably used elsewhere by a virtual sequence. Again, nothing good will happen if a sequencer handle is silently lost.

```
virtual function void add(string key, uvm_sequencer_base item);
  if(key != "") begin
    if(pool.exists(key))
      `uvm_fatal("SQR_POOL",
          $sformatf("Duplicate name_table entry: name %s", key))
```

```
      pool[key] = item;
    end
  endfunction
```

The `do_print()` method is inherited from the `uvm_pool` base class. The `print()` method, which calls the `do_print` method, gives a detailed component-structure report with more information and verbosity than is typically required when debugging a UVM testbench. The `print()` method is still available but not particularly useful due to its complexity and verbosity.

The `sqr_pool` class adds a simplified and abbreviated printout of all sequencer handles stored in the sequencer pool by calling a `dump()` method.

```
virtual function void dump();
  $display("\n--- SEQUENCER POOL ENTRIES -------");
  foreach(pool[name]) begin
    uvm_sequencer_base sqr = pool[name];
    $write  ("%10s : ", name);
    $display("%s", sqr.get_full_name());
  end
  $display("--- END SEQUENCER POOL     -------\n");
endfunction
```

If the simulation is run with `+UVM_VERBOSITY=HIGH` or higher, the `test_base` calls the `dump()` method in the `start_of_simulation_phase()` and again in the `final_phase()`. Sequencer containers should be fully populated by the end of the `end_of_elaboration_phase()`. Calling `dump()` in the `start_of_simulation_phase()` helps debug sequencer handle issues if the simulation aborts during the run phases, and calling `dump()` in the `final_phase()` easily allows an engineer to view the sequencer handles that were stored and available during the simulation, at the end of the simulation in the transcript window.

```
function void start_of_simulation_phase(uvm_phase phase);
  super.start_of_simulation_phase(phase);
  if (uvm_report_enabled(UVM_HIGH)) begin
    this.print();
    factory.print();
    sqrs.dump();
  end
endfunction

function void final_phase(uvm_phase phase);
  if (uvm_report_enabled(UVM_HIGH)) sqrs.dump();
endfunction
```

D. Sequencer Pool and the Top-Level Test

When doing simple block-level testing using a single agent, The test starts a sequence on a named sequencer referenced from the `sqr_pool`. The test is not required to know the full path to the sequencer where the sequence is started. The block-level test simply references the sequencer by a user-friendly name that was assigned when the sequencer handle was stored in the `sqr_pool`.

When coordinating and executing virtual sequences across *multiple sequencers*, the test simply coordinates sequences across multiple uniquely named sequencers whose handles were stored with unique names in the `sqr_pool`. There is no need to retrieve the sequencer handles from a virtual sequencer, or from stored resources using the `uvm_resource_db` API.

The capability to consistently access sequencer handles the same way in both simple sequences or more complex virtual sequences has been the missing link to achieving a simple and unified sequence execution strategy. A globally accessible sequencer pool addresses this missing unified strategy.

E. Uniquely Named Sequencers Limitation?

The **sqr_pool** technique described in this section requires all the testbench sequencers to have unique string-names in the **sqr_pool**. The virtual sequencer technique (described in Section II-B) allows sequencer handles in different virtual sequencers to have the same name. This means the virtual sequencer technique runs the risk that virtual sequences can be started on the correct sequencer-type but on the wrong virtual sequencer. This might be a difficult bug to identify.

Because time spent debugging is often the #1 factor that puts a project behind schedule [8], a coding style that helps avoid bugs in the first place is a worthwhile investment. Taking time to plan to ensure sequencers have unique names is not a burdensome requirement. Executing sequences on the uniquely named sequencers helps to avoid running the wrong virtual sequence on a name that might exist in more than one virtual sequencer.

Since each agent, even agents in arrays, will add its sequencer handle onto the pool, the handles will be unique. One must ensure all names associated with sequencer handles, including those in arrays are unique and easy to understand.

VII. Sequencer Aggregator

The sequencer aggregator is another sequencer container. Like the sequencer pool, it is a container for sequencer handles. It performs the same function as a sequencer pool – supplying sequencer handles to virtual sequences. Whereas the sequencer pool is a singleton, the sequencer aggregator is not, allowing for multiple aggregators. Multiple aggregators allow for separate sequencer domains and namespaces. The sequencer aggregator is called an aggregator because it enables you to aggregate a collection of sequencer handles as the testbench is created (during the UVM build and connect phases).

This section will look closely at the organization and implementation of a sequencer aggregator.

The sequencer aggregator has multiple associative arrays containing sequencer handles. These are the core of the sequencer aggregator.

```
class sqr_aggregator;
  typedef uvm_sequencer_base sqr_q_t[$];

  local uvm_sequencer_base sqr_table[string];
  local uvm_sequencer_base name_table[string];
  local sqr_q_t kind_table[string];
```

The methods in the **sqr_aggregator** class can add a new sequencer handle to the aggregator or find a handle or group of handles based on some criteria. Each of the three associative arrays, **sqr_table**, **name_table**, and **kind_table**, stores sequencer handles differently so that they can be retrieved differently. The **name_table** array lets you look up sequencers by an assigned name. The **kind_table** enables you to look up groups of sequencers by assigned kind. Finally, **sqr_table** lets you look up sequencers by full path name. All three tables are updated when a new sequencer handle is added to an aggregator.

```
  function void add(uvm_sequencer_base sqr, string name, string kind);
    sqr_q_t q;
    string path = sqr.get_full_name();
    sqr_table[path] = sqr;
```

```
    if(kind != "") begin
      if(kind_table.exists(kind))
        q = kind_table[kind];
      q.push_back(sqr);
      kind_table[kind] = q;
    end

    if(name != "") begin
      if(name_table.exists(name))
          `uvm_info("SQR_AGGREGATOR",
                    $sformatf("replacing sequencer with name %s", name),
                    UVM_NONE)
      name_table[name] = sqr;
    end

  endfunction
```

The sequencer aggregator has four lookup functions, one for each of the different ways to look up sequencers. First, `lookup_path()` finds a sequencer by full path name. Finding sequencer handles by full path name is not used often because requiring full path names breaks hierarchical independence. However, it can be helpful in some situations and for debugging.

```
  function uvm_sequencer_base lookup_path(string path);
    if(sqr_table.exists(path))
      return sqr_table[path];
    else
      return null;
  endfunction
```

The `lookup_name()` function returns a single sequencer handle whose name matches a name string. The name is assigned when the sequencer handle is added to the aggregator.

```
  function uvm_sequencer_base lookup_name(string name);
    if(name_table.exists(name))
      return name_table[name];
    else
      return null;
  endfunction
```

`lookup_path()` finds a group of zero or more sequencer handles whose path name matches a regular expression. The UVM regular expression facility, which supports posix regular expression syntax, compiles and matches regular expressions. `lookup_path()` is useful when you want to locate a collection of sequencers whose hierarchical location is subordinate to a particular component, for example.

```
  function sqr_q_t lookup_path_regex(string regex);
    sqr_q_t q = {};
    foreach(sqr_table[path]) begin
      if(uvm_re_match(regex, path))
       q.push_back(sqr_table[path]);
    end
    return q;
  endfunction
```

Finally, **lookup_kind()** lets you find a group of zero or more sequencer handles that all have the same **kind** string. A "kind" is an arbitrary string assigned to sequencer handles as they are added to the aggregator.

```
function sqr_q_t lookup_kind(string kind);
  return kind_table[kind];
endfunction
```

For completeness and debugging our implementation of a sequencer aggregator contains a dump() function. This function prints out all the internal tables.

```
function void dump();

  $display("--- SEQUENCER AGGREGATOR ---");

  $display("  by name:");
  foreach(name_table[name]) begin
    uvm_sequencer_base sqr = name_table[name];
    $display("    %s -> %s", name, sqr.get_full_name());
  end

  $display("  by kind:");
  foreach(kind_table[kind]) begin
    sqr_q_t q = kind_table[kind];
    $display("    %s", kind);
    foreach(q[i]) begin
         uvm_sequencer_base sqr = q[i];
         $display("      %s", sqr.get_full_name());
    end
  end

  $display("  by path:");
  foreach(sqr_table[path]) begin
    $display("    %s", path);
  end

endfunction
```

A working example of the sequencer aggregator is in [6] and more discussion can be found in [5].

VIII. Sequencer Aggregator and Pool Comparison

The sequencer pool and sequence aggregator store sequencer handles so virtual sequences can retrieve and use them. The choice of which to use is application-dependent.

The sequencer pool is a singleton that does not have to be explicitly instantiated. It is immediately available to any object, specifically virtual sequences. It has only one associative array for storing and retrieving sequencer handles, providing a single namespace for sequencer names. All sequencer handles put into the sequencer pool must have unique names. In large-scale systems, there is a risk of a sequencer name clash. A fatal error occurs if a second (or later) sequencer handle is inserted with the same name. This prevents the sequencer pool from silently handing back the wrong sequencer handle. Also, the sequencer pool provides no means to look up groups of sequencer handles by regular expression or kind.

Sequencer aggregators, on the other hand, must be explicitly instantiated and stored in the resource database so that virtual sequences can find them. However, they provide more ways to locate sequencer handles, such as in groups.

The sequencer pool is helpful for a wide variety of applications, particularly in testbenches for relatively simple devices with a small number of interfaces (and thus a small number of sequencers). Sequencer aggregators are best with large-scale designs and multiple interface domains.

Sequencer aggregators are frequently required in an environment where multiple block-level testbenches are built independently and later integrated to form subsystem-level and system-level testbenches. In that development style, it is important to maintain hierarchical independence and ensure no accidental name clashes.

IX. Sequencers Interfaces

UVM agents have three primary interfaces, as shown in Figure 2: the virtual interface (`vif`) for connecting signals on the DUT, an analysis port for sending sampled transactions to analysis components, and a sequencer interface for initiating transactions. The virtual interface and the analysis port are visible on the boundary of the agent. However, the sequencer interface is not. "Reaching in" (a hierarchical reference) is often used by using dot notation to access the agent's class member that holds the sequencer handle. Allowing unrestricted access to the sequencer handle is awkward and can be dangerous.

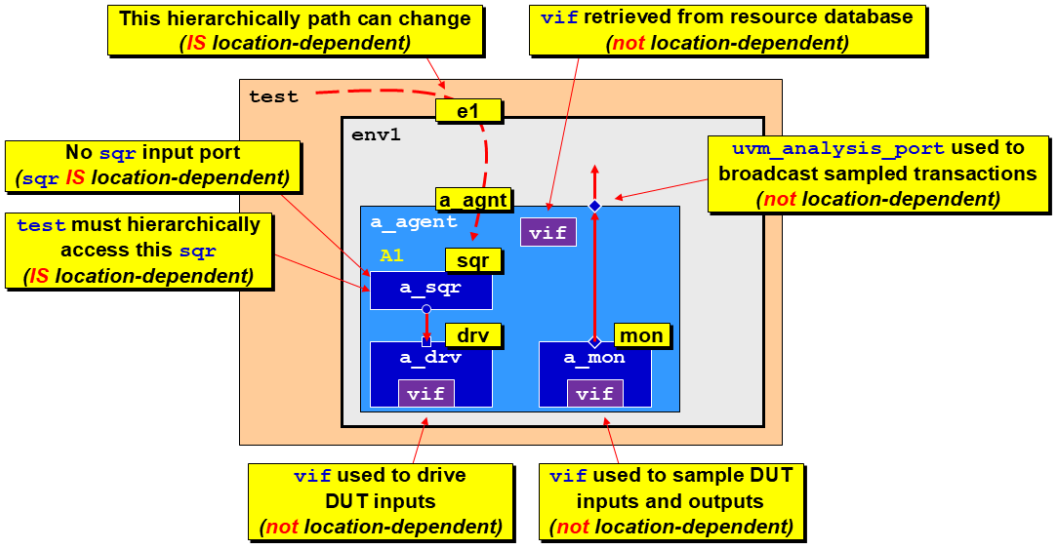

Figure 2- Sequencer interfaces block diagram

We propose a convention where each agent has a `get_sequencer()` method, which returns a handle to the sequencer contained in the agent, and each environment has a `get_sequencers()` method to populate the sequencer container with sequencers in the contained agents. The convention requires that the `get_sequencer()` method be called for each agent in the environment and adds the sequencer handles into the sequencer container(s). An agent's `get_sequencer()` method simply returns the agent's sequencer handle.

```
class agent extends uvm_component;

  local uvm_sequencer#(transaction) sqr;

  function uvm_sequencer_base get_sequencer();
    return sqr;
  endfunction
```

An environment that contains multiple agents populates the sequencer aggregator or sequencer pool with all the sequencer handles from the agents. Note that get_sequencers() takes a reference argument to a sequencer aggregator. It is important that a ref argument is used. This prevents unnecessary copying. If you are using a sequencer pool, the argument is not necessary. Instead, the sequencers are added to the singleton pool.

```
class multi_if_env extends uvm_component;

local agent agt_A;
```

```
  local agent agt_B;

  function void get_sequencers(ref sqr_aggregator sqrs);
    sqrs.add(agt_A.get_sequencer(), "control", "control");
    sqrs.add(agt_B.get_sequencer(), "data", "data");
  endfunction

endclass
```

In the example above, two sequencers are added to the aggregator, one from agent A and one from agent B. Each agent's get_sequencer function is called to retrieve its sequencer handles to be passed to the aggregator's **`add()`** function.

The top-level test creates a new aggregator and populates it with sequencer handles by calling **`get_sequencers()`** in the environment. It does this in the **`end_of_elaboration`** phase after the entire component hierarchy is in place.

```
class test extends uvm_component;

  local sqr_aggregator sqrs;

  function void end_of_elaboration_phase(uvm_phase phase);
    sqrs = new();
    env.get_sequencers(sqrs);
    uvm_resource_db#(sqr_aggregator)::set("*", "sqrs",  sqrs, this);
  endfunction

endclass
```

The test also puts the sequencer aggregator into the resource database to make it available to virtual sequences. This step is not necessary if you are using the sequencer pool.

X. Verification IP (VIP) Considerations

Verification Intellectual Property (VIP) is often purchased or leased from third party vendors and frequently complicates advanced verification environments. The problem is that user sequences must know the full path names to the sequencers inside of the VIP agents to execute user-defined sequences on the VIP agent-sequencers. Once the path name to the VIP agent-sequencer is known, engineers frequently store that handle inside of a virtual sequencer, which is one reason that the **`vsequencer`** technique shown in Section II-B, is so commonly used by verification engineers.

It is recommended that VIP vendors add the simple **`get_sequencer()`** method to the VIP-agent. Users could then call the agent-**`get_sequencer()`** method from the user's environment code to store the VIP sequencer handle in the **`sqr_pool`** for easy reference from the user's block-level sequences and multi-block level virtual sequences.

XI. Summary

Virtual sequencers were invented as a use model for virtual sequences in OVM. They are not hierarchically independent, thwarting reuse and testbench composition. They are also notoriously difficult to debug.

Sequencer containers mitigate the issues of using virtual sequencers. Sequencer containers hold a collection of sequencer handles mapped to string names. Virtual sequences can retrieve sequencer handles from sequencer containers instead of virtual sequencers.

We have outlined two implementations of sequencer containers, the sequencer aggregator (**`sqr_aggregator`**) and sequencer pool (**`sqr_pool`**). The singleton sequencer pool is a good choice when it is feasible to assign unique names to all sequencer handles in a testbench. Sequencer aggregators are a good choice when there are multiple interface domains or testbenches block-level testbenches will be composed into subsystem and system-level testbenches.

Treating sequencers as agent interfaces, like virtual interfaces and analysis ports, enables testbench composition and protects agents from improper access.

The use model for sequencer containers requires that sequencer handles can be located in agents and environments in a uniform manner. We proposed that all agents include a **`get_sequencer()`** method to return its sequencer and each environment includes a **`get_sequencers()`** method to populate the sequencer containers from its contained agents and sub-environments.

Finally, the sequencer container techniques described in this paper simplify and unify how sequences and virtual sequences can be run in a UVM testbench.

XII. Errata & Changes

Readers are encouraged to send email to Cliff Cummings (cliffc@sunburst-design.com) any time they find potential mistakes or if they would like to suggest improvements. Cliff is always interested in other techniques that engineers are using.

A. Revision 1.1 (Post-DVCon2025) - What Changed?

Mark Glasser pointed out that the original code in the **`vseq_base`** class improperly used the **`body()`** task to set the subsequencer handles and then each virtual sequence would improperly call **`super.body()`** from its own **`body()`** task. This is not a proper use of the sequence **`body()`** task in either the **`vseq_base`** class or an extended virtual sequence.

The **`vseq_base`** class in Appendix E now has a proper **`set_sequencer_handles() void function`** method that performs the same actions when called by a virtual sequence, as shown in Appendix F.

XIII. References

[1] Verification Academy Video – Layered Sequences (init_vseq() technique). Video available at https://verificationacademy.com/topics/uvm-universal-verification-methodology/advanced-uvm/layered-sequences/

[2] Clifford E. Cummings, Heath Chambers, Mark Glasser, *"The Untapped Power of UVM Resources and Why Engineers Should Use the uvm_resource_db,"* DVCon 2023 Proceedings, also available at www.sunburst-design.com/papers/CummingsDVCon2023_uvm_resource_db_API.pdf.

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

[4] "IEEE Standard For Universal Verification Methodology Language Reference Manual," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800.2™-2023

[5] Mark Glasser, *Next Level Testbenches – Design Patterns In SystemVerilog and UVM*, July 2024, ISBN: 9798332981975, Published by Mark Glasser.

[6] Mark Glasser, Topaz Library, `https://github.com/mxg/topaz`

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

[8] Wilson Research Group and Siemens EDA study references – *(See Debug Time pie charts)*:
https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/
https://blogs.sw.siemens.com/verificationhorizons/2022/11/06/part-4-the-2022-wilson-research-group-functional-verification-study/
https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/
https://blogs.sw.siemens.com/verificationhorizons/2020/12/02/part-4-the-2020-wilson-research-group-functional-verification-study/
https://blogs.sw.siemens.com/verificationhorizons/2019/01/29/part-8-the-2018-wilson-research-group-functional-verification-study/

https://blogs.sw.siemens.com/verificationhorizons/2019/01/02/part-4-the-2018-wilson-research-group-functional-verification-study/
https://blogs.sw.siemens.com/verificationhorizons/2016/10/04/part-8-the-2016-wilson-research-group-functional-verification-study/
https://blogs.sw.siemens.com/verificationhorizons/2016/08/29/part-3-the-2016-wilson-research-group-functional-verification-study/

XIV. Author & Contact Information

Sunburst Design World Class Training

Sunburst Design merged with Paradigm Works in February of 2020 and still provides World Class SystemVerilog, Synthesis and UVM Verification training. For more information about SystemVerilog and UVM training, contact Cliff Cummings (cliffc@sunburst-design.com) or Michael Hoyt (michael.hoyt@paradigm-works.com)

Paradigm Works Expert Design Services, Verification Services & IP

Paradigm Works provides expert services in Semiconductor Architecture, Design, Synthesis, Functional Verification, and DFT. For more information about Paradigm Works services, contact Michael Hoyt at michael.hoyt@paradigm-works.com

Cliff Cummings is Vice President of Training at Paradigm Works and Founder of Sunburst Design. Paradigm Works and Sunburst Design merged in February of 2020. Cliff has more than 40 years of ASIC, FPGA and system design experience and more than 30 years of combined Verilog, SystemVerilog, UVM verification, synthesis, and methodology training experience.

Cliff has presented more than 100 SystemVerilog seminars and training classes in the past 20 years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Cliff participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee from 1994-2012, and has presented more than 50 papers on SystemVerilog & SystemVerilog related design, synthesis, and UVM verification techniques.

Cliff holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Email address: cliffc@sunburst-design.com

Mark Glasser is a Principal Consulting Engineer at Paradigm Works, where he develops and deploys testbenches and verification methodologies. Mark's experience in functional verification spans more than 35 years.

Mark has been involved in developing various standards, including IP-Xact, SystemC, and UVM. He was one of the original architects of UVM and a significant contributor. Mark is the author of "Next Level Testbenches – Design Patterns in SystemVerilog and UVM" (2024) and "Open Verification Methodology Cookbook" (Springer, 2009) and numerous papers on functional verification topics. In addition, Mark is a co-author of two patents. Mark is a frequent presenter and panelist at verification-related conferences and a presenter at DVClub.

Mark holds a BSCS degree from California State University, Northridge.

Email address: makr.glasser@paradigm-works.com

Last Updated: March 2025

Appendix I. Required Sequencer Pool Coding Modifications

What coding requirements are needed to implement the sequencer pool technique? This Appendix shows the simple modifications required to implement the sequencer pool technique.

A. Agent classes

Each agent declares handles for the sequencer and the driver. It is common to use the handle names **sqr** and **drv**. In the sequencer pool technique, the agent adds the **get_sequencer()** method to return the handle for the declared **sqr** sequencer. The **get_sequencer()** method requires three very simple lines of code shown on Lines 26-28 in Example 1. The full **agent** class code is shown below.

```
 1 class agent extends uvm_component;
 2   `uvm_component_utils(a_agent)
 3
 4   a_driver    drv;
 5   a_sequencer sqr;
 6
 7   function new(string name, uvm_component parent);
 8     super.new(name, parent);
 9   endfunction
10
11   function void build_phase(uvm_phase phase);
12     drv =    a_driver::type_id::create("drv", this);
13     sqr = a_sequencer::type_id::create("sqr", this);
14   endfunction
15
16   function void connect_phase(uvm_phase phase);
17     drv.seq_item_port.connect(sqr.seq_item_export);
18   endfunction
19
20   //------------------------------------------------------------------------
21   // The environment will ask this agent for the sqr handle to
22   //     be stored in the sqr_pool.
23   // The environment will call this agnt.get_sequencer() method to retrieve
24   //     the sequncer handle from this agent.
25   //------------------------------------------------------------------------
26   function uvm_sequencer_base get_sequencer();
27     return sqr;
28   endfunction
29 endclass
```

Example 1 - sqr_pool: agent-example code

B. Envrionment classes

Each environment that builds the agent(s) includes its own **get_sequencers()** method that calls the individual **get_sequencer()** method from each agent and stores the retrieved sequencer handle into a uniquely named string location in the **sqr_pool** singleton. The full **env1** class code is shown in Example 2.

Lines 21-24 - defines the **get_sequencers()** method used in the **env1** block diagram as shown in Figure 3.

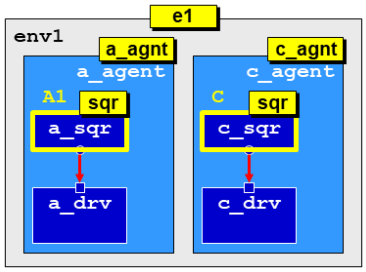

Figure 3 - env1 block diagram

It is the job of the environment(s) to do the naming of the subsequencer handles used by the **sqr_pool**. The agent(s), including any VIP agents, are not required to know what name was used to store the agent-sequencer handles.

```
 1 class env1 extends uvm_env;
 2   `uvm_component_utils(env1)
 3
 4   a_agent a_agnt;
 5   c_agent c_agnt;
 6
 7   function new(string name, uvm_component parent);
 8     super.new(name, parent);
 9   endfunction
10
11   function void build_phase(uvm_phase phase);
12     a_agnt = a_agent::type_id::create("a_agnt", this);
13     c_agnt = c_agent::type_id::create("c_agnt", this);
14   endfunction
15
16   //------------------------------------------------------------------
17   // Wait for build_phase to complete then grab and store the subsequencer
18   // handles into uniquely named locations in the singleton sqr_pool
19   //------------------------------------------------------------------
20
21   function void get_sequencers();
22     sqrs.add("A1", a_agnt.get_sequencer());
23     sqrs.add( "C", c_agnt.get_sequencer());
24   endfunction
25   //-------------------------------------------------------
26   // In this example, the stored sqr handles are:
27   //-------------------------------------------------------
28   // A1 Sequencer handle is stored in the sqr_pool:
29   // pool["A1"] : "e_top.e1.a_agnt.sqr"
30   //-------------------------------------------------------
31   // C Sequencer handle is stored in the sqr_pool:
32   // pool["C"]  : "e_top.e1.c_agnt.sqr"
33   //-------------------------------------------------------
34 endclass
```

Example 2 - sqr_pool: environment-example code

C. sqr_pool class

The **sqr_pool** class is a singleton extended from the functionality already defined in the **uvm_pool** base class, as described in Section VI-A. The full **sqr_pool** class code is shown in Example 3.

Line 1 – parameterizes the **sqr_pool** class to the **uvm_sequencer_base** type, which is extended from the **uvm_pool** base class parameterized with the **KEY** type **string**.

Line 3 – declare **this_type** to be a **sqr_pool** parameterized to the **uvm_sequencer_base** type.

Line 5 – declare the **protected** singleton **m_global_pool** handle.

Line 6 – shows that the **protected pool** with access type **string** is inherited from the **uvm_pool** base class. **pool** is the singleton **sqr_pool** handle referenced inside of this class.

Lines 9-11 – implements a **protected new()** constructor, which restricts calling this constructor from anything, but the **static get_global_pool()** method that starts on line 13.

Lines 13-17 – defines the **static get_global_pool()** method. The first time this method is called, the **m_global_pool** handle will be **null** so this **static** method will call the **protected new()** constructor to create the one and only (singleton) copy of the **m_global_pool** handle. On line 16, the first time and every other time this **static** method is called, it will return the singleton handle to the **sqr_pool** (**m_global_pool**).

Lines 19-24 – defines a **static get_global(KEY)** method to return the **m_global_pool** handle of a **string**-parameterized version of the **sqr_pool**. This appears to be a safety function in case multiple **sqr_pools** of types other than **string** happen to exist in the UVM testbench. We do not have other **sqr_pool#(KEY)** types in this example so this method is not called from this example and might never be called from a UVM testbench.

Lines 26-34 – defines the **virtual get()** method to retrieve a user-named sequencer handle from the **sqr_pool**. If a handle by this name does not exist in the **sqr_pool**, the simulation will report a missing **`uvm_fatal** message and abort the simulation.

Lines 36-43 – defines the **virtual add()** method to push a uniquely user-named sequencer handle onto the **sqr_pool**. If there is already a sequencer by this name in the **sqr_pool**, the simulation will report a duplicate **`uvm_fatal** message and abort the simulation.

Lines 45-53 – defines the **virtual dump()** function to print out in a simple format, the current contents of the **sqr_pool** showing the unique user-defined sequencer names and the actual path to each sequencer.

```
 1 class sqr_pool #(type T=uvm_sequencer_base) extends uvm_pool #(string,T);
 2
 3   typedef sqr_pool #(T) this_type;
 4
 5   static protected this_type m_global_pool;
 6   // protected T pool[KEY];    // Inherited - this is the sqr_pool
 7
 8   // Should the new-constructor be protected to make sqr_pool a singleton?
 9   protected function new (string name="");
10     super.new(name);
11   endfunction
12
13   static function this_type get_global_pool ();
14     if (m_global_pool==null)
15       m_global_pool = new("pool");
16     return m_global_pool;
17   endfunction
18
19   // KEY is the string type passed as a parameter to uvm_pool
20   static function T get_global (KEY key);
21     this_type gpool;
22     gpool = get_global_pool();
23     return gpool.get(key);
24   endfunction
25
26   virtual function T get (string key);
27     if (pool.exists(key)) return pool[key];
28     else begin
```

```
29        //print();
30        dump();
31        `uvm_fatal("SQR_POOL",
32        $sformatf("No pool entry exists for sqr name %s", key))
33      end
34    endfunction
35
36    virtual function void add(string key, uvm_sequencer_base item);
37      if(key != "") begin
38        if(pool.exists(key))
39          `uvm_fatal("SQR_POOL",
40          $sformatf("Duplicate name_table entry: name %s", key))
41        pool[key] = item;
42      end
43    endfunction
44
45    virtual function void dump();
46      $display("\n--- SEQUENCER POOL ENTRIES -------");
47      foreach(pool[name]) begin
48        uvm_sequencer_base sqr = pool[name];
49        $write  ("%10s : ", name);
50        $display("%s", sqr.get_full_name());
51      end
52      $display("--- END SEQUENCER POOL     -------\n");
53    endfunction
54  endclass
```

Example 3 - sqr_pool class code

D. test_base class

Key features of the **test_base** class are described in this section. Technically, there is no need to reference the **sqr_pool** from the **test_base** class but the ability to print all the stored sequencer handles before the run phases and again at the very end of the simulation are especially useful debugging tools if something goes wrong during the simulation. The dumping for the stored sequencer handles only happens when the user adds **+UVM_VERBOSITY=HIGH** (or higher) to the simulation command line, so these printouts do not happen by default. The full **test_base** class code is shown in Example 4.

Line 4 – creates a **sqr_pool_type** abbreviation defined to be the **sqr_pool** parameterized to the **uvm_sequencer_base** type, which is used on line 7.

Line 7 – declares a **sqrs** handle of the **sqr_pool_type** and retrieves the singleton **sqr_pool** handle using the static **get_global_pool()** method defined in the **sqr_pool** class.

Line 23 – dumps the stored sequencer handles just before the simulation run phases. It is useful to have this list of handles for debugging purposes if the run phases abort with **null** pointer references.

Line 28 – dumps to the transcript window the stored sequencer handles at the end of the simulation. This makes the stored sequencer handles easily visible for examination at the end of the simulation.

```
1  class test_base extends uvm_test;
2    `uvm_component_utils(test_base)
3
4    typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;
5
6    uvm_factory   factory =   uvm_factory::get();
```

```
 7   sqr_pool_type sqrs    = sqr_pool_type::get_global_pool();
 8   env_top       e_top;
 9
10   function new(string name, uvm_component parent);
11     super.new(name, parent);
12   endfunction
13
14   function void build_phase(uvm_phase phase);
15     e_top = env_top::type_id::create("e_top", this);
16   endfunction
17
18   function void start_of_simulation_phase(uvm_phase phase);
19     super.start_of_simulation_phase(phase);
20     if (uvm_report_enabled(UVM_HIGH)) begin
21       this.print();
22       factory.print();
23       sqrs.dump();
24     end
25   endfunction
26
27   function void final_phase(uvm_phase phase);
28     if (uvm_report_enabled(UVM_HIGH)) sqrs.dump();
29   endfunction
30 endclass
```

Example 4 - sqr_pool: test_base-example code

After the **build_phase()**, **connect_phase()**, and **end_of_elaboration_phase()**, and just before executing the **run_phases()**, UVM executes the **start_of_simulation_phase()**, which we often refer to as the pre-run phase. If something goes wrong during the UVM testbench simulation, it frequently happens during the **run_phases()**, so an engineer can re-run a simulation with **+UVM_VERBOSITY=HIGH** (or higher), which does the following:

this.print()	(Line 21) prints out the entire testbench hierarchical structure.
factory.print()	(Line 22) prints the contents of the factory.
sqrs.dump()	(Line 23) prints the contents of the **sqr_pool** to show the sequencers that are available during the simulation.

At the end of the simulation after all running is complete, and if **+UVM_VERBOSITY=HIGH** (or higher) has been added to the simulation command, the **final_phase()** can be used to conveniently print out the contents of the **sqr_pool** just before the simulation finishes.

Printing out the testbench structure, the factory contents and the **sqr_pool** list of available sequencers can be useful debugging aids.

E. vseq_base class

As is typical for virtual sequence base classes, The **vseq_base** class declares handles for the subsequencers, but the declarations are of the **uvm_sequencer_base** class type, which permits any extended and parameterized sequencer class handle to be copied to the **uvm_sequencer_base** class type. The handle names are not important, but we have selected handle names that will match the **sqr_pool** string names that were used when **env1** and **env2** stored their respective subsequencer handles. The full **vseq_base** class code is shown in Example 5.

Line 4 – creates a **sqr_pool_type** abbreviation defined to be the **sqr_pool** parameterized to the **uvm_sequencer_base** type.

Lines 6-9 – declare subsequencer handles of the **uvm_sequencer_base** type for the four subsequencers used in this example.

Line 10 – declares a **sqrs** handle of the **sqr_pool_type** and retrieves the singleton **sqr_pool** handle using the static **get_global_pool()** method defined in the **sqr_pool** class.

Lines 16-21 – defines the **body()** task in the **vseq_base** class. The **body()** task retrieves the stored subsequencer handles by their unique name from the **sqrs sqr_pool**.

```
 1 class vseq_base extends uvm_sequence #(uvm_sequence_item);
 2   `uvm_object_utils(vseq_base)
 3
 4   typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;
 5
 6   uvm_sequencer_base A1;
 7   uvm_sequencer_base A2;
 8   uvm_sequencer_base B;
 9   uvm_sequencer_base C;
10   sqr_pool_type sqrs = sqr_pool_type::get_global_pool();
11
12   function new(string name = "vseq_base");
13     super.new(name);
14   endfunction
15
~~16   task body();  // modified in Rev 1.1~~
16   function void set_sqr_handles();
17     A1 = sqrs.get("A1");
18     A2 = sqrs.get("A2");
19     B  = sqrs.get("B");
20     C  = sqrs.get("C");
21   endfunction
~~21   endtask       // modified in Rev 1.1~~
22 endclass
```

Example 5 - sqr_pool: vseq_base-example code

F. virtual sequence classes

Shown in Example 6 and Example 7 are two virtual sequences that extend from the **vseq_base** class.

In the first virtual sequence example shown in Example 6:

Line 3 – (blank) is where the subsequencer handles have been inherited from the **vseq_base** class.

Lines 9-11 - the **body()** task first declares and factory-creates two **a**-sequence handles and one **b**-sequence handle.

Line 13 – calls the **vseq_base body()** (**super.body()**) method that sets the inherited subsequencer handles.

Lines 15-20 – runs sequence **a** on sequencer **A1**, followed by sequences **b** and **a2** running in parallel on sequencers **B** and **A2** respectively, followed by sequence **a** again being run on sequencer **A1**.

```
1 class vseq_A1_B_A2_A1 extends vseq_base;
2   `uvm_object_utils(vseq_A1_B_A2_A1)
3
4   function new(string name="vseq_A1_B_A2_A1");
5     super.new(name);
6   endfunction
7
```

```
 8   task body();
 9     a_seq  a = a_seq::type_id::create("a");
10     b_seq  b = b_seq::type_id::create("b");
11     a_seq a2 = a_seq::type_id::create("a2");
12
13     super.body();  // modified in Rev 1.1
13     set_sequencer_handles();
14
15     a.start(A1);
16     fork
17        b.start(B);
18       a2.start(A2);
19     join
20     a.start(A1);
21   endtask
22 endclass
```

Example 6 - sqr_pool: virtual sequence example 1

In the second virtual sequence example shown in Example 7:

Line 3 – (blank) is where the subsequencer handles have been inherited from the **vseq_base** class.

Lines 9-11 - the **body()** task first declares and factory-creates one **a**-sequence handle, one **b**-sequence handle and one **c**-sequence handle.

Line 13 – calls the **vseq_base body()** (**super.body()**) method that sets the inherited subsequencer handles.

Lines 15-19 – runs sequence **a** on sequencer **A1**, followed by sequences **b** and **c** running in parallel on sequencers **B** and **C** respectively.

```
 1 class vseq_A1_B_C extends vseq_base;
 2   `uvm_object_utils(vseq_A1_B_C)
 3
 4   function new(string name = "vseq_A1_B_C");
 5     super.new(name);
 6   endfunction
 7
 8   task body();
 9     a_seq a = a_seq::type_id::create("a");
10     b_seq b = b_seq::type_id::create("b");
11     c_seq c = c_seq::type_id::create("c");
12
13     super.body();
14
15     a.start(A1);
16     fork
17       b.start(B);
18       c.start(C);
19     join
20   endtask
21 endclass
```

Example 7 - sqr_pool: virtual sequence example 2

Appendix II. Sequencer naming conventions

One way to ensure unique names for the sequencer handles stored in the **sqr_pool** is to use either the software camelCase naming convention that includes the environment class or handle name followed by the desired sequencer name with uppercase letter as the first character of the desired sequencer name, or the software snake_case naming convention that includes the environment class or handle name followed by underscore and the desired sequencer name.

Example from the **env1** class - camelCase:

```
virtual function void get_sequencers();
  sqrs.add("env1A", a_agnt.get_sequencer());
  sqrs.add("env1C", c_agnt.get_sequencer());
endfunction
```

Example 8 - env1 environment sequencer camelCase naming convention

Example from the **env1** class – snake_case:

```
virtual function void get_sequencers();
  sqrs.add("env1_A", a_agnt.get_sequencer());
  sqrs.add("env1_B", c_agnt.get_sequencer());
endfunction
```

Example 9 - env1 environment sequencer snake_case naming convention

Examples from the **env2** class - camelCase:

```
virtual function void get_sequencers();
  sqrs.add("env2A", a_agnt.get_sequencer());
  sqrs.add("env2B", b_agnt.get_sequencer());
endfunction
```

Example 10 - env2 environment sequencer camelCase naming convention

Examples from the **env2** class – snake_case:

```
virtual function void get_sequencers();
  sqrs.add("env2_A", a_agnt.get_sequencer());
  sqrs.add("env2_B", b_agnt.get_sequencer());
endfunction
```

Example 11 - env2 environment sequencer snake_case naming convention

Since each environment class or handle name is unique within an upper-level environment, each lower-level environment can independently name the stored sequencers with names that will be unique to the entire testbench.