A Summary and Examination of UVM Virtual Sequence Techniques

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

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

Smita Kulkarni
Paradigm Works, Inc.
smita.kulkarni@paradigm-works.com

***Abstract* - UVM Virtual Sequences are used to coordinate stimulus activity across multiple design interfaces. The newest virtual sequence technique utilizes a sequencer pool, which simplifies and unifies the execution of sequences and virtual sequences. The other three virtual sequence techniques described in this paper are (1) the Virtual Sequencers technique, (2) the test_base init_vseq Technique, and (3) storing and Retrieving Sequencer Handles in the UVM Resource Database.**

This paper summarizes and examines: (1) the four virtual sequence techniques. (2) Why engineers should use one of the sequencer aggregator techniques. (3) Why the most commonly used Virtual Sequencer technique is no longer recommended.

I. Introduction

The first publication that included virtual sequences and virtual sequencers was "A Practical Guide to Adopting the Universal Verification Methodology" by Kathleen Meade and Sharon Rosenberg, published in 2010 and revised in 2013 [1]. Ray Salemi mentioned virtual sequences and virtual sequencers in his 2013 book, "The UVM Primer" [2].

For almost a decade, engineers have been utilizing the virtual sequencer (`vsequencer`) technique, as described by Cliff Cummings and Janick Bergeron in their 2016 DVCon paper [3]. Although that technique has proven effective in numerous UVM verification environments, this paper explains why it is now obsolete and why engineers should discontinue its use.

II. Sequencers and Sequences

A. Sequences

Sequences, essential elements of a UVM-based testbench, are responsible for generating stimuli directed at the DUT. Each sequence contains a `body()` task whose role is to create new stimuli to send to the DUT. When a sequence launches, the `body()` task is executed (after some preliminaries). The expectation is that `body()` will produce one or more sequence items, units of stimulus, to be sent to the DUT. Often, a sequence will produce a stream of sequence items in some order or sequence. Hence, the name "sequence."

A sequence item (item) is a class object whose members are used to control the DUT in some way. A sequence creates new items and populates them before sending them downstream. Often, sequences use randomization and constraints to populate items.

In a UVM testbench, sequences are not connected directly to the DUT. Sequences connect to a sequencer, which, in turn, is connected to a driver. The driver connects to the DUT. The role of the driver is to convert items into a pin-level protocol that the DUT understands.

The role of the sequencer is to forward items to the driver. If UVM allowed only one sequence to connect to a driver, we would not need sequencers. We could connect the sequence directly to the driver. The sequencer allows multiple concurrent sequences to send sequence items to a driver. The sequencer is akin to an abstract bus with multiple initiators and a single target. Each initiator (sequence) generates stimulus (sequence items) independently of other initiators. The sequencer forwards the items to the target (driver). The sequencer is responsible for arbitration among concurrent sequences, ensuring that the order of items is correct according to the chosen arbitration algorithm. The sequencer also forwards responses from the DUT back to the appropriate sequence.

B. Sequencers

Sequencers, unlike sequences, are part of the UVM component hierarchy. The class `uvm_sequencer#()` is derived from `uvm_component` (with `uvm_sequencer_param_base#()` and `uvm_sequencer_base` in between in UVM-

1800.2-2020 [4]). Like all components, sequencers are created in the `build_phase()` and persist throughout the remainder of the simulation. Sequences, on the other hand, are not derived from `uvm_component` and have a lifetime that could be a single transaction or for the entire simulation time.

Concurrent sequences are helpful in many contexts. Multiple independent concurrent streams of stimuli could be used to model background traffic, for example. One sequence represents the primary flow of data, while another represents arbitrary traffic or noise to ensure the DUT can distinguish the primary data stream correctly. Alternatively, each concurrent sequence can represent a data stream destined for a different physical or virtual channel on the DUT, thereby saturating the DUT with as much data traffic as possible. Or, each sequence represents a device on a bus, each accessing a different portion of the address space. The modeling possibilities are limitless.

III. Virtual Sequences

Virtual sequences differ from standard sequences in one and only one way. A typical sequence is bound to a sequencer when launched; a virtual sequence is not. Otherwise, they are the same thing. Typically, a sequence is bound to a sequencer through the `start()` task, where the `start()` task is used to launch the sequence. E.g.

```
sequence.start(sequencer);
```

The sequence stores the sequencer handle, which establishes the path from the sequence to the driver. Later, when `start_item()` and `finish_item()` transfer an item to the sequencer, those tasks know which sequencer to pass the sequence items to. A virtual sequence, on the other hand, is launched without a sequencer handle.

```
sequence.start(); // Launching a virtual sequence
```

It is the responsibility of the virtual sequence to find a sequencer handle to send a sequence item. UVM is silent on how a virtual sequence can locate sequencer handles.

Other than not having a sequencer bound at launch time, virtual sequences and standard sequences are the same thing. They have the same UVM base class, `uvm_sequence#()`. Sequences and virtual sequences operate the same way within the UVM kernel. Their virtual functions operate the same way, etc. They are identical, with the difference that virtual sequences need to find a sequencer handle rather than having it handed to them.

IV. Virtual Sequence Techniques

This paper describes two virtual sequence techniques in detail: virtual sequencers and sequencer containers. This paper briefly describes two additional virtual sequence techniques that engineers rarely use: a `test_base init_vseq` technique and storing and retrieving sequencer handles in the UVM resource database.

This paper demonstrates virtual sequence techniques using a simple example that starts with three block-level designs utilizing simple sequences that execute different transaction types and progresses to the coordination of the existing sequences on the same blocks, which we assemble into a larger design.

V. Sample Design

To demonstrate the various virtual sequence techniques, we employed a sample design that instantiates three simple block-level designs with complete UVM testbenches, including self-checking scoreboards. Each block is driven by a different agent using a different transaction type. We drive other sequences to each block-level design, and the scoreboards report that the tests pass.

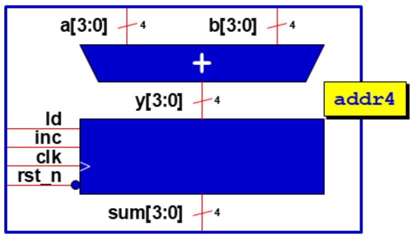
Then we connect the three blocks into a larger design, and use a virtual sequence to coordinate the execution of existing block-level stimulus sequences across the three interfaces of the multi-block design.

To enhance learning, we have intentionally designed each block and its corresponding block-level sequence to be simple. The simplicity of the blocks and sequences allows the reader to follow simple learning steps to understand the overall techniques. Readers can then apply and expand the techniques to more complex blocks and systems.

A. 4-bit Adder (addr4) w/ Increment and Load

The first block, `addr4`, is a simple 4-bit adder with registered output. Three signals control the register with the following priority: (1) asynchronous low-true reset, (2) load the register with the input value, (3) increment the last registered value, or (4) hold the last register value.

Figure 1 shows the block diagram and RTL code for the **addr4** blockFigure 1.

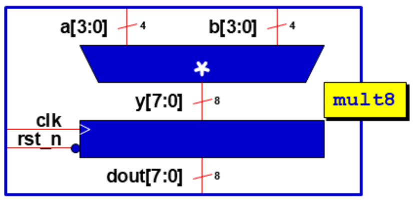


```
module addr4 (
  output logic [3:0] sum,
  input  logic [3:0] a, b,
  input  logic       ld, inc,
  input  logic       clk, rst_n);

  logic  [3:0] y;

  assign y = a + b;

  always_ff @(posedge clk, negedge rst_n)
    if       (!rst_n) sum <= '0;
    else if     (ld)  sum <= y;
    else if    (inc)  sum <= sum + 1;
endmodule
```

Figure 1 - addr4 block diagram and code

B. 8-bit Multiplier (mult8)

The second block, **mult8**, is a simple multiplier that takes two 4-bit inputs and generates an 8-bit result. The 8-bit result is loaded into a **dout** register. The register can also be reset using an asynchronous low-true reset.
Figure 2 shows the block diagram and RTL code for the **mult8** block.

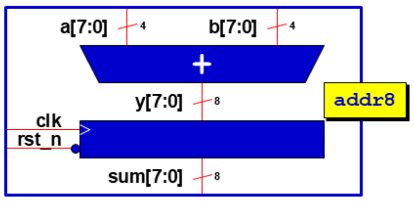


```
module mult8 (
  output logic [7:0] dout,
  input  logic [3:0] a, b,
  input  logic       clk, rst_n);

  logic  [7:0] y;

  assign y = a * b;

  always_ff @(posedge clk, negedge rst_n)
    if       (!rst_n) dout <= '0;
    else              dout <= y;
endmodule
```

Figure 2 - mult8 block diagram and code

C. 8-bit Adder (addr8)

The third block, **addr8**, is a simple adder that takes two 8-bit inputs and generates an 8-bit result. The 8-bit result is loaded into a **sum** register. The register can also be reset using an asynchronous low-true reset.
Figure 3 shows the block diagram and RTL code for the **addr8** block. Figure 3



```
module addr8 (
  output logic [7:0] sum,
  input  logic [7:0] a, b,
  input  logic       clk, rst_n);

  logic  [7:0] y;

  assign y = a + b;

  always_ff @(posedge clk, negedge rst_n)
    if       (!rst_n) sum <= '0;
    else              sum <= y;
endmodule
```

Figure 3 - addr8 block diagram and code

D. DUT Interface Files (a4_dut_if.sv / m8_dut_if.sv / a8_dut_if.sv)

The three previously described blocks are connected to block-level UVM testbenches using the DUT interface files shown in Figure 4.

Each of the DUT interface files uses common file-guard techniques on the first two and last lines of the files. When compiling UVM package files, some compilers require knowledge of existing DUT interface files, both when the package is compiled and again when top-level designs are compiled. The file guards ensure that the DUT interface files can be read multiple times during compilation, but are only compiled the first time the file is read. The file guards remove annoying errors and warnings that are otherwise generated by some EDA tools.

More information about the **`` `ifndef AMA ``** content in two of the DUT interface files is described in section V-H.

```
`ifndef A4_DUT_IF__SV
`define A4_DUT_IF__SV
`include "CYCLE.sv"
`define Tdrive #(0.2*`CYCLE)

interface a4_dut_if (input clk);
  logic [3:0] sum;
  logic [3:0] a, b;
  logic       inc, ld, rst_n;

  clocking drv_cb @(posedge clk);
    default output `Tdrive;
    output a, b, inc, ld;
    output rst_n;
  endclocking

  clocking mon_cb @(posedge clk);
    default input #1step;
    input  sum;
    input  a, b, inc, ld;
    input  rst_n;
  endclocking
endinterface
`endif
```

```
`ifndef M8_DUT_IF__SV
`define M8_DUT_IF__SV
`include "CYCLE.sv"
`define Tdrive #(0.2*`CYCLE)

interface m8_dut_if (input clk);
  logic [7:0] dout;
  logic [3:0] a, b;
  logic       rst_n;

  clocking drv_cb @(posedge clk);
    default output `Tdrive;
    `ifndef AMA
      output a;
    `endif
    output b;
    output rst_n;
  endclocking

  clocking mon_cb @(posedge clk);
    default input #1step;
    input  dout;
    input  a, b;
    input  rst_n;
  endclocking
endinterface
`endif
```

```
`ifndef A8_DUT_IF__SV
`define A8_DUT_IF__SV
`include "CYCLE.sv"
`define Tdrive #(0.2*`CYCLE)

interface a8_dut_if (input clk);
  logic [7:0] sum;
  logic [7:0] a, b;
  logic       rst_n;

  clocking drv_cb @(posedge clk);
    default output `Tdrive;
    `ifndef AMA
      output a;
    `endif
    output b;
    output rst_n;
  endclocking

  clocking mon_cb @(posedge clk);
    default input #1step;
    input  sum;
    input  a, b;
    input  rst_n;
  endclocking
endinterface
`endif
```

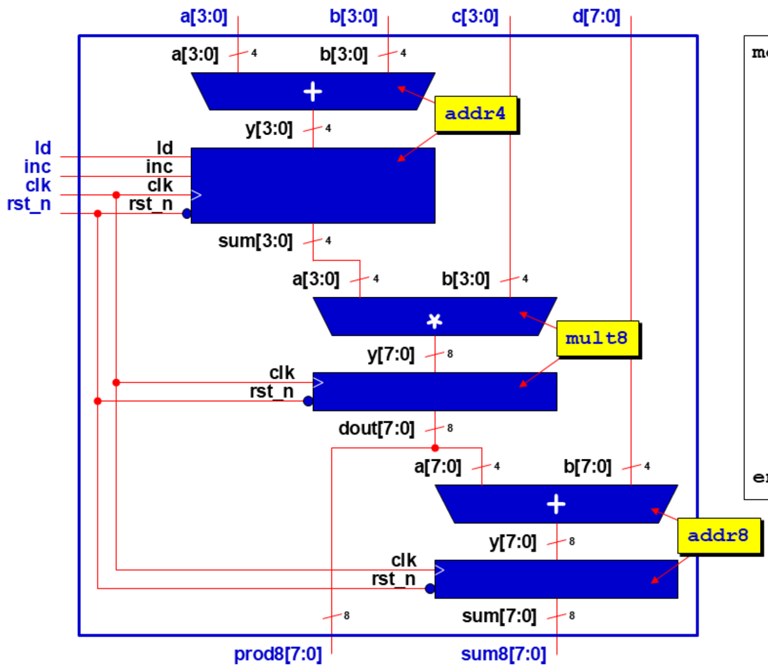
Figure 4 - a4_dut_if.sv / m8_dut_if.sv / a8_dut_if.sv files

These three DUT interface files will also be utilized in the larger **`ama_blk`** UVM testbench, allowing us to use the same block-level scoreboards to ensure that the blocks continue to function correctly. There are a couple of considerations that are addressed in the **`ama_blk`** top-module to resolve minor issues as described in sections V-F, V-G, and V-H.

E. Multi-block Adder-Multiplier-Adder (ama_blk)

We instantiate the three smaller blocks into a larger Adder-Multiplier-Adder (AMA) block, which uses three 4-bit data inputs (**`a`**, **`b`**, **`c`**) and one 8-bit data input (**`d`**). The **`ama_blk`** uses the same control signals as the **`addr4`** block and shares the **`clk`** and **`rst_n`** signals with the other two blocks. The **`ama_blk`** outputs an 8-bit multiplication product (**`prod8`**) from the **`mult8`** block, and an 8-bit sum (**`sum8`**) from the **`addr8`** block.

Figure 5 shows the block diagram and RTL code for the **`ama_blk`** block.



```
module ama_blk (
  output logic [7:0] sum8, prod8,
  input  logic [3:0] a, b, c,
  input  logic [7:0] d,
  input  logic       inc, ld,
  input  logic       clk, rst_n);

  logic [3:0] sum4;

  addr4 i1 (.sum(sum4), .*);

  mult8 i2 (.dout(prod8),
            .a(sum4), .b(c), .*);

  addr8 i3 (.sum(sum8),
            .a(prod8), .b(d), .*);

endmodule
```

Figure 5 - ama_blk block diagram and code

F. *Multi-block Interface Issues*

Reusing block-level testbenches at the system level introduces two issues that we must address to ensure the existing self-checking block-level testbenches continue to function correctly. These two issues are described in the following two sections.

G. *Missing Block-level Interface Connections Issue*

The block-level interfaces used in each block-level UVM testbench connect the block-level ports hierarchically to the internal signals declared in each interface. These connections are necessary to enable the self-checking scoreboards to function correctly.

Once we connect the three blocks from our example together in a top-level testbench, some of the block-level ports are now hidden inside the larger **ama_blk** design. To ensure the block-level scoreboards continue to function correctly, they require access to these previously visible ports.

In this example, the top-level **ama_blk** has been instantiated using the instance name i1. Internal signals from the **i1** instance of the **ama_blk** will be driven onto the missing connections of the three block-level interfaces.

The **a4_dut_if sum** bus (**addr4 sum**-output bus) needs a connection to the **ama_blk** internal **sum4** bus. The a4_dut_if sum bus connection is accomplished in the **top** module using a continuous assignment and hierarchically driving the internal **i1.sum4** bus onto the **a4_dif.sum** bus as shown below.

```
    assign a4_dif.sum  = i1.sum4;
```

The **m8_dut_if a** bus (**mult8 a**-input bus) also needs a connection to the **ama_blk** internal **sum4** bus. The m8_dut_if establishes a bus connection in the **top** module using a continuous assignment and hierarchically drives the internal **i1.sum4** bus onto the **m8_dif.a** bus as shown below.

```
    assign m8_dif.a    = i1.sum4;
```

The **a4_dut_if a** bus (**addr8 a**-input bus) needs a connection to the **ama_blk** internal **prod8** bus. The a4_dut_if bus connection is accomplished in the **top** module using a continuous assignment and hierarchically driving the internal **i1.prod8** bus onto the **a8_dif.a** bus as shown below.

```
    assign a8_dif.a    = i1.prod8;
```

All other block-level interface signals are connected to ports on the upper-level `ama_blk` as shown below.

```
ama_blk  i1 (.sum8(a8_dif.sum), .prod8(m8_dif.dout),
             .a(a4_dif.a), .b(a4_dif.b), .c(m8_dif.b),
             .d(a8_dif.b), .ld(a4_dif.ld), .inc(a4_dif.inc),
             .rst_n(a4_dif.rst_n), .clk(clk), .*);
```

These continuous assignments enable the block-level scoreboards to access all the necessary signals to verify each block-level design, allowing the upper-level UVM testbench to continue utilizing the self-checking block-level scoreboards.

H. *Multiple Drivers Connection Issue*

The second issue when connecting the individual blocks to the upper-level design is that the block-level clocking block in the DUT interface was previously responsible for driving a block-level input, but now that input is connected to the output from another block. This means that there is a driving source from the output of one block to the input of another, and simultaneously, a clocking block drives the same input. An example compiler error message is shown below.

```
Illegal combination of driver and output clockvar to variable 'a'
detected (output clockvar found in clocking block at line ## …
```

All block-level inputs previously driven by a clocking block are now also driven from another block; we turn off the clocking block output declaration in the `drv_cb` as shown below.

```
clocking drv_cb @(posedge clk);
  default output `Tdrive;
  `ifndef AMA
    output a;
  `endif
  output b;
  output rst_n;
endclocking
```

For our example, at the top of the `ama_blk` compilation command file (`top_run.f`), we added `+define+AMA` to indicate that this is a top-level design. The defined `AMA` macro prevents the extra clocking block output declarations from being compiled and included in the simulation.

VI. VSEQUENCER TECHNIQUE

A technique commonly used to locate sequencer handles is a "virtual sequencer" (described in this section as the `vsequencer` technique), a sequencer that serves as a container for other sequencer handles. Why use a `vsequencer` as a container? If you recall, sequencers are part of the UVM component hierarchy, whereas sequences are not. However, a sequence can attach to a virtual sequencer (which is not really virtual) to locate sequencer handles. Although commonly used, engineers should replace the `vsequencer` technique with one of the more efficient sequencer container techniques described in section VII.

Since all sequences must be started on a sequencer, this technique includes a `vsequencer` class extended from `uvm_sequencer`. The `vsequencer` class is just a container that holds handles to the subsequencers. It is the job of the environment to use hierarchical references to copy the subsequencer handles from the agents to the handles declared in the `vsequencer` wrapper class.

Figure 6 shows the `vsequencer` class used with the `AMA_BLK` example. The `vsequencer` is nothing more than a sequencer container that declares and holds handles to the three subblock sequencers. It is the job of the `top_env` to copy the hierarchical sequencer handles from the subblocks to these handles declared in the `vsequencer`.

```
class vsequencer extends uvm_sequencer;
  `uvm_component_utils(vsequencer)

  a4_sequencer a4_sqr;
  m8_sequencer m8_sqr;
  a8_sequencer a8_sqr;

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

Figure 6 - vsequencer class for AMA_BLK example

Since a sequence cannot directly access the subsequencer handles from a UVM environment, sequences are started on a sequencer, and they must retrieve the subsequencer handles stored in a sequencer, or in this case, a virtual sequencer (**`vsequencer`**) container class. Using a **`vsequencer`** typically requires the use of the **`` `uvm_declare_p_sequencer ``** macro, **`p_sequencer`**, and **`m_sequencer`** handles. For a better understanding of **`m_sequencer`**, **`p_sequencer`**, and the **`` `uvm_declare_p_sequencer ``** macro, refer to (Cummings, 2024)[5]. Verification engineers could eliminate these **`m_sequencer`**, **`p_sequencer`**, and macro gymnastics if the **`vseq_base`** class could directly retrieve the subsequencer handles stored in the sequencer container. Figure 7 illustrates the top virtual sequencer base class used with the **`AMA_BLK`** example.

Note that many users of the **`vsequencer`** technique put the **`p_sequencer`** assignments into the **`body()`** task of the virtual sequence base class. Although coding the assignments in the **`body()`** task and calling **`super.body()`** from the extended virtual sequence works, this is an abuse of the **`body()`** task, which was designed to hold the default sequence activity. It is better to put the **`p_sequencer`** assignments into a **`virtual void function`** (**`set_sqr_handles()`** in this example) and then call the **`super.set_sqr_handles()`** method from the extended virtual sequence, which is what is shown in Figure 7 andFigure 8.

```
class top_vseq_base extends uvm_sequence;
  `uvm_object_utils(top_vseq_base)

  `uvm_declare_p_sequencer(vsequencer);

  a4_sequencer A4;
  m8_sequencer M8;
  a8_sequencer A8;

  function new (string name = "top_vseq_base");
    super.new(name);
  endfunction

  virtual function void set_sqr_handles();
    A4 = p_sequencer.a4_sqr;
    M8 = p_sequencer.m8_sqr;
    A8 = p_sequencer.a8_sqr;
  endfunction
endclass
```

Figure 7 - top_vseq_base class for AMA_BLK example

The **`top_vseq_base`** class did the hard work of declaring and retrieving the subsequencer handles. Now, all virtual sequences can extend the **`top_vseq_base`** class, and the **`body()`** of the virtual sequence can call the inherited **`set_sqr_handles()`** method to set the required sequencer handles that the virtual sequence will reference. Figure 8 shows that the **`top_vseq1`** virtual sequence coordinates sequence activity across the multiple DUT interfaces. Figure 8

```
class top_vseq1 extends top_vseq_base;
  `uvm_object_utils(top_vseq1)

  function new(string name="top_vseq1");
    super.new(name);
  endfunction

  task body();
    a4_sequence a4 = a4_sequence::type_id::create("a4");
    m8_sequence m8 = m8_sequence::type_id::create("m8");
    a8_sequence a8 = a8_sequence::type_id::create("a8");

    set_sqr_handles();

    fork
      a4.start(A4);
      m8.start(M8);
      a8.start(A8);
    join
  endtask
endclass
```

Figure 8 - top_vseq1 class for AMA_BLK example

As mentioned earlier, it is the job of the top-level environment to copy the subsequencer handles to the sequencer handles declared in the **`vsequencer`**. Figure 9 shows that in the **`AMA_BLK`** example, there are two layers of environments, and the **`vsequencer`** is built and connected in the **`top_env`**. The **`top_env`** will copy the hierarchical paths of each subsequencer located in each lower-level environment to the handles declared in the **`vsequencer`** located in the upper-level environment.

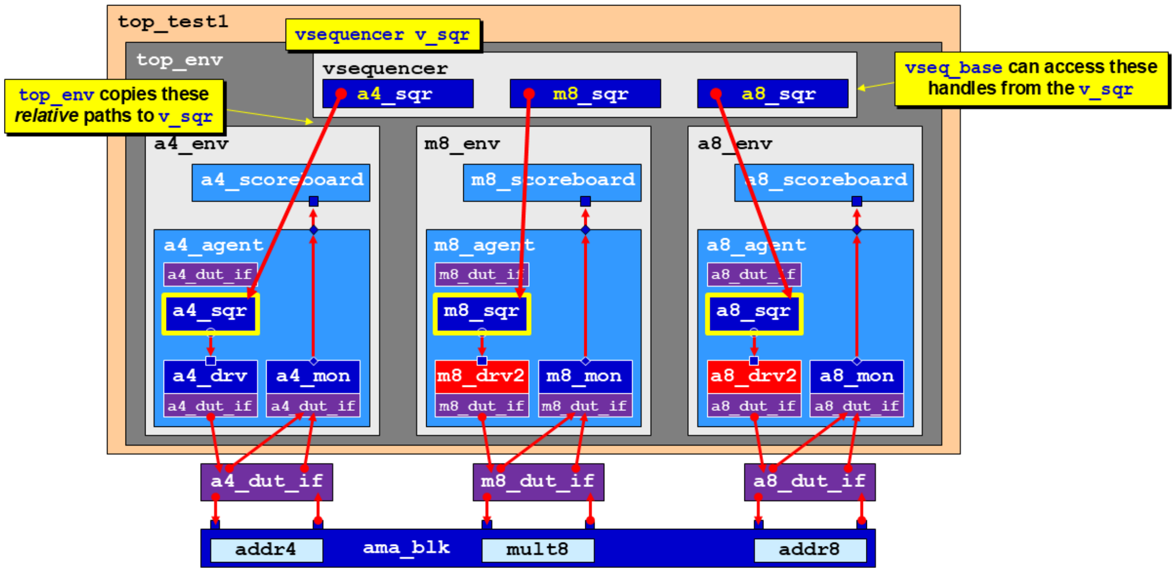


Figure 9 - Top environment block diagram using virtual sequencer component

Figure 10 shows the top-level environment class, `top_env`, that we use with the `AMA_BLK` example.

```
class top_env extends uvm_env;
  `uvm_component_utils(top_env)

  a4_env       env_a4;
  m8_env       env_m8;
  a8_env       env_a8;
  vsequencer   v_sqr;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_a4 =     a4_env::type_id::create("env_a4", this);
    env_m8 =     m8_env::type_id::create("env_m8", this);
    env_a8 =     a8_env::type_id::create("env_a8", this);
    v_sqr  = vsequencer::type_id::create("v_sqr",  this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    v_sqr.a4_sqr = env_a4.agnt.sqr;
    v_sqr.m8_sqr = env_m8.agnt.sqr;
    v_sqr.a8_sqr = env_a8.agnt.sqr;
  endfunction
endclass
```

Figure 10 - top_env class for AMA_BLK example

The `top_test1` class shown in Figure 11 starts the `top_vseq1` "virtual" sequence (from Figure 8) on the `vsequencer` located in the `top_env` (`e.vsqr`).

```
class top_test1 extends top_test_base;
  `uvm_component_utils(top_test1)

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

  // sqrs inherited from top_test_base
  task run_phase(uvm_phase phase);
    top_vseq1 seq;
    seq = top_vseq1::type_id::create("seq");
    //---------------------------------------
    phase.raise_objection(this);
    `uvm_info("top_test1", "about to do seq.start", UVM_FULL)
    seq.start(e.v_sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Figure 11 - top_test1 class for AMA_BLK example

In reality, what `vsequencer` technique typically refers to as a "virtual" sequence does not match the strict definition of a virtual sequence described in section II. This "virtual" sequence is really a regular sequence because it is not started on `null`. The `seq.start(e.v_sqr)` command initiates a regular sequence on the `vsequencer` for the sole purpose of retrieving the subsequencer handles stored in the **`vsequencer`**, which acts as a sequencer container. The **`top_vseq1`** then coordinates the activity of other sequences on their respective subsequencer handles, which represents the real virtual sequence activity. The **`seq.start(e.v_sqr)`** command did not actually run on the **`vsequencer`**; it simply retrieved subsequencer handles that the actual virtual sequence (**`top_vseq1`**) then used to execute the sequences on named sequencer handles.

The **`vsequencer`** technique is a holdover from OVM, where the only way to access configuration items was through components. UVM has the resource database instead of the component-based **`set_config`**/**`get_config`** interface for locating configuration information. The resource database, which is independent of components and the component hierarchy, makes virtual sequencers obsolete. The **`vsequencer`** technique is commonly used in UVM testbenches; however, verification engineers can replace it with a superior sequencer container technique, as described in the following two sections.

VII. Sequencer-Handle Container Techniques

Sequencer-handle containers store sequencer handles in globally accessible tables, accessed by **`string`**-names. These containers are hierarchically independent, meaning that the location of the stored handles does not matter because the stored handles are automatically updated with their current component paths when an environment calls the built-in agent method, **`get_sequencer()`**. The environment calls the **`get_sequencer()`** method to store the handle at a globally available **`string`**-name location. The virtual sequence (or virtual sequence base) retrieves the stored handles from the global **`string`**-name locations using a **`set_sequencers()`** method to set the sub-sequencer handles required by the virtual sequence base (**`vseq_base`**) class.

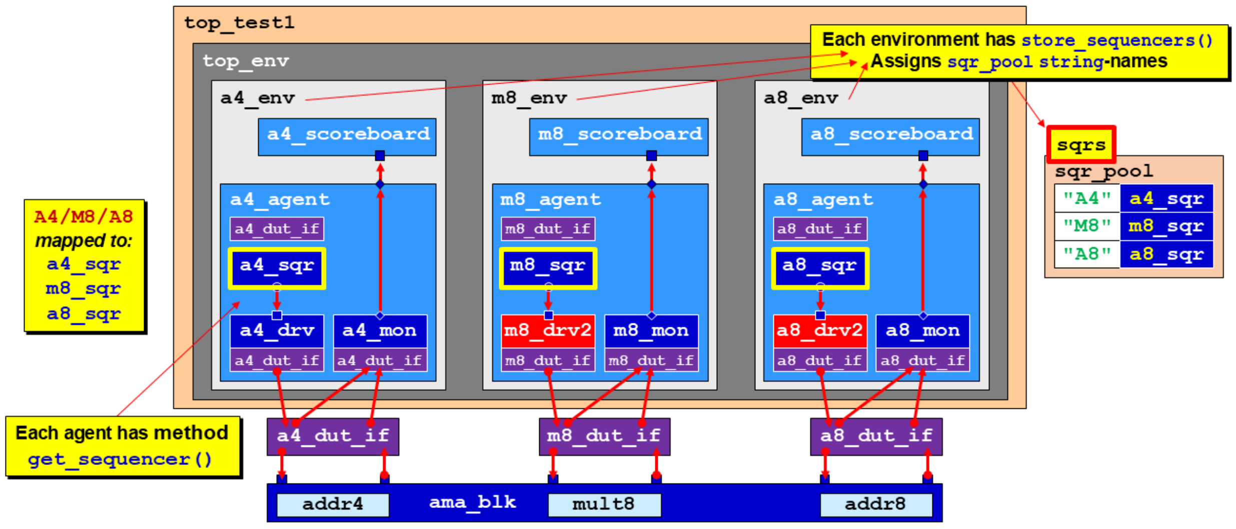


Figure 12 - Top environment block diagram using the sqr_pool container

VIII. Sequencer Aggregators

A sequencer aggregator is a class object that contains a set of sequencer handles. The sequencer aggregator has APIs (Application Programming Interfaces) for inserting sequencer handles into the container and for locating and retrieving sequencer handles. Virtual sequences locate sequencer handles by first finding a sequencer aggregator and then looking up the sequencer handle in the sequencer aggregator. Environments populate sequencer aggregator(s) during the **`build_phase()`** and **`connect_phase()`**.

There are two kinds of sequencer aggregators. One kind is a singleton, also known as a sequencer pool (**`sqr_pool`**). The other is a dynamic class object stored in the resource database, known as a sequencer aggregator (**`sqr_aggregator`**).

The paper "Sequencer Containers - A Unified and Simple Technique to Execute Both Sequences and Virtual Sequences" by Cliff Cummings and Mark Glasser [6] details these techniques.

A. Sequencer Pool (sqr_pool)

For the singleton kind, sequences only need to get a handle to the singleton object, which in our **AMA_BLK** example is named **sqrs**.

```
typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;
sqr_pool_type sqrs = sqr_pool_type::get_global_pool();
```

Then, sequencer handles can be looked up by **string**-name or by any lookup provided by the data structure. In our **sqr_pool** example, the only lookup method provided is a **string**-name lookup **get()** method. Figure 12 shows the **sqr_pool** class used with our examples. Figure 13

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

  typedef sqr_pool #(T) this_type;
  static protected this_type m_global_pool;
  // protected T pool[KEY];    // Inherited - This is the sqr_pool

  protected function new (string name="");
    super.new(name);
  endfunction

  static function this_type get_global_pool ();
    if (m_global_pool==null)
      m_global_pool = new("pool");
    return m_global_pool;
  endfunction

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

  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

  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
endclass
```

Figure 13 - sqr_pool class code

1) `get_sequencer()`: Each agent should implement the `get_sequencer()` method to retrieve the component path to the agent's sequencer, no matter where the agent and its sequencer are located in the overall UVM testbench. The agent does not need to know that a `sqr_pool` even exists. Engineers and Verification IP (VIP) developers can safely add the get_sequencer() method to any agent, which may or may not ever be used (engineers using the `sqr_pool` will use the `get_sequencer()` method). To allow engineers to use the advanced sequencer container techniques described in this paper, every verification engineer or developer of commercially sold VIP agents should add the get_sequencer() method to all agent code. The same `get_sequencer()` method will be called by an environment, whether using either of the `sqr_pool` or `sqr_aggregator` sequencer container techniques.

2) `store_sequencers()`: Each environment includes one or more agents, and each environment should implement its own version of the `store_sequencers()` method, which calls the `get_sequencer()` method from each agent, which the environment code will build. Figure 14 shows the code for one of the three block-level environments (`a4_env`) used in both the block-level and multi-block design examples.

```
class a4_env extends uvm_env;
  `uvm_component_utils(a4_env)

  typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;
  sqr_pool_type sqrs = sqr_pool_type::get_global_pool();

  a4_agent      agnt;
  a4_scoreboard sbd;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agnt =      a4_agent::type_id::create("agnt", this);
    sbd  = a4_scoreboard::type_id::create("sbd",  this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    agnt.ap.connect(sbd.axp);
    store_sequencers();
  endfunction

  //-----------------------------------------------------------
  // Each environment that instantiates one or more agents
  // contains a function store_sequencers(), which adds the
  // sequencers from the agents into the sequencer pool
  //-----------------------------------------------------------
  virtual function void store_sequencers();
    sqrs.add("A4", agnt.get_sequencer());
  endfunction
endclass
```

Figure 14 - a4_env class for AMA_BLK example

The environment builds the agent in the `connect_phase()`, and calls the `store_sequencers()` method to store the `string`-named agent handle using the `add()` method from the `sqr_pool`. The lower-level environment requires access to retrieve the `sqrs sqr_pool` singleton handle at the top of the file. When using the `sqr_pool` sequencer container, the `string`-names used across all the environments must be unique.

An upper-level environment (one that builds the lower-level environments) does not implement the `store_sequencers()` method and does not need to know about the `sqr_pool`. Figure 15 shows the `top_env` class that we used with the `AMA_BLK` example. The `top_env` declares and builds the lower-level environments. Lower-level environments store the sequencer handles by calling the `store_sequencers()` method in the `connect_phases()` method of the lower-level environment.

```
class top_env extends uvm_env;
  `uvm_component_utils(top_env)

  a4_env       env_a4;
  m8_env       env_m8;
  a8_env       env_a8;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_a4 = a4_env::type_id::create("env_a4", this);
    env_m8 = m8_env::type_id::create("env_m8", this);
    env_a8 = a8_env::type_id::create("env_a8", this);
  endfunction
endclass
```

Figure 15 - top_env class for AMA_BLK example

3) ***`dump()`****:* To simplify debugging the sequencer paths used in a UVM testbench, the `sqr_pool` includes a `dump()` method to concisely print out each stored sequencer name and its corresponding path in the current UVM testbench. In the `top_test_base` example shown in Figure 16, we `use the dump()` function to print the `sqr_pool` names and paths in both the `start_of_simulation_phase()` and the `final_phase()`.

```
class top_test_base extends uvm_test;
  `uvm_component_utils(top_test_base)

  typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;

  // Get the factory and sqr_pool handles
  uvm_factory   factory = uvm_factory::get();
  sqr_pool_type sqrs    = sqr_pool_type::get_global_pool();

  top_env e;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    e = top_env::type_id::create("e", this);
  endfunction

  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
    `uvm_info("TB", "top_test_base: start_of_simulation", UVM_FULL)
  endfunction
  function void final_phase(uvm_phase phase);
    if (uvm_report_enabled(UVM_HIGH)) sqrs.dump();
  endfunction
endclass
```

Figure 16 - top_test_base class for AMA_BLK example

Note that dumping only occurs if an engineer uses a UVM verbosity of **`UVM_HIGH`** or higher. We **`dump()`** the sequencer handles in the **`start_of_simulation_phase()`** to easily view and examine the handles if the simulation aborts during the **`run_phase()`**. We also **`dump()`** the sequencer handles in the **`final_phase()`** so that the handles are visible at the bottom of the simulation report for easy examination. Figure 17 shows the test-passed messages from the **`AMA_BLK`** simulation, followed by the **`dump()`** of the Sequencer Pool Entries.

```
*** ADDR4 TEST PASSED - Vectors: 102 Ran / 102 Passed ***
*** ADDR8 TEST PASSED - Vectors: 102 Ran / 102 Passed ***
*** MULT8 TEST TEST PASSED - Vectors: 102 Ran / 102 Passed ***

--- SEQUENCER POOL ENTRIES -------
        A4 : uvm_test_top.e.env_a4.agnt.sqr
        A8 : uvm_test_top.e.env_a8.agnt.sqr
        M8 : uvm_test_top.e.env_m8.agnt.sqr
--- END SEQUENCER POOL     -------
```

Figure 17 - Test Passed and sqr_pool dump() output for the AMA_BLK example

Summarizing the methods used by the structural, component-portion of a UVM testbench, each agent should include the three simple lines of code that make up the **`get_sequencer()`** method, and the agent does not need to know that a **`sqr_pool`** or **`sqr_aggregator`** even exists. The lower-level environments and the **`test_base`** both require access to the **`sqr_pool`** singleton handle, so they must understand that a **`sqr_pool`** exists. The lower-level environment is responsible for retrieving the agent-sequencer handles (by calling **`get_sequencer()`** for each agent in the environment) and implementing a **`store_sequencers()`** method to store uniquely **`string`**-named handles in the **`sqr_pool`** singleton. The **`top_test_base`** calls the **`dump()`** method to display the sequencer paths that exist for the current UVM testbench.

Now for highlights regarding the necessary virtual sequence-execution code.

*4) **`set_sqr_handles()`**:* Once the testbench has stored the sequencer handles in the **`sqr_pool`**, virtual sequences need to retrieve those handles to coordinate and drive the execution of the different sequences to the interface of a block-level design, or the various interfaces of the multi-block design. The recommended way to implement the different virtual sequences is to extend a **`vseq_base`** class that does the hard work of retrieving all the required **`sqr_pool`** handles, assigning them to handles declared as a **`uvm_sequencer_base`**-type, and implementing a **`set_sqr_handles()`** method. Virtual sequences will call the **`set_sqr_handles()`** method inherited from the **`vseq_base`** class. Figure 18 shows the **`top_vseq_base`** class used in the **`AMA_BLK`** simulation.

```
class top_vseq_base extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(top_vseq_base)

  typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;
  sqr_pool_type sqrs = sqr_pool_type::get_global_pool();

  uvm_sequencer_base A4;
  uvm_sequencer_base M8;
  uvm_sequencer_base A8;

  function new (string name = "top_vseq_base");
    super.new(name);
  endfunction

  virtual function void set_sqr_handles();
    if (A4 == null) begin
```

```
        A4 = sqrs.get("A4");
        M8 = sqrs.get("M8");
        A8 = sqrs.get("A8");
      end
    endfunction
  endclass
```
Figure 18 - top_vseq_base class used by the AMA_BLK example

The **top_vseq_base** class requires access to the **sqr_pool** singleton, declares **A4**, **M8**, and **A8** handles of the **uvm_sequencer_base** type, and implements the **set_sqr_handles()** method to retrieve the **string**-named sequencer handles and assign them to the **A4**, **M8**, and **A8** handles. Since all of the sequencers in any UVM testbench are derivatives of the **uvm_sequencer_base** type, any sequencer handle can be upcast to the **uvm_sequencer_base** declared handle names.

The **set_sqr_handles()** method performs a quick **if**-test to check if any of the handles are **null** and only assigns the handles if another virtual sequence has not previously assigned them. There is no need to retrieve the handles again if another virtual sequence has already set them.

Figure 19 shows an example virtual sequence (**top_vseq1**) used in the **AMA_BLK** UVM testbench. The **top_vseq1** extends the **top_vseq_base**, thereby inheriting the **A4**, **M8**, and **A8** sequencer handles, along with the **set_sqr_handles()** method. In the **body()** of the virtual sequence, each of the block-level sequences is declared and created. The virtual sequence calls the **set_sqr_handles()** method, which sets the inherited sequencer handles, and then some coordinated execution of the block-level sequences takes place. In this example, the block-level sequences are executed in parallel using a **fork-join** statement.

```
class top_vseq1 extends top_vseq_base;
  `uvm_object_utils(top_vseq1)

  function new(string name="top_vseq1");
    super.new(name);
  endfunction

  task body();
    a4_sequence a4 = a4_sequence::type_id::create("a4");
    m8_sequence m8 = m8_sequence::type_id::create("m8");
    a8_sequence a8 = a8_sequence::type_id::create("a8");

    set_sqr_handles();

    fork
      a4.start(A4);
      m8.start(M8);
      a8.start(A8);
    join
  endtask
endclass
```
Figure 19 - top_vseq1 class (example virtual sequence) used by the AMA_BLK example

B. Sequencer Aggregator (sqr_aggregator)

As previously mentioned, the **sqr_aggregator** allows for the creation of multiple sequencer containers as opposed to the **sqr_pool** singleton. If your current UVM testbench uses multiple **vsequencers**, you should replace those **vsequencers** with multiple **sqr_aggregators**. Even the single **sqr_pool** example described in section VIII.A could be replaced with a single **sqr_aggregator**.

The dynamic class object style requires a lookup in the resource database to get a handle to the sequencer aggregator.

```
if (!uvm_resource_db#(sqr_aggr_t)::read_by_name(get_full_name(),
                "sqr_aggr", sqr_aggr))
  `uvm_fatal(...)
```

For a more comprehensive description of the design and use of the **sqr_aggregator**, refer to the paper by Cummings and Glasser (2025)[6] and Mark Glasser's book on Next Level Testbenches [7].

C. *Comparing sqr_pool to sqr_aggregator*

For comparison purposes, Table 1 shows the significant differences between the **sqr_pool** and **sqr_aggregator** sequencer containers. Choosing which of the **sqr_pool** and **sqr_aggregator** sequencer containers to use is a matter of debate.

If a UVM testbench currently uses just one older-style **vsequencer**, it only uses one **sqr_pool**. If a simple **string**-name retrieval method is sufficient, then the **sqr_pool** provides the required sequencer container functionality.

If a UVM testbench currently uses multiple older-style **vsequencers**, then a unique **sqr_aggregator** should be used to replace each of the different **vsequencers** in the UVM testbench.

See Table 1 for more capabilities and information about using the **sqr_pool** and **sqr_aggregator**.

Table 1 - sqr_pool features versus sqr_aggregator features

Good for simple UVM testbenches	**Better for advanced UVM testbenches**
sqr_pool **Features**	**sqr_aggregator** **Features**
Singleton – only one **sqr_pool** possible	***NOT*** a singleton - one or more **sqr_aggregators** possible
add(name) method Adds sequencer handle to just *one* table: • **string**-indexed **name_table[name]**	**add(sqr, name, kind)** method Simultaneously adds **sqr** handle to *three* different tables: • **string**-indexed **name_table[name]** • **string**-indexed **sqr_table[path]** • **string**-indexed **kind_table[kind]**
get() method Only *one* **get()** handle-retrieval method to retrieve a single **sqr** handle: • **get[string name]** *handle from* **name_table[name]**	**lookup()** methods *Four* different **lookup** methods to retrieve **sqr** handle *or queues* of **sqr** handles (**sqr_q_t**) from *three* different tables: • **lookup_name[string name]** *handle from* **name_table[name]** • **lookup_path[string path]** *handle from* **name_table[path]** • **lookup_kind[string kind]** *queue from* **kind_table[kind]** • **lookup_path_regex[string regex]** *queue from* **sqr_table[path]**

IX. INIT_VSEQ TECHNIQUE

This section provides a brief introduction and summary of how the **init_vseq** technique works, as well as its numerous disadvantages.

Engineers do not commonly use the **init_vseq** technique. Engineers can safely skip this section in favor of using one of the preferred sequencer-container virtual sequence techniques. If an engineer discovers that the **init_vseq** technique is used in an existing UVM testbench, they can refer back to this section for information about this technique.

The **init_vseq** technique is described by the Siemens Verification Methodology Team [8] but suffers from three significant flaws:

(1) The **init_vseq** technique uses fixed paths to sequencers, which makes the testbench environment inflexible and subject to difficult debugging if the environment topology changes.

(2) The **init_vseq** technique utilizes a **test_base** class to establish fixed paths to sequencers within an environment. Aside from including **config** object configuration settings, the environment should largely shield a test from the structure of the environment. UVM developers never intended the test to be a sequencer handle container. Multiple tests and multiple **test_base** classes can use the same environment.

(3) The `init_vseq` technique typically calls the `init_vseq()` method before calling each unique virtual sequence. It is not unusual for the test to make multiple calls to the `init_vseq()` method preceding each call to a virtual sequence.

We discourage the use of the `init_vseq` technique.

X. UVM Resource Database Storage and Retrieval of Sequencer Handles

Engineers do not commonly use the storage and retrieval of sequencer handles using the UVM resource database technique. Engineers can safely skip this section in favor of using one of the preferred sequencer-container virtual sequence techniques.

Cliff and Mark's DVCon 2023 paper [9] describes a technique for storing and retrieving sequencer handles from the UVM resource database, offering a more straightforward approach than the `vsequencer` technique. Making multiple UVM resource database accesses to sequencer handles in the UVM Resource Pool is an expensive simulation operation, which is why we encourage engineers to use one of the preferred sequencer-container virtual sequence techniques described in Section VIII over this UVM resource database technique.

XI. Summary and Conclusions

Table 2 shows a summary of the capabilities of the older-style Virtual Sequencers (`vsequencers`) to the recommended Sequencer Container (`sqr_pool` and `sqr_aggregator`) techniques. The most compelling reason to adopt sequencer containers is that the sequencer handles are continuously updated automatically as the UVM testbench topology changes; there is no need to update sequencer and sequence paths as the UVM testbench evolves.

Table 2 - Comparison of Virtual Sequencer to Sequencer Pool Techniques

	`vsequencer`	`sqr_pool`	`sqr_aggregator`
Only one storage container - Singleton	×	✓	×
Multiple storage containers possible - Not singleton	✓	×	✓
Agents include `get_sequencers()` method	×	✓ Great!	✓ Great!
Handles ***automatically*** update as testbench changes	×	✓ Great!	✓ Great!
Handles stored by low-level environment(s)	×	✓	✓
Handles stored by top-level environment(s)	✓	×	×
Handles stored/retrieved - `vsequencer` component	BAD ✓	×	×
Handles stored/retrieved - name-table associative array	×	✓	✓
Handles stored/retrieved - kind- & path-table assoc. arrays	×	×	✓
"Virtual" sequences `start()` on `vsequencer` comp.	BAD ✓	×	×
Virtual sequences started on `null`	×	✓	✓
"Virtual" seq. `start()` might need manual update	BAD ✓	×	×
"Virtual" seq. uses `` `uvm_declare_p_sequencer ``	✓	×	×
"Virtual" seq. uses `p_sequencer`	✓	×	×
Virtual seq. ***automatically*** retrieves updated sqr handles	×	✓ Great!	✓ Great!

Sequences are the UVM building blocks for creating stimulus. Virtual sequences introduce tremendous flexibility for constructing very complex, abstracted stimuli. The UVM base class library does not address all of the issues associated with using sequences. In this paper, we have gone beyond the introductory sequence/sequencer arrangement to discuss how to use virtual sequences effectively. The paper discusses tools such as sequencer pools and sequencer aggregators to illustrate how to create complex stimulus structures.

We recommend replacing all existing techniques that execute a virtual sequence with either the `sqr_pool` or `sqr_aggregator` described in Section VIII of this paper.

References

[1] Meade, K. A., & Rosenberg, S. (2013). Practical guide to adopting the Universal Verification Methodology (UVM). Cadence Design Systems.

[2] Salemi, R. (2013). The UVM Primer: An introduction to the universal verification methodology. Boston Light Press.

[3] Cummings, C. E., Bergeron, J. (2019, September). *Using UVM Virtual Sequencers & Virtual Sequences.* Sunburst Design Papers. www.sunburst-design.com/papers/CummingsDVCon2016_Vsequencers.pdf

[4] "IEEE Standard for Universal Verification Methodology Language Reference Manual," in *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, vol., no., pp.1-458, 14 Sept. 2020, doi: 10.1109/IEEESTD.2020.9195920.

[5] Cummings, C.E. (2024, April). *Understanding the UVM m_sequencer, p_sequencer Handles, and the `uvm_declare_p_sequencer Macro.* Sunburst Design Papers. www.sunburst-design.com/papers/CummingsSNUG2024SV_m_sequencer_p_sequencer.pdf

[6] Cummings, C. E., & Glasser, M. (2025, March). *A Unified and Simple Technique to Execute Both Sequences and Virtual Sequences.* Sunburst Design Papers. www.sunburst-design.com/papers/CummingsDVCon2025_SqrPool.pdf

[7] Glasser, M. (2024). Next Level Testbenches: Design Patterns in SystemVerilog and UVM. Mark Glasser.

[8] Siemens Verification Methodology Team. (2014, March). *Virtual Sequences*. SIEMENS Verification Academy. https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/virtual-sequences/

[9] Cummings, C. E., Chambers, H., & Glasser, M. (2023, March). *The Untapped Power of UVM Resources and Why Engineers Should Use the uvm_resource_db.* Sunburst Design Papers. www.sunburst-design.com/papers/CummingsDVCon2023_uvm_resource_db_API.pdf

Appendix – Sequences versus Virtual Sequences – Historical Notes

Note from Cliff Cummings: One of the authors of this paper, Mark Glasser, is the co-inventor of Mentor's AVM (Advanced Verification Methodology), co-inventor of OVM (Open Verification Methodology), co-inventor of UVM (Universal Verification Methodology), the inventor of UVM Resources and the **uvm_resource_db** API (and co-inventor of the secondary and inferior **uvm_config_db** API [9]), and the inventor of the sequencer aggregator (**sqr_aggregator**) described in this paper.

While creating this paper, Cliff, Smita and Mark had many UVM technology-related conversations and exchanges. Those exchanges exposed common UVM misunderstandings about sequences, sequencers, virtual sequences and virtual sequencers. For informational and historical purposes, one of those exchanges is included below. We believe this will be beneficial to engineers who want a deeper understanding of the design, intent, and functionality of UVM.

CLIFF: So, I need one more clarification. Perhaps I have been teaching sequence-coding wrong for years. I always thought that sequences should be parameterized to the item-type it was processing, but perhaps that was never a requirement (??) A quick look at the **uvm_sequence_base** class seems to indicate that it might not be necessary and is only considered when calling the built-in **print()** method. Is that correct?

MARK: Short answer: parameters are not required for sequences. Let's look at the classic configuration:

sequence —> **sequencer** —> **driver**

The sequence does indeed take parameters, but they have default types, and thus the parameters are optional. All the heavy lifting is done in the **uvm_sequence_base** and **uvm_sequencer_base** base classes. The derived class **uvm_sequence#()** uses the parameters (i.e., the types defined by the parameters) to send requests and retrieve responses. If you are not sending requests or retrieving responses then you don't have to worry about parameter types, you can just use the defaults. If you are sending requests and retrieving responses in a sequence then the sequence must be defined with the **REQ** and **RSP** data types. You can omit the **RSP** parameter if you are only sending requests and not retrieving responses, which is what people do most of the time.

The sequencer connection to the driver must have a type. The connection uses a **seq_item_port** which is part of the uvm_tlm package.

```
uvm_seq_item_pull_imp #(REQ, RSP, this_type) seq_item_export;
```

uvm_seq_item_pull_imp must be defined with a type, which it gets from the sequencer class parameters.

CLIFF: I have always taught that virtual sequences coordinate the execution of other sequences that could have different item-types.

MARK: That's a very common thing they can do, but not the only thing.

CLIFF: I know there is no such thing as a true virtual sequencer, but the **vsequencer** technique has always relied on a sequence (virtual sequence) that is started on the **vsequencer** (extended from **uvm_sequencer**) in order to get a handle to a container (the **vsequencer**) that had stored the subsequencer handles. This is very common in industry, and it was certainly common at <*previous workplace where Cliff and Mark worked*>, and our paper is attempting to get the UVM verification industry to abandon the practice in favor of sequencer containers such as the **sqr_pool** and **sqr_aggregator**.

MARK: This is where we may have a difference in terminology. I'm not sure what "the `vsequencer` technique" is. Do you mean a sequencer that contains sequencer handles, commonly referred to as a virtual sequencer?

CLIFF: The way we word the paper, this type of "virtual sequence" (a higher-level sequence to coordinate sequences that run on different DUT interfaces) is simply a regular sequence, as it is started on a `vsequencer` to obtain the handles for stored subsequencer handles. If I am now correct, I should do some rewording of the paper to make it clear that this is not a virtual sequence, per your definition. And I should make it clear that the only definition of a virtual sequence is as you have described in the paper. Is that correct?

MARK: The idea is that **the notion of virtual sequences is independent of the notion of virtual sequencers**. As the two terms are almost identical except for an extra 'r', it's easy to conflate the two. As I mentioned in my previous email, virtual sequences were invented long before virtual sequencers.

CLIFF: It does make it confusing that the sequences that are started on a `vsequencer` are in fact NOT virtual sequences (because they are not started on a `null` handle).

MARK: Yes, I can see that this is confusing, but in fact this is the case. This is one of the (many) reasons to abolish virtual sequencers. In fact, the terms sequence and sequencer are a bit confusing. I would have preferred "behavior" and "arbiter" or something like that. I don't remember all the terms that were proposed, but somehow sequence and sequencer made sense at the time. Virtual sequences are not virtual in any way. They are tangible objects. Unlike virtual functions which are sort of like proxies to other functions. I think that someone thought that "virtual sequencers" was a clever turn of phrase and no one stopped to think through the implications.

The bottom line is that virtual sequences can be thought of as objects that contain arbitrary behaviors. Unlike components they can come and go during the course of a simulation. This gives them a lot of flexibility to do many things, not only control other sequences.

CLIFF: I just want to be accurate per your definition. You continue to update my flawed practices!