Using UVM Virtual Sequencers & Virtual Sequences

Clifford E. Cummings
Sunburst Design, Inc.
1639 E 1320 S
Provo, UT 84606
cliffc@sunburst-design.com

Janick Bergeron
Synopsys Inc
2025 NW Cornelius Pass Road
Hillsboro, OR
Janick.Bergeron@synopsys.com

Introduction: What are virtual sequencers and virtuals sequences and when should they be used?

Tests that require coordinated generation of stimulus using multiple driving agents need to use virtual sequences.

This paper will clarify important concepts and usage techniques related to virtual sequencers and virtual sequences that are not well documented in existing UVM reference materials. This paper will also detail the `m_sequencer` and `p_sequencer` handles and the macros and methods that are used with these handles. The objective of this paper is to simplify the understanding of virtual sequencers, virtual sequences and how they work.

When do you need a virtual sequencer?

Figure 1 shows when virtual sequencers are required.

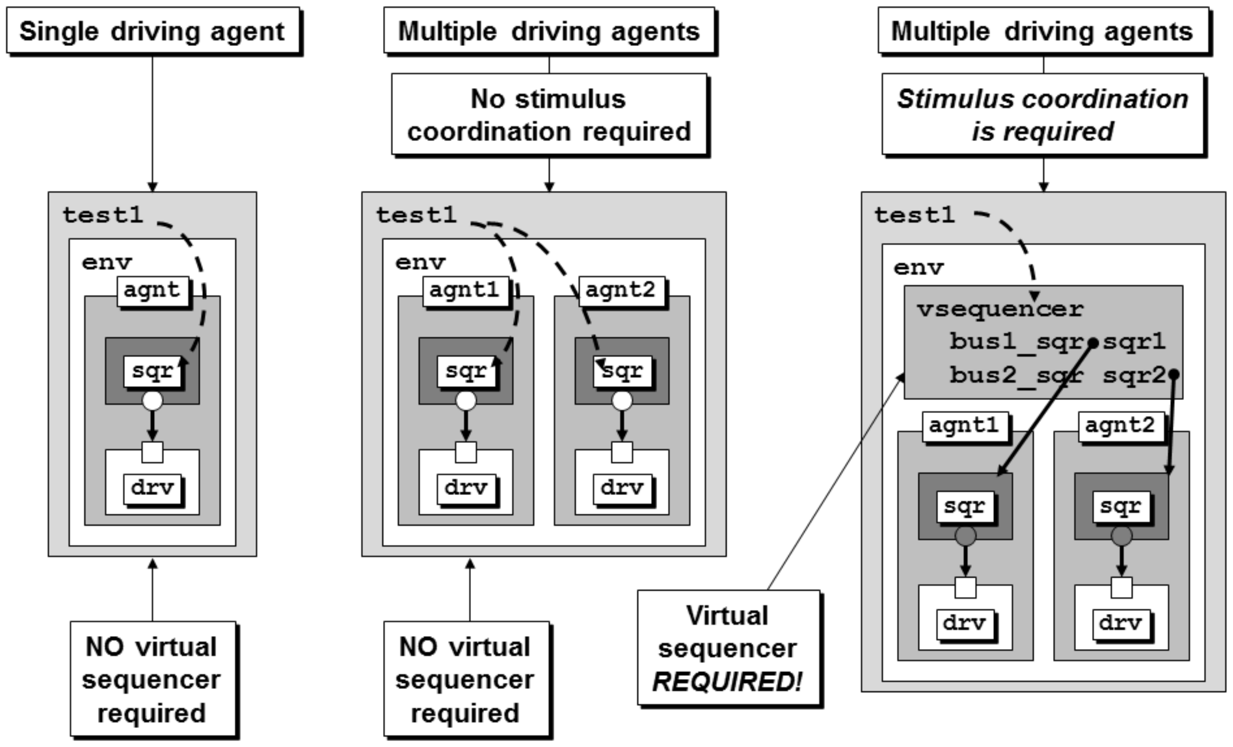


Figure 1 - When is a virtual sequencer required?

If you only have a single driving agent, you do not need a virtual sequencer.

If you have multiple driving agents but no stimulus coordination is required, you do not need a virtual sequencer.

If you have multiple driving agents and stimulus coordination IS required, you need a virtual sequencer.

It should be noted that if a testbench with multiple agents and non-coordinated stimulus is ever extended in the future to require coordinated stimulus, then the environment will require updates to include one or more virtual sequencers. Those updates, performed later in the projet, could be quite painful as opposed to building in a virtual sequencer from the beginning and taking advantage of the virtual sequencer when needed. Engineers might want to make a habit of adding the virtual sequencer in most of their UVM testbenches.

Why "virtual" sequencer/sequence

SystemVerilog has virtual classes, virtual methods and virtual interfaces and all three require the "virtual" keyword.

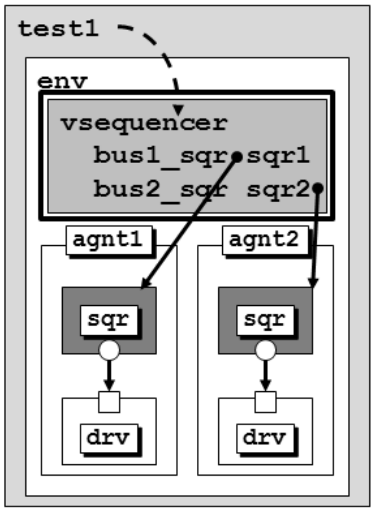
UVM has virtual sequencers and virtual sequences but neither one requires the "virtual" keyword. There are no **`uvm_virtual_sequencer`** or **`uvm_virtual_sequence`** base classes in UVM. All sequencers and virtual sequencers are derivatives of the **`uvm_sequencer`** class and all sequences and virtual sequences are derivatives of the **`uvm_sequence`** class.

So why are virtual sequencers and virtual sequences "virtual?"

Three attributes of a virtual sequencer are:

- It controls other sequencers.
- It is not attached to a driver.
- It does not process items itself.

A virtual sequencer is not connected to a driver. Instead of executing individual sequence items on a driver via a sequencer port, it executes subsequences and sequence items on sequencers via handles to subsequencer targets. The UVM User guide[3] sometimes refers to the subsequencers as "driver-sequencers." A virtual sequencer is "virtual" because typically an engineer is not really running sequences on this sequencer, the sequences are being run on the subsequencers via handles defined in the virtual sequencer.



A virtual sequence can run multiple transaction types on multiple real sequencers. The virtual sequence is typically just coordinating execution of the other sequences on the appropriate subsequencers.

3 virtual sequencer modes:

The UVM User Guide describes three ways a user can use virtual sequences to interact with subsequencers: (1) "Business as usual" (also known as parallel traffic generation), (2) Disable subsequencers, and (3) Use **`grab()`** and **`ungrab()`**.

The UVM User Guide claims that "most users disable the subsequencers and invoke sequences only from the virtual sequence," but our experience and the experience of many verification colleagues is that the most popular virtual sequencer mode is parallel tragffic generation, also known as "business as usual." This is the mode that is described in this paper.

How are virtual sequencers implemented?

A virtual sequencer is little more than a component providing a locus and scope to configure virtual sequences and provide handles to the subsequencers that will be required by virtual sequences.

The code for a virtual sequencer is rather simple. The subsequencer handles declared in the virtual sequencer will be specified, via the configuration database, after all components are built (after the **`build_phase()`**) and are typically set by the environment in the **`connect_phase()`**.

Consider the virtual sequencer code in Example 1.

```
class vsequencer extends uvm_sequencer;
  `uvm_component_utils(vsequencer)
  tb_ahb_sequencer ahb_sqr;
  tb_eth_sequencer eth_sqr;

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

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    if (!uvm_config_db#(tb_ahb_sequencer)::get(this, "ahb_sqr", "", ahb_sqr)) begin
        `uvm_fatal("VSQR/CFG/NOAHB", "No ahb_sqr specified for this instance");
    end
    if (!uvm_config_db#(tb_eth_sequencer)::get(this, "eth_sqr", "", eth_sqr)) begin
        `uvm_fatal("VSQR/CFG/NOETH", "No eth_sqr specified for this instance");
    end
  endfunction
endclass
```

Example 1 - Sample virtual sequencer code

Example 1 is a typical structure for a virtual sequencer. The user-selected name for this example is **`vsequencer`**. Virtual sequencers are extended from **`uvm_sequencer`**, NOT **`uvm_virtual_sequencer`** (which does not exist). Unlike normal sequencers, the virtual sequencer of Example 1 is not user-parameterized to a transaction type because this sequencer will be able to execute multiple transaction types. Extending a virtual sequencer from the **`uvm_sequencer`** base class without any parameters means that the virtual sequencer will use the default parameterized values of **`uvm_sequence_item`**.

The virtual sequencer declares subsequencer handles. In Example 1, the subsequencer handles are called **`ahb_sqr`** and **`eth_sqr`** respectively. These two subsequencer handles will be assigned from values specified in the configuration database during the **`end_of_elaboration_phase()`**.

Unlike Transaction Level Model (TLM) connections that are used to connect most components in a UVM testbench, the subsequencer handles are not set using a TLM **`connect()`** method, but are specified by the environment using the configuration database. It is then the job of the virtual sequencer to extract those handles from the configuration database and assign them to the two handles declared in the virtual sequencer. The actual subsequencers will be created in the **`build_phase()`**. Therefore, their handles will only be available to be put in the configuration database by the environment in its **`connect_phase()`**. Thus, the virtual sequencer will have to retrieve them in the next phase: **`end_of_elaboration_phase()`**.

Finally, the **`vsequencer`** example includes the typical **`new()`** constructor that is common to all UVM components.

It can be seen from this example, that the **`vsequencer`** is just a container for the handles to subsequencers and other configuration parameters. The virtual sequences assume the virtual sequencer has been properly configured before the virtual sequences execute in the **`run_phase()`**. They can then access these configuration paratemers in the virtual sequencer via their **`p_sequencer`** handle.

Sequence Details

Sequences are run on a sequencer and are parameterized to the transaction type that is processed by that sequencer.

Sequences are started on a sequencer using the built-in sequence **`start()`** method or by using the **``` `uvm_do() ```** macros.

Every sequence has a handle to the sequencer that is running that sequence. That handle is called the **`m_sequencer`** handle.

What is the m_sequencer handle?

All sequences are started on sequencers: **`tr_seq.start(env.vsqr)`**. The **``` `uvm_do ```** macros also execute this command. After starting a sequence on a sequencer, the **`m_sequencer`** handle for the sequence is set to **`env.vsqr`**. The **`m_sequencer`** handle is just a handle in every sequence that points to the sequencer that is running that sequence and it was set when the **`start()`** method was passed the handle of the sequencer (**`env.vsqr`** in this case).

Just like any other sequence, when a virtual sequence is started on a virtual sequencer, using either the **`start()`** method or the **``` `uvm_do ```** macros, the virtual sequence will automatically have an **`m_sequencer`** handle that correctly points to the virtual sequencer.

What is the p_sequencer handle?

Frequently asked questions include:
- What is the **`p_sequencer`**?
- How is the **`p_sequencer`** different from the **`m_sequencer`**?

All sequences have an **`m_sequencer`** handle but sequences do not automatically have a **`p_sequencer`** handle. Furthermore, the **`m_sequencer`** variable is an internal implementation variable that is poorly documented and should not be used directly by verification engineers. It is an artifact of the SystemVerilog language, which lacks C++'s concept of “friend” classes that this variable is public. Any variable or method with the “**`m_`**” prefix should similarly not be used directly.

`p_sequencer` is not automatically declared and set, but can be declared and set by using the **``` `uvm_declare_p_sequencer ```** macro. As will be shown later in this paper, the **``` `uvm_declare_p_sequencer ```** macro and **`p_sequencer`** handle are user-conveniences.

Technically, the **`p_sequencer`** handle is never required but when used with the **``` `uvm_declare_p_sequencer ```** macro, it is automatically (1) declared, (2) set and (3) checked when a virtual sequence is started, and properly points to the virtual sequencer that is running the virtual sequence.

More about the **`p_sequencer`** handle and its usage is described below.

What is the `uvm_declare_p_sequencer(SEQUENCER) macro?

The **``` `uvm_declare_p_sequencer ```** macro code is defined in the **`src/macros/sequence_define.svh`** file and is rather simple code:

```
1  `define uvm_declare_p_sequencer(SEQUENCER) \
2    SEQUENCER p_sequencer;\
3    virtual function void m_set_p_sequencer();\
4      super.m_set_p_sequencer(); \
5      if( !$cast(p_sequencer, m_sequencer)) \
6          `uvm_fatal("DCLPSQ", \
7          $sformatf("%m %s Error casting p_sequencer, please verify that this
7a         sequence/sequence item is intended to execute on this type of sequenc-
er",
8          get_full_name())) \
9    endfunction
```

Figure 2 - `uvm_declare_p_sequencer macro definition

The **`uvm_declare_p_sequencer(***SEQUENCER***)** macro executes two useful steps:
(1) The macro declares a **p_sequencer** handle of the *SEQUENCER* type.
(2) The macro then **$casts** the **m_sequencer** handle to the **p_sequencer** handle and checks to make sure the sequencer executing this sequence is of the appropriate type.

A closer look at this macro and what it does is instructive. This macro is typically placed in a sequence base class that will be extended to create all of the sequences that use the designated sequencer, virtual or not.

On line 1, the user calls this macro and passes the type of the sequencer that will be used by the sequences. For virtual sequences, this is the class name of the designated virtual sequencer they will execute on.

On line 2, the designated sequencer is declared with the handle name **p_sequencer**. For the remainder of the code in this macro and everywhere else in the user-defined virtual sequence base and extended virtual sequence classes, the virtual sequencer will be referenced by the name **p_sequencer**. From this point forward, there is no need to reference the name of the virtual sequencer that is being used, the user can simply reference the **p_sequencer** (virtual sequencer) handle. This is simply a convenience, not a requirement.

On line 3 is the start of a **virtual void function** declaration that continues through line 9. The void function is called **m_set_p_sequencer** and this function is called whenever a sequence **start()** method is called on one of the virtual sequences or when a **`uvm_do_on()** macro is used to start a virtual sequence.

Line 4 ensures that if the virtual sequence is an extension of another virtual sequence, the base virtual sequence will also execute its own **m_set_p_sequencer** method .

Line 5 casts the internal **m_sequencer** handle, which should be the handle of the virtual sequencer to the local **p_sequencer** handle declared on line 2. The **if**-test checks to see if the **$cast** operation failed (**!$cast(...)**) and if the **$cast** did fail, the fatal message on lines 6-8 will terminate the UVM simulation and report a consistent message that there was a problem casting to the specified virtual sequencer type, i.e. the sequence is executing on a sequencer of the wrong type. The **if**-test, **$cast** operation and corresponding consistent error message are also shown in Figure 3.

```
5      if( !$cast(p_sequencer, m_sequencer)) \
6        `uvm_fatal("DCLPSQ", \
7        $sformatf("%m %s Error casting p_sequencer, please verify that this
7a       sequence/sequence item is intended to execute on this type of sequencer",
8        get_full_name())) \
```
(NOTE: the **$sformatf()** command is one long string on one line of code, lines 7 & 7a, in the macro).

Figure 3 - `uvm_declare_p_sequencer: casts m_sequencer to p_sequencer

Example virtual sequencer testbench

Trying to describe virtual sequencer testbench construction and operation without a block diagram requires a great deal of concentration on the part of the reader, so example files to run virtual sequences on the virtual sequencer testbench of Figure 4 will be described in this paper. Any files required to run this simulation that are not described in the body of this paper have been added to Appendix 2 at the end of this paper.

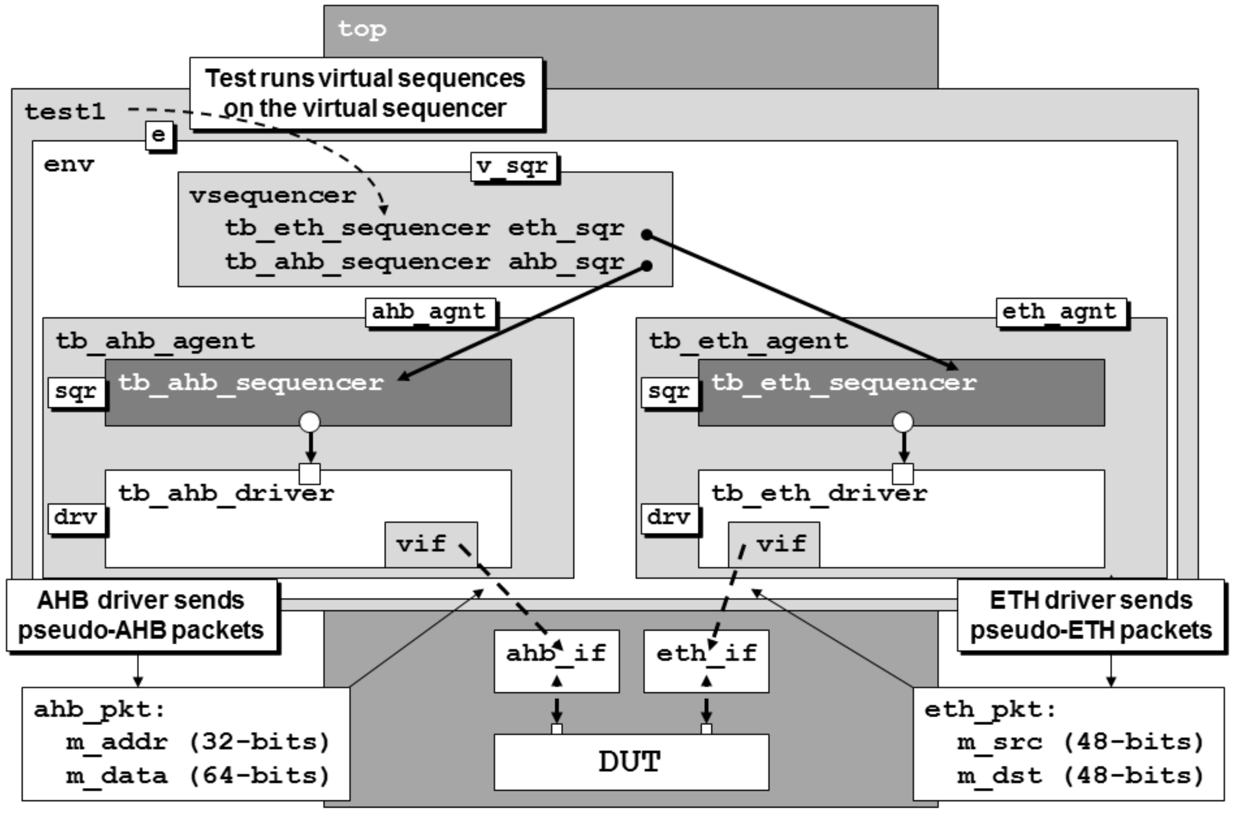


Figure 4 - Example virtual sequencer / sequence block diagram

The virtual sequencer for this testbench was shown in Example 1. All other testbench files will be described in the remainder of this paper.

Virtual sequence base classes.

All virtual sequences need access to the subsequencer handles defined in the virtual sequencer. To gain access to the subsequencer handles, virtual sequences need to use the **`` `uvm_declare_p_sequencer ``** macro to declare and set the **`p_sequencer`** variable so the subsequencer handles are readily accessible.

Since every virtual sequence needs to execute these steps, it is recommended to put this code into a virtual sequence base class (**`vseq_base`**) and then create all virtual sequences by extending the **`vseq_base`** class.

Example vseq_base

For the virtual sequencer shown in Example 1, we can use the **vseq_base** definition shown in Example 2.

```
class vseq_base extends uvm_sequence;
  `uvm_object_utils(vseq_base)
  `uvm_declare_p_sequencer(vsequencer)

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

  tb_ahb_sequencer ahb_sqr;
  tb_eth_sequencer eth_sqr;

  virtual task body();
    ahb_sqr = p_sequencer.ahb_sqr;
    eth_sqr = p_sequencer.eth_sqr;
  endtask
endclass
```

Example 2 - Virtual sequence base class example called vseq_base

The **vseq_base** class uses the **`uvm_declare_p_sequencer(vsequencer)** macro to declare a **p_sequencer** handle of the **vsequencer** type. The **vseq_base** then declares **ahb_sqr** and **eth_sqr** handles of the same types that were declared in the virtual sequencer shown in Example 1. The **vseq_base** then copies the virtual sequencer (**p_sequencer**) **ahb_sqr** and **eth_sqr** handles to the local **ahb_sqr** and **eth_sqr** handles. The **vseq_base** class used the **p_sequencer** handle (which should have been properly assigned by the **`uvm_declare_p_sequencer** macro) to copy the handles from the virtual sequencer to this virtual sequence base class. Using the **`uvm_declare_p_sequencer** macro, it was not necessary for the **vseq_base** class to check the type of the **vse-quencer** class since the macro setup a void function to perform that check.

Note that the **vseq_base** class assumes that the handles were already set in the virtual sequencer. It is the job of the enviroinment to ensure that the subsequencer handles are properly set.

Creating virtual sequences

Once the virtual sequence base class has been created, it is possible to create virtual sequences that are an extension of the virtual sequence base class. Every virtual sequence that is extended from the virtual base class inherits the subsequencer handles of the correct type, and already properly assigned.

Consider the two virtual sequence examples shown in Example 3 and Example 4. These sequences are examples of virtual sequences that are extensions of the base virtual sequence shown in Example 2.

There are two accepted methods for executing sequences in UVM: (1) use the **`uvm_do** macros, which are generally considered to be the easiest to use but may also be less simulation efficient (because the subsequences are always allocated and randomized before being executed) and more difficult to understand if the user ever expands the **`uvm_do** macro code, and (2) use explicit allocation, and direct assignments or calls to **randomize()** before using the **start()** method to execute the sequences on the chosen subsequencer, which is generally considered to require more user-coding effort but that are straightforward and allow the creation and execution of more directed sequenc-es.

The Example 3 vitual sequence uses the **`uvm_do** macros to run a virtual sequence to randomly generate pseudo-AHB packets followed by two sequences of pseudo-Ethernet packtes and concludes with another sequence of pseu-do-AHB packets. The code for the pseudo-ethernet and AHB transactions, along with the Ethernet and AHB se-quences, and the test that runs **v_seq1** will be shown later.

```
class v_seq1 extends vseq_base;
  `uvm_object_utils(v_seq1)

  function new(string name="v_seq1");
    super.new(name);
  endfunction

  virtual task body();
    ahb_seq1 ahb_seq;
    eth_seq1 eth_pkts;
    //-----------------------------------------------
    super.body();
    `uvm_info("v_seq1", "Executing sequence", UVM_HIGH)
    `uvm_do_on(ahb_seq,  ahb_sqr)
    `uvm_do_on(eth_pkts, eth_sqr)
    `uvm_do_on(eth_pkts, eth_sqr)
    `uvm_do_on(ahb_seq,  ahb_sqr)
    `uvm_info("v_seq1", "Sequence complete", UVM_HIGH)
  endtask
endclass
```

Example 3 - v_seq1 - extended from vseq_base - uses `uvm_do_on() macros

The Example 4 vitual sequence uses calls to the **`randomize()`** and **`start()`** methods to run a virtual sequence to randomly generate pseudo-AHB packets followed by two sequences of pseudo-Ethernet packtes and concludes with another sequence of pseudo-AHB packets. The code for the pseudo-ethernet and AHB packets, along with the Ethernet and AHB sequences, and the test that runs **`v_seq2`** will be shown later.

```
class v_seq2 extends vseq_base;
  `uvm_object_utils(v_seq2)

  function new(string name="v_seq2");
    super.new(name);
  endfunction

  virtual task body();
    ahb_seq1 ahb_seq = ahb_seq1::type_id::create("ahb_seq");
    eth_seq1 eth_seq = eth_seq1::type_id::create("eth_seq");
    //-----------------------------------------------
    super.body();
    `uvm_info("v_seq2", "Executing sequence", UVM_HIGH)
    if(!ahb_seq.randomize()) `uvm_error("RAND","FAILED");
    ahb_seq.start(ahb_sqr);
    if(!eth_seq.randomize()) `uvm_error("RAND","FAILED");
    eth_seq.start(eth_sqr);
    if(!eth_seq.randomize()) `uvm_error("RAND","FAILED");
    eth_seq.start(eth_sqr);
    if(!ahb_seq.randomize()) `uvm_error("RAND","FAILED");
    ahb_seq.start(ahb_sqr);
    `uvm_info("v_seq2", "Sequence complete", UVM_HIGH)
  endtask
endclass
```

Example 4 - v_seq2 - extended from vseq_base - uses sequence.randomize() and sequence.start() methods

Calling sequences from virtual sequences

One important feature of virtual sequences is that they can run exsiting sequences without modification. The user can create a library of sequences that are used to test individual subblocks and then use the same sequences in a co-ordinated virtual sequence to test multiple subblocks using the original subblock sequence libraries. There is no need to re-code or modify the subblock sequences.

Consider the pseudo AHB packet code shown Example 5. This is a "pseudo-AHB packet" because it only contains two non-standard AHB fields and an example DUT will recognize when these fields have been sent to the DUT and print that information to the screen for examination.

```
class ahb_pkt extends uvm_sequence_item;
  `uvm_object_utils(ahb_pkt)
  rand bit [31:0] addr;
  rand bit [63:0] data;

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

  virtual function string convert2string();
    return $sformatf("addr=%8h, data=%16h", addr, data);
  endfunction
endclass
```

Example 5 - ahb_pkt.sv - pseudo AHB packet code

Consider the AHB sequence code shown Example 6. This simple AHB sequence randomly generates 2-5 AHB packets. The virtual sequences in Example 3 and Example 4 send this randomly generated set of AHB packets to the `ahb_sqr` handle declared in the `vsequencer` component.

```
class ahb_seq1 extends uvm_sequence #(ahb_pkt);
  `uvm_object_utils(ahb_seq1);

  rand int cnt;
  constraint c1 {cnt inside {[2:5]};}

  function new(string name = "ahb_seq1");
    super.new(name);
  endfunction

  virtual task body();
    ahb_pkt ahb_pkt1;
    `uvm_info("AHBcnt", $sformatf("** Loop cnt=%0d **", cnt), UVM_MEDIUM)
    repeat(cnt) `uvm_do(ahb_pkt1)
  endtask
endclass
```

Example 6 - ahb_seq1.sv - AHB sequence code called by virtual sequences

Consider the pseudo Ethernet packet code shown Example 7. This is a "pseudo-Ethernet packet" because it only contains two non-standard Ethernet fields and an example DUT will recognize when these fields have been sent to the DUT and print that information to the screen for examination.

```
class eth_pkt extends uvm_sequence_item;
  `uvm_object_utils(eth_pkt)
  rand bit [47:0] src;
  rand bit [47:0] dst;

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

  function string convert2string();
    return $sformatf("src=%12h, dst=%12h", src, dst);
  endfunction
endclass
```

Example 7 - eth_pkt.sv - pseudo Ethernet packet code

Consider the Ethernet sequence code shown Example 8. This simple Ethernet sequence randomly generates 2-4 Ethernet packets. The virtual sequences in Example 3 and Example 4 send this randomly generated set of Ethernet packets to the **eth_sqr** handle declared in the **vsequencer** component.

```
class eth_seq1 extends uvm_sequence #(eth_pkt);
  `uvm_object_utils(eth_seq1);

  rand int cnt;
  constraint c1 {cnt inside {[2:4]};}

  function new(string name = "eth_seq1");
    super.new(name);
  endfunction

  virtual task body();
    eth_pkt eth_pkt1 = eth_pkt::type_id::create("eth_pkt1");
    `uvm_info("ETHcnt", $sformatf("** Loop cnt=%0d **", cnt), UVM_MEDIUM)
    repeat(cnt) begin
      start_item(eth_pkt1);
      if(!eth_pkt1.randomize()) `uvm_error("RAND", "FAILED")
      finish_item(eth_pkt1);
    end
  endtask
endclass
```

Example 8 - eth_seq1.sv - Ethernet sequence code called by virtual sequences

Starting virtual sequences

In general, virtual sequences are started from a test using the sequence **start()** method. The **`uvm_do_on** macro cannot be called from a test component. The **`uvm_do_on** macro is only called from derivatives of sequences. This was a bit tricky to determine but even if a **`uvm_do_on** is called from a test on a valid virtual sequencer handle, the **`uvm_do_on** macro calls methods that are defined in the **uvm_sequence_base** class located in the **uvm/src/seq/uvm_sequence_base.svh** file. The required methods are **create_item()**, **start_item()** and **finish_item()** and none of these methods are available in the **uvm_test** base class or any other **uvm_component** class or derivative.

It is a good idea to create a **test_base** class with common declarations and methods that will be used by every other test in the verification suite. The **test_base** class shown in Example 9, declares the environment handle and creates the environment in the **build_phase()**. These actions will not need to be repeated in tests that extend this **test_base** class. This **test_base** also includes a **start_of_simulation_phase()** to print the testbench structure and factory contents before the simulation executes in the **run_phase()**. It is useful to print the testbench structure and factory contents in the **start_of_simulation_phase()** because troubles most often appear in the **run_phase()** so these pre-run printouts can help diagnose if any components were incorrectly constructed or if some of the testbench classes were omitted from the factory.

```
`timescale 1ns/1ns
class test_base extends uvm_test;
  `uvm_component_utils(test_base)

  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 = 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();
    end
  endfunction
endclass
```

Example 9 - test_base.sv - declares and builds the environment and prints the testbench structure

Once a **`test_base`** class is coded, each of the tests can extend the **`test_base`** to create the individual tests. Example 10 shows the **`test1`** class definition that is extended from the **`test_base`** class. The **`test1`** example defines the **`run_phase()`** for this test, which declares the first virtual sequence (**`v_seq1`**) handle **`vseq`** and creates the **`vseq`** object. The test then calls the **`raise_objection()`** method, prints a message, calls the **`start()`** method on the **`vseq`** sequence and passes the environment-virtual sequencer path (**`e.v_sqr`**) to the **`start()`** method. Once the virtual sequence has completed, the test prints one more message and then calls the **`drop_objection()`** method and finishs.

```
`timescale 1ns/1ns
class test1 extends test_base;
  `uvm_component_utils(test1)

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

  task run_phase(uvm_phase phase);
    v_seq1 vseq = v_seq1::type_id::create("vseq");
    uvm_test_done.raise_objection(this);
    `uvm_info("test1 run", "Starting test", UVM_MEDIUM)
    vseq.start(e.v_sqr);
    `uvm_info("test1 run", "Ending test", UVM_MEDIUM)
    uvm_test_done.drop_objection(this);
  endtask
endclass
```

Example 10 - test1.sv - declares a v_seq1 vseq handle and calls vseq.start(e.v_sqr)

The **`test2`** code in Example 11 does the same thing as the **`test1`** code of Example 10 except that the **`test2`** code declares the **`vseq`** handle to be the second virtual sequence type, **`v_seq2`**.

```
`timescale 1ns/1ns
class test2 extends test_base;
  `uvm_component_utils(test2)

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

  task run_phase(uvm_phase phase);
    v_seq2 vseq = v_seq2::type_id::create("vseq");
    uvm_test_done.raise_objection(this);
    `uvm_info("test2 run", "Starting test", UVM_MEDIUM)
    vseq.start(e.v_sqr);
    `uvm_info("test2 run", "Ending test", UVM_MEDIUM)
    uvm_test_done.drop_objection(this);
  endtask
endclass
```

Example 11 - test2.sv - declares a v_seq2 vseq handle and calls vseq.start(e.v_sqr)

The environment sets the handles in the virtual sequencer

The environment code shown in Example 12 is pretty typical environment code with the exceptions that it declares a virtual sequencer handle (**`vsequencer v_sqr`**), builds the virtual sequencer, and stores the Ethernet (**`eth_agnt`**) and AHB (**`ahb_agnt`**) sequencer handles (**`sqr`** and **`sqr`**) in the configuration database . As was discussed in the section "How are virtual sequencers implemented?" the virtual sequencer handles are stored in the configuration database by the environment in the **`connect_phase()`** for retrieval by the virtual sequencer in the **`end_of_elaboration()`** phase as opposed to using TLM connections that connect most testbench components.

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

  tb_eth_agent   eth_agnt;
  tb_ahb_agent   ahb_agnt;
  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);
    eth_agnt  =  tb_eth_agent::type_id::create("eth_agnt", this);
    ahb_agnt  =  tb_ahb_agent::type_id::create("ahb_agnt", this);
    v_sqr     =    vsequencer::type_id::create("v_sqr"   , this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    uvm_config_db#(tb_ahb_sequencer)::set(this,"*","ahb_sqr",ahb_agnt.sqr);
    uvm_config_db#(tb_eth_sequencer)::set(this,"*","eth_sqr",eth_agnt.sqr);
  endfunction
endclass
```

Example 12 - env.sv - Environment with virtual sequencer

m_sequencer handle creation - details

For those readers who want to know the details about how the `m_sequencer` handle is created inside of UVM, see Appendix 1.

Summary

This paper describes the necessary steps to create a working virtual sequencer environment and explains the purpose of `m_sequencer` and `p_sequencer` handles and the `` `uvm_declare_p_sequencer `` macro. These are topics that are often confusing to new and experienced UVM verification engineers.

This paper also includes all the code necessary to test the example described in this paper.

Acknowledgements

The authors would like to thank the following colleagues: JL Gray who helped Cliff put together the first version of the Virtual Sequence / Sequencer example shown in this paper many years ago. Mark Litterick of Verilab for helping to refine some of the techniques used in this example especially related to the use of the `p_sequencer` handle and `` `uvm_declare_p_sequencer `` macro. Heath Chambers who has helped correct these materials for use in UVM training. Multiple Verilab engineers who responded to Cliff's survey years ago regarding their preferred virtual sequencer usage mode. Logie Ramachandran for providing valuable feedback and suggestions for this paper.

References:

[1] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2012

[2] Universal Verification Methodology (UVM) 1.2 Class Reference, June 2014, Accellera

[3] Universal Verification Methodology (UVM) 1.1 User's Guide, May 2011, Accellera

Appendix 1 **m_sequencer handle creation - details**

If you trust that the **m_sequencer** handle is properly created in the UVM Base Class Library (BCL) and if you don't care how the **m_sequencer** handle is created, then you do not need to read this section. This section is included for those who wish to understand the details of how UVM creates the **m_sequencer** handle.

Finding the correct UVM base class routines to create the **m_sequencer** handle is a bit tricky. Assuming the use of a sequence called **tr_seq**, that is started on the **e** (environment) **agnt1** (agent) **sqr** (sequencer), here is how the **m_sequencer** handle is created:

(1) All sequences are started on sequencers: **tr_seq.start(e.agnt1.sqr);**

(2) **uvm_sequence** inherits the **start()** method from the **uvm_sequence_base** class.

(3) The inherited **start()** task is defined as shown in Figure 5.

```
virtual task start (uvm_sequencer_base sequencer,
                    uvm_sequence_base parent_sequence = null,
                    int this_priority = -1,
                    bit call_pre_post = 1);
  set_item_context(parent_sequence, sequencer);
```

Figure 5 - start() task method definition

Typically only the sequencer handle is passed to the **start()** method and then **uvm_sequence_base** calls **set_item_context(null, e.agnt1.sqr);**

(4) The **set_item_context()** method is defined in the **uvm_sequence_item** class, which is inherited by the **uvm_sequence_base** and **uvm_sequence** classes.

(5) The **set_item_context()** method calls the **set_sequencer(e.agnt1.sqr)** method.

(6) The **set_sequencer()** method is defined in the **uvm_sequence_item** class (inherited by **uvm_sequence_base** and **uvm_sequence**).

(7) The **set_sequencer()** method sets the **m_sequencer** handle, declared to be a **protected uvm_sequencer_base** handle in the **uvm_sequence_item** class.

```
protected  uvm_sequencer_base m_sequencer;
…
virtual function void set_sequencer(uvm_sequencer_base sequencer);
  m_sequencer = sequencer;
  m_set_p_sequencer();
endfunction
```

Figure 6 - protected uvm_sequencer_base m_sequencer declaration and set_sequencer() method

To summarize, the **tr_seq.start()** method passes a sequencer handle to the **set_item_context()** method, which passes the handle to the **set_sequencer()** method, which sets the **m_sequencer** handle. The **m_sequencer** handle is a handle to the sequencer that is running this sequence (**tr_seq** in this example). The **m_sequencer** handle can be retrieved by calling the **get_sequencer()** method.

Appendix 2 **UVM virtual sequencer /sequence example code**

The code that corresponds to the virtual sequencer testbench shown in Figure 4 is included in this appendix. Each of the files for this example are listed in alphabetical order in the appendix subsections.

Appendix 2.1 **ahb_if.sv**

```
interface ahb_if;
  logic [31:0] ahb_addr;
  logic [63:0] ahb_data;
endinterface
```

Example 13 - ahb_if.sv code

Appendix 2.2 **ahb_pkt.sv**

This code is shown in Example 5 - ahb_pkt.sv - pseudo AHB packet code

Appendix 2.3 **ahb_seq1.sv**

This code is shown in Example 6 - ahb_seq1.sv - AHB sequence code called by virtual sequences

Appendix 2.4 **dut.sv**

```
module dut (
  input logic [31:0] ahb_addr,
  input logic [63:0] ahb_data,
  input logic [47:0] eth_src,
  input logic [47:0] eth_dst );

  import uvm_pkg::*;
  `include "uvm_macros.svh"

  always @* begin
    `uvm_info("DUT AHB",  $sformatf("ahb_addr=%8h       ahb_data=%16h",
                                    ahb_addr, ahb_data), UVM_MEDIUM);
  end

  always @* begin
    `uvm_info("DUT ETH",  $sformatf("eth_src =%12h   eth_dst =%12h",
                                    eth_src, eth_dst), UVM_MEDIUM);
  end
endmodule
```

Example 14 - dut.sv code

Appendix 2.5 **env.sv**

This code is shown in Example 12 - env.sv - Environment with virtual sequencer

Appendix 2.6 **eth_if.sv**

```
interface eth_if;
  logic [47:0] eth_src;
  logic [47:0] eth_dst;
endinterface
```

Example 15 - eth_if.sv

Appendix 2.7 **eth_pkt.sv**

This code is shown in Example 7 - eth_pkt.sv - pseudo Ethernet packet code

Appendix 2.8 **eth_seq1.sv**

This code is shown in Example 8 - eth_seq1.sv - Ethernet sequence code called by virtual sequences

Appendix 2.9 **run.f**

```
tb_pkg.sv
top.sv
dut.sv
ahb_if.sv
eth_if.sv
```

Example 16 - run.f code

Appendix 2.10 **tb_ahb_agent.sv**

```
class tb_ahb_agent extends uvm_agent;
  `uvm_component_utils(tb_ahb_agent)

  virtual ahb_if vif;

  tb_ahb_driver    drv;
  tb_ahb_sequencer sqr;

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

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Instantiate driver and sequencer for AHB agent
    drv =    tb_ahb_driver::type_id::create("drv", this);
    sqr = tb_ahb_sequencer::type_id::create("sqr", this);
    get_vif();
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    drv.seq_item_port.connect(sqr.seq_item_export);
    drv.vif = vif;
  endfunction

  function void get_vif;
    if(!uvm_config_db#(virtual ahb_if)::get(this,"","ahb_vif",vif))
        `uvm_fatal("NOVIF",{"virtual interface must be set for:",
                        get_full_name(),".vif"})
  endfunction
endclass
```

Example 17 - tb_ahb_agent.sv code

Appendix 2.11 **tb_ahb_driver.sv**

```
class tb_ahb_driver extends uvm_driver #(ahb_pkt);
  `uvm_component_utils(tb_ahb_driver)

  virtual ahb_if vif;

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

  task run_phase(uvm_phase phase);
    ahb_pkt apkt;
    forever begin
      seq_item_port.get_next_item(apkt);
      drive_item(apkt);
      seq_item_port.item_done();
    end
  endtask

  task drive_item(ahb_pkt tr);
    `uvm_info("ahb_driver-run", "Driving AHB transaction...", UVM_HIGH)
    #134;
    vif.ahb_addr= tr.addr;
    vif.ahb_data= tr.data;
    `uvm_info("ahb_driver-run", "Finished AHB transaction...", UVM_HIGH)
  endtask
endclass
```

Example 18 - tb_ahb_driver.sv code

Appendix 2.12 **tb_ahb_sequencer.sv**

```
class tb_ahb_sequencer extends uvm_sequencer #(ahb_pkt);
  `uvm_component_utils(tb_ahb_sequencer)

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

Example 19 - tb_ahb_sequencer.sv code

Appendix 2.13 **tb_eth_agent.sv**

```
class tb_eth_agent extends uvm_agent;
  `uvm_component_utils(tb_eth_agent)

  virtual eth_if vif;

  tb_eth_driver    drv;
  tb_eth_sequencer sqr;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Instantiate driver and sequencer for ethernet agent
    drv =    tb_eth_driver::type_id::create("drv", this);
    sqr = tb_eth_sequencer::type_id::create("sqr", this);
    get_vif();
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    drv.seq_item_port.connect(sqr.seq_item_export);
    drv.vif = vif;
  endfunction

  function void get_vif;
    if(!uvm_config_db#(virtual eth_if)::get(this,"","eth_vif",vif))
        `uvm_fatal("NOVIF",{"virtual interface must be set for:",
                             get_full_name(),".vif"})
  endfunction
endclass
```

Example 20 - tb_eth_agent.sv code

Appendix 2.14 **tb_eth_driver.sv**

```
class tb_eth_driver extends uvm_driver #(eth_pkt);
  `uvm_component_utils(tb_eth_driver)

  virtual eth_if vif;

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

  task run_phase(uvm_phase phase);
    eth_pkt epkt;
    forever begin
      seq_item_port.get_next_item(epkt);
      drive_item(epkt);
      seq_item_port.item_done();
    end
  endtask

  task drive_item(eth_pkt tr);
    `uvm_info("eth_driver-run", "Driving ETH transaction...", UVM_HIGH)
    #90;
    vif.eth_src = tr.src;
    vif.eth_dst = tr.dst;
    `uvm_info("eth_driver-run", "Finished ETH transaction...", UVM_HIGH)
  endtask
endclass
```

Example 21 - tb_eth_driver.sv code

Appendix 2.15 **tb_eth_sequencer.sv**

```
class tb_eth_sequencer extends uvm_sequencer #(eth_pkt);
  `uvm_component_utils(tb_eth_sequencer)

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

Example 22 - tb_eth_sequencer.sv code

Appendix 2.16 **tb_pkg.sv**

```
package tb_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  `include "eth_pkt.sv"
  `include "ahb_pkt.sv"
  `include "ahb_transaction.sv"

  `include "eth_seq1.sv"
  `include "ahb_seq1.sv"

  `include "tb_eth_driver.sv"
  `include "tb_eth_sequencer.sv"
  `include "tb_eth_agent.sv"

  `include "tb_ahb_driver.sv"
  `include "tb_ahb_sequencer.sv"
  `include "tb_ahb_agent.sv"

  `include "vsequencer.sv"

  `include "vseq_base.sv"
  `include "v_seq1.sv"
  `include "v_seq2.sv"

  `include "env.sv"

  `include "test_base.sv"
  `include "test1.sv"
  `include "test2.sv"
endpackage
```

Example 23 - tb_pkg.sv code

Appendix 2.17 **test1.sv**

This code is shown in Example 10 - test1.sv - declares a v_seq1 vseq handle and calls vseq.start(e.v_sqr)

Appendix 2.18 **test2.sv**

This code is shown in Example 11 - test2.sv - declares a v_seq2 vseq handle and calls vseq.start(e.v_sqr)

Appendix 2.19 **test_base.sv**

This code is shown in Example 9 - test_base.sv - declares and builds the environment and prints the testbench structure

Appendix 2.20 **top.sv**

```
`timescale 1ns/1ns
`include "uvm_macros.svh"

// Example of virtual sequences controlling two sub-sequencers...
// one for Ethernet and one for AHB
module top;
  import uvm_pkg::*;
  import  tb_pkg::*;

  dut    dut (.ahb_addr(ahb_if.ahb_addr), .ahb_data(ahb_if.ahb_data),
              .eth_src (eth_if.eth_src),  .eth_dst (eth_if.eth_dst));

  ahb_if ahb_if ();
  eth_if eth_if ();

  initial begin
    uvm_config_db#(virtual ahb_if)::set(null, "*", "ahb_vif", ahb_if);
    uvm_config_db#(virtual eth_if)::set(null, "*", "eth_vif", eth_if);
    run_test();
  end
endmodule
```

Example 24 - top.sv code

Appendix 2.21 **v_seq1.sv**

This code is shown in Example 3 - v_seq1 - extended from vseq_base - uses `uvm_do_on() macros

Appendix 2.22 **v_seq2.sv**

This code is shown in Example 4 - v_seq2 - extended from vseq_base - uses sequence.randomize() and sequence.start() methods

Appendix 2.23 **vseq_base.sv**

This code is shown in Example 2 - Virtual sequence base class example called vseq_base

Appendix 2.24 **vsequencer.sv**

This code is shown in Example 1 - Sample virtual sequencer code