UVM Rapid Adoption: A Practical Subset of UVM

Stuart Sutherland
Tom Fitzpatrick

***Abstract* — The Universal Verification Methodology (UVM) is a powerful verification methodology that was architected to be able to verify a wide range of design sizes and design types. One meaning of the "Universal" in the UVM name is that the methodology is intended to be capable of verifying anything and everything in the universe — at least all things in the realm of integrated circuits. There is value to this universal capability, but it also means there will likely be many capabilities in UVM that are not necessary for any specific project. Indeed, the authors maintain that there are many things in UVM that are not necessary for most projects. Furthermore, UVM originates from a blend of other verification methodologies such as OVM and VMM. These roots mean there are parts of UVM that were inherited from other methodologies, but which are not really needed in a pure UVM testbench. This paper focusses on defining a subset of the UVM base classes, methods, and macros that will enable engineers to learn UVM more quickly and become productive with using UVM for the verification of most types and sizes of digital designs modeled in VHDL, Verilog or SystemVerilog. You might be surprised at just how small of a subset of UVM is really needed in order to verify complex designs effectively with UVM.**

1.0 Introduction

The Universal Verification Methodology, commonly referred to as UVM, is purposely designed to have capabilities for verifying all types of digital logic designs, large or small, FPGA or ASIC or full-custom, and control-oriented or data-oriented or processor-oriented. This universal nature of UVM means there are constructs and capabilities in the methodology that, though perhaps useful for certain verification projects, are not necessary for most projects. The heritage of UVM further complicates the complexity of the methodology. UVM was not created from a clean slate. Rather, the UVM base class libraries and general testbench architecture were leveraged from other methodologies that have proven effective for verification of digital designs.

The universality and heritage of UVM has led to a bloated library of classes, methods, and macros. There are often multiple classes, methods or macros that do nearly the same thing, with only slight, often obscure, differences. For example, UVM provides both a *resource data base* and a *configuration data base*. A quick search of the internet will reveal scores of conflicting opinions as to which data base is best, or which data base should be used in this circumstance or that circumstance. Another example of redundancy is how UVM provides multiple ways to "report" a message. There are nearly identical reporting methods, such as `uvm_report_error()`, and reporting macros, such as `` `uvm_error ``.

The nature of Object Oriented Programming adds to the complexity of the UVM class libraries. Only some of the classes listed in the UVM Class Reference Manual are intended to be used by those who write UVM-based testbenches. These classes, following good OOP practices, extend and inherit from other base classes. Unfortunately, the UVM Class Reference Manual documents all of these classes – which it should – but the manual does not make it clear which classes are intended for end users of UVM to use in a UVM testbench, and which classes are intended for internal use within the UVM methodology.

The goal of this paper is to simplify learning and adopting UVM by suggesting preferred choices when redundancy exists, and clarifying which classes are for end-use of UVM. We will show that nearly all verification projects can be achieved with a simple subset of UVM classes, methods and macros. This "UVM-light" subset will help make UVM easier to learn, easier to maintain, easier to re-use, and less prone to coding errors or non-ideal coding styles.

1.1 History and Growth

UVM draws from a long and rich history of digital logic verification that spans more than 15 years of well-organized and documented methodologies, and many more years of company-specific ad-hoc methodologies. Established verification methodologies such as *vAdvisor*, *eRm*, *RVM*, *VMM*, *AVM*, *URM* and *OVM* have all contributed to the verification approaches used in UVM. While each of these methodologies had a common goal of

verifying digital logic designs, these methodologies often had very different approaches for accomplishing this goal. It is not the intent of this paper to provide the details of this heritage, but rather to emphasize that this heritage affected the bloat and redundancy that exists in UVM.

As a starting point, the UVM development committee drew substantially from the Open Verification Methodology (OVM) approach for architecting a verification testbench. Indeed, the first preliminary release of UVM, UVM 1.0EA ("EA" stood for "Early Adopter) in 2010, was mostly a direct port of OVM, with only a few changes or additions. UVM rapidly evolved from this early adopter version, adding in features from VMM, such as the Register Layer, and proven verification concepts from other documented and ad-hoc methodologies. The UVM 1.0 version was released just 9 months after 1.0EA, and was followed only 6 months later by UVM 1.1, which was released in June 2011. In that short period of about 15 months, UVM had evolved to be quite different than OVM, though the heritage is very apparent.

There were a number of minor updates to UVM 1.1 over the next three years, but, overall, there were no major evolutionary changes. The next major change to UVM occurred in June 2014, with the release of UVM 1.2. UVM 1.2 adds several substantial new capabilities to UVM, some of which will be discussed later in this paper. The evolution is not complete. UVM is now in the early stages of becoming an IEEE standard (designated IEEE P1800.2), and will very likely see more growth during that process.

This rapid evolution from an OVM starting point to UVM 1.1 and now to UVM 1.2 is a primary cause for the bloated nature of UVM. As UVM has evolved, new approaches for architecting a testbench and writing tests have been added to UVM, but the older ways of doing things are still available. Some older methodology features have been officially deprecated and others have been modified, but the baggage of the older features is still in the implementations of UVM class libraries and macros, still in use in existing UVM testbenches and Verification Intellectual Property (VIP), and still appears in books, papers and training courses. UVM's evolutionary process has resulted in having both older ways and newer ways to write a UVM testbench.

As previously noted, the purpose of this paper is to move much of this excess baggage into a closet, and show a simple subset of UVM classes, methods and macros that can handle nearly all verification projects. This recommended subset is based on UVM 1.2, but only uses constructs that are backward compatible with UVM 1.1.

1.2 Our Approach for This Paper

This paper examines UVM from three perspectives, the roles of the ***Test Writer***, the ***Environment Writer*** and the ***Sequence Writer***.

Note: *This is a "**what-to-use**" paper, not a "how-to-write" paper!*

We are not teaching *how* to write UVM tests, drivers, monitors and scoreboards in this paper. Nor are we discussing the definition, syntax and semantics of UVM constructs. Rather, we are focusing on *what* UVM constructs are really needed to write effective UVM tests, environments, and stimuli for most verification projects.

A UVM testbench can be partitioned into three primary parts: the test, the environment, and the sequence (stimuli). UVM is structured so that different engineers can focus on writing different parts of the testbench, with very little knowledge about the other portions. In this paper, we will examine the *what-to-use* by first considering a simple—but complete—UVM test, environment, and sequence. We will show exactly what UVM constructs a ***Test Writer***, ***Environment Writer*** and ***Sequence Writer*** need to know for this simple UVM verification project. We will then look at several advanced verification situations, and discuss any additional UVM constructs needed to handle those more difficult testbenches.

While the focus of this paper is on what UVM constructs *are needed* for simple and more advanced UVM verification, we will also discuss some constructs that are not needed. This discussion of unnecessary constructs is not comprehensive, however. Perhaps another way to think of this guide for a practical subset of UVM is simply: if a construct is not listed in the recommended subset, then you probably don't need it.

The UVM constructs that are not needed generally fall into one or more of four categories: 1) the construct is a left-over from UVM's OVM legacy, but was replaced by a more preferred UVM construct, 2) the construct is redundant with a construct that is in our recommended UVM subset (in which case we chose the construct we feel is the easiest or most versatile, 3) the construct might only be useful for some rare verification requirement, but is not a construct that is needed for the vast majority of verification projects, or 4) the construct is not intended for end-users of UVM, but rather is intended to be used internally within the UVM methodology.

2.0 Test Writer Guidelines

The ***Test Writer*** is responsible for using the environment and sequences available to specify stimuli and results checking, in order to verify a particular set of features in the Device Under Test (DUT). UVM is set up so that the ***Test Writer*** can specify useful tests without having to know very much of the underlying details of the environment, or of UVM itself. From the perspective of the ***Test Writer***, the UVM environment is mostly a black box. In this section, we will discuss the specific tasks relegated to the ***Test Writer*** and show which UVM constructs are required by the ***Test Writer***.

2.1 Connect to the DUT

The first task for the ***Test Writer*** is to connect the class-based UVM testbench to the module-based DUT in a top-level module[1]. This is done using the SystemVerilog `interface` construct. The ***Test Writer*** must make the instance of the interface, referred to as a "*virtual interface*", available to the environment. This is done by using the UVM configuration database. Example 1 shows a top-level module for a simple UVM testbench.

NOTE: Throughout this paper, UVM-specific code is shown in **bold text**. UVM-specific code that has not been discussed in previous examples is shown as **highlighted text**.

```
module test_top;
  import uvm_pkg::*;
  import my_test_pkg::*;

  my_dut_interface my_dut_if();
  my_dut_rtl my_dut(.if(my_dut_if());

  initial begin
    uvm_config_db #(virtual my_dut_interface)::set(null, "uvm_test_top",
                                                   "DUT_IF", my_dut_if);
    run_test();
  end
endmodule
```

Example 1 – Top-level module

Importing the `uvm_pkg` package is standard for all UVM testbenches. The code for the UVM environment is typically contained in one or more user-defined packages (`my_test_pkg` in this example). Importing this user-defined package, and the instantiation and connection of the testbench interface (`my_dut_interface`) are standard SystemVerilog programming. The use of UVM itself in this top-level module comes into play when we use the `uvm_config_db` data base to pass the virtual interface handle to the UVM test (and ultimately down into the environment).

The `uvm_config_db` data base is a parameterized class, where a parameter is used for the type of object being passed into the data base – in this case, a virtual interface of type `my_dut_interface`. The `uvm_config_db` class includes a static method called `set`. The ***Test Writer*** only needs to know the syntax of calling this method, and not the implementation of the data base. At this top-level, the first two arguments of the `set()` method will always be `null` and `"uvm_test_top"` respectively. In UVM, "`uvm_test_top`" always represents the instance name of the top-level test. The third argument is the "field name" in the data base, and is a string that will be used to identify the particular interface in the UVM test (`"DUT_IF"` in this example). Larger verification projects will often use multiple interfaces to connect the UVM testbench to the design. Each interface is given a unique field name in the data base. The last argument is the name of the interface instance. The instance name is referred to as a *virtual interface* handle for that instance. A combination of the first three arguments will be used by the ***Environment Writer*** to get the virtual interface back out of the configuration database.

The `run_test()` method is what starts UVM running. It causes an instance of the top-level test class (as specified on the command line via `+UVM_TESTNAME=<testname>`) to be created, and its methods run accordingly. Creating an instance of each interface being used between the testbench and the DUT, and passing them into the test via the `uvm_config_db` using the `set()` call, and starting UVM using `run_test()`, is all that the ***Test Writer*** needs to do in the top-level module to get a UVM test started.

[1] When developing a testbench for emulation, it is recommended to use dual top-level modules, as shown in [1].

2.2 UVM Tests

A UVM test has several basic responsibilities:

- Get the virtual interface handle(s) from the configuration database
- Instantiate the UVM environment
- Use the configuration database to pass the virtual interface handle(s) and other information down to the environment
- Instantiate and start any sequences necessary for the test case
- Manage phase objections, to ensure the test successfully completes

The test must be registered with the factory, and must implement the `build_phase()` and `run_phase()` methods. It may optionally implement other phase methods, although these are typically not required to implement an effective test. The following may be considered a template for a UVM test:

```
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  my_env m_env;
  my_env_config_obj m_env_cfg;

  function new(string name = "my_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    ...
  endfunction

  task run_phase(uvm_phase phase);
    ...
  endtask
endclass
```

Example 2 – A UVM test class template

The `uvm_component_utils` macro is used to register the test class with the factory so that the call to `run_test()` in the top-level module will be able to create an instance of this test and run it. An advantage of the factory registration macros provided in UVM is that you do not need to worry about the details of how to register classes in the UVM factory. It is only required that you call the macro as the first thing in your test, and include the test class name as the argument. Note that macro calls do not use a semicolon at the end of the line.

The class `new()` constructor is the same for all UVM components. The arguments to `new()` are default values that may (and probably will) be overridden when the component is instantiated by the factory. However, since the test is instantiated from the call to `run_test()`(which was shown in Example 1), its default values will be used. Therefore, we specify the name to be the name of the class (so it is easily identified) and we specify `null` as the parent (because the `uvm_test` is at the top of the UVM hierarchy).

2.3 Instantiate and Set up the Environment

The test uses the *build_phase* to instantiate and configure the environment.

```
function void build_phase(uvm_phase phase);
  m_env_cfg = my_env_config_obj::type_id::create("m_env_cfg");
  m_env = my_env::type_id::create("my_env", this);
  if(!uvm_config_db#(virtual my_dut_interface)::get(this, "" ,"DUT_IF",
                                                    m_env_cfg.dut_if))
    `uvm_error("TEST", "Failed to get virtual interface in test")
  // set other aspects of m_env_cfg
  uvm_config_db#(my_env_config_obj)::set(this, "my_env", "m_env_cfg", m_env_cfg);
endfunction
```

Example 3 – A UVM test build_phase

UVM objects and components are constructed using the factory `create()` method, rather than the conventional class `new()` constructor. The ***Test Writer*** only needs to know the usage syntax of this `create()` method. It is a static method call with a static path made of the typename of the class being constructed and a static `type_id` handle. The `create()` method for classes extended from the `uvm_object` base class requires one argument, which is a name string. The `create()` method for classes extended from the `uvm_component` base class requires two arguments, which are a name string and a handle to the component that will contain the instance of the component being created. This second argument will always be `this` – the component being created lives within the component calling `create()`.

The use of the configuration object, `my_env_config_obj`, is simply a convenience mechanism that allows all of the information for the environment to be encapsulated in one object, thus allowing a single `uvm_config_db` call to pass the information to the environment. The configuration object itself is simply an extension of `uvm_object`, so it is treated as any other `uvm_object` in that it gets created and registered with the factory via the `` `uvm_object_utils `` macro.

To maximize reuse, we recommend that each layer of the hierarchy in UVM should get configuration information from its parent and pass the information down to the next layer.

The environment itself is simply a UVM component, so it is created from the factory, just as any other component would be. It is often the case that basic operations like instantiating and configuring the environment are done in a "base test", which is then extended to allow additional customization for specific testing objectives. In this case, we may extend the base test and use the factory to instantiate a different environment.

```
class my_extended_test extends my_test;
  `uvm_component_utils(my_extended_test)
  ...

  function void build_phase(uvm_phase phase);
    my_env::type_id::set_type_override(my_env2::get_type());
    // optionally override type of my_env_cfg object
    super.build_phase(phase);
    // optionally make additional changes to my_env_cfg object
  endfunction
```

Example 4 – An extended UVM test that overrides the environment type

There are a few things to note about the extended test. We use the factory override in the `build_phase()` before calling `super.build_phase()`, so that the override will be in place before the environment is created (similarly with the configuration object, if we choose to override that as well). While type overrides are more common, the UVM factory also includes the ability to override a type for a specific instance of an object as well[2]. The second thing to notice is that, while we obviously call `super.build_phase()` from the extended test, we do not call `super.build_phase()` in the base test. The general rule in UVM is as follows:

Never call `super.build_phase()` in a component extended from a UVM library base class.

The reason for this rule is that the built-in UVM component base classes in the UVM class library include code in their *build_phase* methods for auto configuration, which is unnecessary, slow, and difficult to debug.

2.4 Starting Sequences

In addition to setting up the environment, the test is chiefly responsible for starting specific test sequences in UVM that will define the behaviors actually exercised. The sequences themselves will be written by the ***Sequence Writer***, but it is up to the ***Test Writer*** to start them.

```
class my_extended_test extends my_test;
  `uvm_component_utils(my_extended_test)
  ...
```

[2] `my_comp::type_id::set_inst_override(my_comp2::get_type(), "top.env.c2");`

```
  function void build_phase(uvm_phase phase);
    ...
  endfunction

  task run_phase(uvm_phase phase);
    ...
    my_seq seq = my_seq::type_id::create("seq");
    //optionally randomize sequence
    assert(seq.randomize() with {src_addr == 32'h0100_0800;
                                 xfer_size == 128;});
    seq.start(m_env.m_agent.m_sequencer);
    ...
  endtask
```

Example 5 – Starting a sequence in the UVM test run_phase

It is recommended that any particular extension of the base test should create and start a specific set of test sequences. In general, since the test extension is what gets called from the command line, there is no need to override the sequence type(s) started in the test. The only reason for the test to override a sequence type would be if there were a background sequence also running that is started from somewhere other than the test.

The UVM provides a facility called a "*default sequence*" that lets you use the `uvm_config_db` to specify the type of a sequence that a given sequencer will start in the desired phase to act as "background traffic." When the desired phase starts, the specified sequencer will automatically create, randomize, and start a sequence instance of the specified type. This mechanism is not recommended, because it does not provide enough control to the ***Test Writer***. Instead, the ***Environment Writer*** should provide a configuration hook that will allow the ***Test Writer*** to pass in an instance of the desired background sequence and have the environment start the sequence in the desired phase. The easiest way to do this is simply to make the background sequence a member of the environment's configuration object. Since all the environment does is get the sequence instance (if it exists) from the configuration object and start it, the test is free to pass in an extension of a base background sequence.

Example 6 shows how an extended test class can create a background sequence and copy the handle of the sequence object to a `background_seq` handle in the configuration object. Note that this code does not require the use of any additional UVM constructs beyond the simple set of constructs already discussed.

```
class my_extended_test extends my_test;
  `uvm_component_utils(my_extended_test)
  ...

  function void build_phase(uvm_phase phase);
    my_env::type_id::set_type_override(my_env2::get_type());
    // optionally override type of my_env_cfg object
    super.build_phase(phase);
    my_bkgrnd_seq bkgrnd_seq = my_bkgrnd_seq::type_id::create("bkgrnd_seq");
    assert(bkgrnd_seq.randomize() with {burst_len == 16;});
    my_env_cfg.background_seq = bkgrnd_seq;
  endfunction
```

Example 6 – Starting a background sequence in a UVM test build_phase

With this approach, the environment simply starts the sequence from the configuration object in the *run_phase*.

It is often the case that a test will start a *virtual sequence*, which is responsible for coordinating the execution of multiple other sequences. The details of how to do this are discussed in Section 4.0, ***Sequence Writer***. All the ***Test Writer*** needs to know is to how to initialize and start a virtual sequence.

```
class my_virtual_test extends my_test;
  `uvm_component_utils(my_extended_test)
  ...
  task run_phase(uvm_phase phase);
    ...
    my_vseq virt_seq = my_seq::type_id::create("virt_seq");
    virt_seq.init(.bus_seqr(m_env.m_agent1.m_sequencer),
                  .gpio_seqr(m_env.m_agent2.m_sequencer));
    virt_seq.start(null);
```

```
    ...
  endtask
```

Example 7 – Starting a virtual sequence in a UVM test run_phase

The virtual sequence is not started on a specific sequencer, since it doesn't actually create transactions. Rather, the virtual sequence has handles to the underlying sequencers on which its subsequences will run. While there are several alternatives on how to initialize the virtual sequence, the important point to consider is that the test will know which sequencers to use for the subsequences. There is no need to include a *virtual sequence**r*** in either the test or the environment. All the virtual sequencer does is add more inter-component connections and complexity to the environment, with no benefit of portability nor flexibility.

2.5 Phasing and Objections

In UVM, the *run_phase* is the only time-consuming phase of execution. In parallel with the *run_phase*, UVM includes several other phases (*reset_phase*, *configure_phase*, *main_phase*, *shutdown_phase*, and *pre_*/*post_* variants thereof) that may be used to subdivide the execution of the test case, but these phases add unnecessary complexity[2]. All execution of all components, including the test and environment, should be handled by *run_phase*. In order to ensure that all of your desired transactions execute in your test case, you must tell UVM not to exit the *run_phase* until your desired stimuli complete. This is done using *objections*.

```
class my_extended_test extends my_test;
  `uvm_component_utils(my_extended_test)
  ...
  task run_phase(uvm_phase phase);
    phase.raise_objection("Starting test");
    my_seq seq = my_seq::type_id::create("seq");
    //optionally randomize sequence
    assert(seq.randomize() with {src_addr == 32'h0100_0800;
                                 xfer_size == 128;});
    seq.start(m_env.m_agent.m_sequencer);
    phase.drop_objection("Ending test");
  endtask
```

Example 8 – Using objection flags in a UVM test run_phase

The `raise_objection()` call must be made before the first nonblocking assignment is made in that phase. The phase method will continue until all raised objections are dropped. Since the `seq.start()` call is blocking, it will return when the stimulus sequence has completed sending its transactions. Dropping the objection upon completion of the sequence is usually sufficient to allow the *run_phase* to complete correctly. However, it is possible that some transactor(s) may need to delay the end of the phase to complete processing the last transaction. In this case, the ***Environment Writer*** will need to implement the `phase_ready_to_end()` method, as shown in Section 5.1.

3.0 Environment Writer

The ***Environment Writer*** is responsible for getting stimulus generated by the test into the Device Under Test (DUT), and verifying that the DUT responds correctly to that stimulus. To accomplish this, the ***Environment Writer*** defines a UVM *environment*, *agent* and *scoreboard*.

3.1 UVM Environments

A UVM *environment* is a UVM component that is created and configured by a UVM test, as has already been discussed. The environment drives stimulus into the DUT, monitors the inputs and outputs of the DUT, and verifies that the DUT behaves as intended.

A UVM environment encapsulates the structural aspects of a UVM testbench. A UVM environment contains:

- One or more agents, each of which handles driving DUT inputs and monitoring DUT input and output activity on a specific DUT interface.
- A scoreboard, which handles verifying DUT responses to stimulus
- Optionally, a coverage collector, which records transaction information for coverage analysis

- A configuration component, which allows the test to set up the environment and agent for specific test requirements.

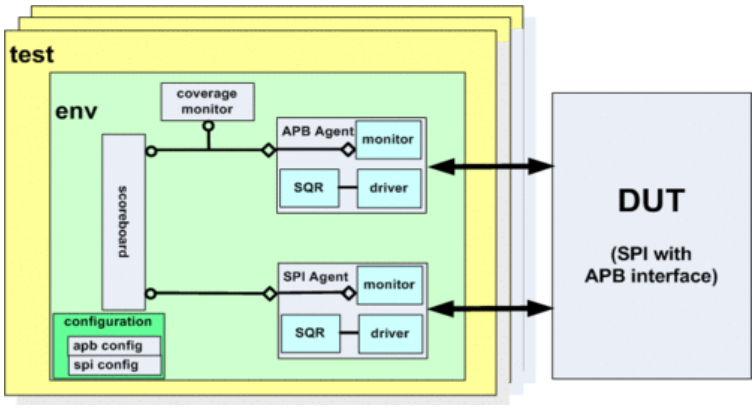

Figure 1 – A basic UVM Environment

A more complex UVM environment, such as shown in Figure 1, might contain additional components, such as multiple agents and a register model. This is discussed later in this paper in the Section 5.0, Advanced Examples. *Spoiler Alert!* We will see in these more complex environments that the ***Environment Writer*** needs to know very few more UVM constructs than the simple set of constructs used in a basic UVM environment.

Example 9 illustrates the code for a simple UVM environment with a single agent and scoreboard.

```
class my_env extends uvm_env;

  `uvm_component_utils(my_env)

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

  my_agent  agent;
  my_scoreboard scoreboard;

  function void build_phase(uvm_phase phase);
    agent = my_agent::type_id::create("agent", this);
    scoreboard = my_scoreboard::type_id::create("scoreboard", this);
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    agent.dut_inputs_port.connect(scoreboard.dut_in_imp_export);
    agent.dut_outputs_port.connect(scoreboard.dut_out_imp_export);
  endfunction: connect_phase

endclass: my_env
```

Example 9 – UVM environment

3.1.1 The Environment Declaration

The *environment* is a user-defined class, called `my_env` in this example. An environment class is extended from the `uvm_env` base class, which is derived from the `uvm_component` base class. UVM components are objects that form the hierarchy of testbench, and are objects that are generally created at the beginning of a simulation and retained throughout the simulation.

At the top of the class is boilerplate code that is required for all classes extended from `uvm_component`. This is the same general code that was discussed in Section 2.3 of this paper, and includes registering the class name with the factory by using the macro `` `uvm_component_utils() `` and defining the class `new()` constructor. As

mentioned above, there is no benefit in using the auto-configuration and field-automation features of UVM, so there is no reason to use the `uvm_component_utils_begin/end macros to set up the field automation.

3.1.2 The Environment Build Phase

The UVM ***build_phase*** is used to construct the components contained in the environment. In this simple example, these are an `agent` and a `scoreboard`. These components are constructed using the factory `create()` method, discussed earlier, in Section 2.3. The details of agents and scoreboards are discussed later in this paper.

3.1.3 The Environment Connect Phase

The UVM ***connect_phase*** is used to establish connections between the agent and the scoreboard. UVM uses the Transaction Level Modeling (TLM) 1.0 communication protocol to allow UVM components to pass information from one class object to another object. TLM 1.0 originates from the SystemC standard. With TLM 1.0, there are two primary communication protocols:

- A ***port***/***export*** pair – a 1-to-1 connection used to push or pull a `uvm_sequence_item` object handle from one component to another. UVM sequencers and drivers most often use this 1-to-1 protocol.
- An ***analysis port***/***imp export*** pair – a 1-to-many connection, used to broadcast a `uvm_sequence_item` object handle to zero or more destinations. UVM monitors typically use this protocol to allow the agent to send information to both a scoreboard and a coverage collector.

TLM ports and exports are classes defined in the UVM base class library. Each port and each export is an object (an instance) of one of these classes. The name of a port or export is actually the handle to that port or export object.

Each port or analysis port has a `connect()` method. The argument to this connect method is the corresponding export or imp export to which the port is paired.

At the environment class level, this `connect()` method is the only UVM construct needed to complete the environment. In Example 9, the agent contains two analysis ports, which are being connected to two imp exports on the scoreboard. (The ***Environment Writer*** will need to know additional TLM methods, in order to write the driver and monitor, which will be shown in the discussion on those topics below.)

3.2 UVM Agents

A UVM agent is a low-level building block that is associated with a specific set of DUT I/O pins and the communication protocol for those pins. For example, the USB bus to a DUT will have an agent for that set of ports. Likewise, the AXI bus would have an agent specific for that bus. An *agent* contains three required components: a *sequencer*, *driver* and *monitor*. In addition, agents may contain an optional *coverage collector* component.

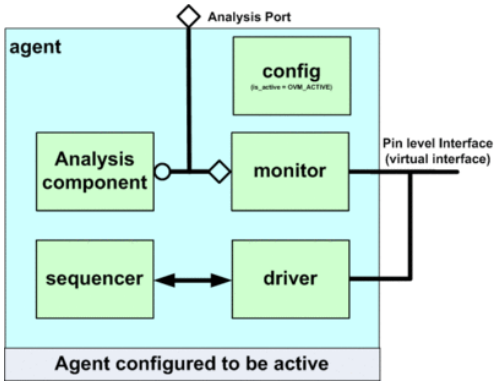

Figure 2 – A typical UVM Agent

Example 10 contains the code for our simple agent.

```
class my_agent extends uvm_agent;

  `uvm_component_utils(my_agent)  // register this class in the factory
```

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

  // handles for agent's components
  my_sequencer          sqr;
  my_driver             drv;
  my_monitor            mon;
  my_coverage_collector cov;
  my_config             m_config;

  // configuration knobs
  localparam OFF=1'b0, ON=1'b1;
  bit enable_coverage = OFF;                       // default of disabled
  uvm_active_passive_enum is_active = UVM_ACTIVE; // default of active

  // handles to the monitor's analysis ports
  uvm_analysis_port #(my_tx) dut_inputs_port;
  uvm_analysis_port #(my_tx) dut_outputs_port;

  function void build_phase(uvm_phase phase);
    ...
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    ...
  endfunction: connect_phase

endclass: my_agent
```

Example 10 – A simple UVM agent

3.2.1 The Agent Declaration

The *agent* is a user-defined class, which we called `my_agent` in this example, and which is extended from the `uvm_agent` base class. The class name is registered with the factory, using the macro `` `uvm_component_utils() ``, and the class `new()` constructor is defined, with its `name` and `parent` inputs.

3.2.2 Agent Configuration Knobs

Agents need to be configurable to meet the requirements for a specific test. The controls for configuring UVM components are often referred to as *"knobs"*. These knobs might have simple on/off values (typically a 1-bit 0 is used to represent OFF, and a 1-bit 1 to represent ON), or the knobs might be set to a value, such as the number of transactions a sequence should generate.

The `enable_coverage` knob is used to configure the agent to either collect or not collect functional coverage data for the agent. The knob has a default value of OFF (disabled).

The `is_active` knob is used to configure the agent to be active or passive. An *active agent* can both monitor and drive DUT ports. A *passive agent* can only monitor DUT ports. The data type of `is_active` is an enumerated type defined in the UVM base class library called `uvm_active_passive_enum`, which has two possible enumerated values, `UVM_ACTIVE` and `UVM_PASSIVE`. The default value for this knob is for the agent to be active.

There is an `is_active` knob already defined in the `uvm_agent` base class, but this property is not documented in the UVM Class Reference Manual as one of the variables in this base class. Instead, the Class Reference Manual documents a `get_is_active()` method that returns the value of the base class `is_active` property. However, the Class Reference Manual does not provide a method for setting this knob. Since the base class `is_active` knob is not officially documented, and there is no method for setting the variable, the authors recommend explicitly defining a local `is_active` knob, rather than using the undocumented `is_active` base class property. Having a local `is_active` property also allows defining an explicit default value. There is no functional reason for an agent to be extended from the `uvm_agent` base class. An agent can also be extended from the `uvm_component` base class.

3.2.3 The Agent Port Handles

The environment needs to connect the agent to a scoreboard, in order for the scoreboard to receive information about the DUT input and output values captured by the monitor. This is done by declaring UVM analysis ports at the agent level that correspond to the same ports on the monitor. One port will pass handles to sequence_item objects that contain the values of the DUT input ports being used by this agent. The other port will pass handles to sequence_item objects that contain the values of the DUT output ports used by this agent. UVM analysis ports are instances of a base class called `uvm_analysis_port`. The `uvm_agent` base class contains a type parameter, which defines the sequence_item class type that will be communicated through the port. In this example, both ports pass handles to a `my_tx` sequence_item type. The agent port names, `dut_inputs_port` and `dut_outputs_port`, are object handles, which will be assigned values in the *connect_phase* of the agent.

3.2.4 The Agent Build Phase

```
class my_agent extends uvm_agent;
  ...
  function void build_phase(uvm_phase phase);
    if (uvm_config_db #(my_config)::get(this, "", "test_config", m_config))
      begin // get() succeeded
        this.is_active = this.m_config.is_active;
        this.enable_coverage = this.m_config.enable_coverage;
      end
    else `uvm_warning("LAB", Failed to access config_db -- using defaults instead.\n")
    mon = my_monitor::type_id::create("mon", this);
    if (is_active == UVM_ACTIVE) begin
      sqr = my_sequencer::type_id::create("sqr", this);
      drv = my_driver::type_id::create("drv", this);
    end
    if (enable_coverage)
      cov = my_coverage_collector::type_id::create("cov", this);
  endfunction: build_phase
```

Example 11 – A simple UVM agent build_phase

The UVM ***build_phase*** is used to construct the components contained in the agent. In this simple example, these are: `my_monitor`, `driver`, `my_sequencer`, `my_coverage_collector`, and a `my_config`. These components are constructed using the factory `create()` method, discussed earlier. The details of these components are discussed later in this paper.

Before constructing the agent's components, the *build_phase* first attempts to retrieve a configuration object from the configuration data base by using the `uvm_config_db:get()` static method. Configuration objects provide a way for the ***Test Writer*** to configure environments and agents to meet the requirements of a specific test, and were discussed in Section 2.3 earlier in this paper.

In Example 11 above, the configuration object handle is retrieved from the data base and stored into the `m_config` property. The configuration "knobs" used by the agent are the `is_active` and `enable_coverage` configuration variables, which are set by the ***Test Writer*** as part of the test class. These knobs are used to determine whether the driver, sequencer, and coverage collector components within the agent are to be constructed. If the test configures the agent as passive, then there is no driver/sequencer pair. If the test configures the agent to not collect coverage, then there is no coverage collector component.

3.2.5 The Agent Connect Phase

```
class my_agent extends uvm_agent;
  ...
  function void connect_phase(uvm_phase phase);
    // set agent's analysis ports to point to the monitor's ports
    dut_inputs_port  = mon.dut_inputs_port;
    dut_outputs_port = mon.dut_outputs_port;
    if (is_active == UVM_ACTIVE) begin
      drv.seq_item_port.connect(sqr.seq_item_export); // connect driver to sequencer
    end
    if (enable_coverage)
      mon.dut_inputs_port.connect(cov.analysis_export);
```

```
  endfunction: connect_phase
```

Example 12 – A simple UVM agent connect_phase

The ***connect_phase*** of the agent is used for three purposes: to connect the sequencer and driver, to connect the coverage collector and monitor, and to set the agent's analysis ports to map to the monitor's ports.

If the agent is configured as active, the driver's port is connected to the sequencer's export. These ports and exports are defined in the `uvm_driver` and `uvm_sequencer` base classes, with the names `seq_item_port` and `seq_item_export`, respectively. (These port names are found in the UVM Class Reference Manual.)

Second, if the agent is configured to collect coverage, the monitor's port is connected to the coverage collector's export. The monitor's port is a user-defined port name, which is `dut_inputs_port` in this example. The coverage collector's export is defined in the `uvm_subscriber` base class, with the name `analysis_export` (documented in the UVM Class Reference Manual).

Third, the handles of the monitor's ports are copied to the corresponding port handles of the agent. The names of the port handles are user-defined. In this example, the monitor and the agent use the same port names of `dut_inputs_port` and `dut_outputs_port`.

There is an alternate coding style for connecting the agent's ports to the monitor's ports. UVM provides a TLM pass-through port type called an *analysis export.* An *analysis export* , also referred to as a *hierarchical export*, is connected to an *analysis port* on the monitor (which writes out sequence_item handles) and to an *analysis imp export* on the scoreboard (which receives the sequence_item handles). The analysis export passes the sequence_item handles from the port to the imp export. At the agent level, however, these pass-through analysis exports are not needed. Since the monitor's ports are simply object handles, it is a simpler coding style for the agent to just have a copy of the monitor's port handles. This is the style described in the preceding paragraph and used in Example 10.

3.3 UVM Sequencers

A sequencer serves as a router of sequence_items (transactions). The sequencer can receive sequence_items from any number of sequences (stimulus generators) and route these items to the agent's driver. Sequencers are extended from the `uvm_sequencer` base class, and inherit all necessary routing and arbitration functionality from this base class. The `uvm_sequencer` base class contains a type parameter that defines what type of sequence_item class the sequencer can route. This parameter must be defined, in order to specialize the sequencer to match the driver to which it will be connected.

Since the `uvm_sequencer` base class functionality does not need to be extended, it is possible to use the base class directly within an agent, by simply defining the sequencer's type parameter to a specific sequence_item type.

```
typedef uvm_sequencer #(my_tx) my_sequencer;
```

Example 13 – Defining a sequencer using typedef

In this example, the `my_sequencer` user-defined type represents a `uvm_sequencer` class that has been specialized to work with `my_tx` sequence_item types.

It is also possible to define a user-defined class name that extends `uvm_sequencer` and specializes its type parameter. The class must contain the standard boilerplate code that registers the class name with the factory and defines a `new()` constructor. The only reason for doing this would be if the arbitration algorithms in the `uvm_sequencer` base class do not meet the verification requirements, and the sequencer needs to be extended with a user-defined arbitration algorithm. The majority of verification tasks will never need to do this.

3.4 UVM Drivers

A UVM driver requests a handle to a sequence_item object from its associated sequencer, and assigns values in the sequence_item properties to corresponding signals in a SystemVerilog interface, thus driving the DUT inputs.

Example 14 illustrates the UVM driver for the simple example used in this paper.

```
class my_driver extends uvm_driver #(my_tx);

  `uvm_component_utils(my_driver)
```

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

  virtual tb_if tb_vif;  // virtual interface pointer

  function void build_phase(uvm_phase phase);
    if (!uvm_config_db #(virtual my_dut_interface)::get(this, "", "DUT_IF", tb_vif))
      `uvm_fatal("NOVIF", Failed to get virtual interface from uvm_config_db.\n")
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    my_tx tx;
    forever begin
      @tb_vif.clk  // drive values synchronized to interface clock
      seq_item_port.get_next_item(tx);  // get a transaction
        tb_vif.operand_a <= tx.operand_a;
        tb_vif.operand_b <= tx.operand_b;
        tb_vif.opcode    <= tx.opcode;
      seq_item_port.item_done();
    end
  endtask: run_phase
endclass: my_driver
```

Example 14 – A simple UVM driver

3.4.1 The Driver Declaration

A UVM driver is extended from the `uvm_driver` base class. This base class contains a type parameter that defines what type of sequence_item the driver will receive from its associated sequencer. When `uvm_driver` is extended, this parameter must be defined, in order to specialize the driver to this sequence_item type.

The class name is registered with the factory using the `` `uvm_component_utils() `` macro, and the class `new()` constructor is defined, with its required `name` and `parent` inputs.

3.4.2 Driver/Sequencer Communication

The `uvm_driver` base class has a TLM port called `seq_item_port`, which is inherited by the `my_driver` derived class. As shown previously in Example 10, the agent connects this port to the sequencer's export. This port/export pair is used to pass a handle to a sequence_item object, which was created by the sequence class, and routed through the sequencer.

3.4.3 Driver/Interface Communication

The driver sends values to the DUT by assigning values to variables in a SystemVerilog interface, which the top-level module instantiated and connected to the DUT. This interface is specific to the agent, and encapsulates the DUT signals required for a specific bus protocol used by the DUT. If, for example, the DUT had a USB bus and an AXI bus, there would be a separate interface and corresponding agent (with its driver) for each bus.

In order for a class definition to assign values to interface variables, the class needs to have a pointer to the interface instance. This pointer is referred to as a *virtual interface*. The top-level module discussed earlier in this paper (see Section 2.1) instantiates the interface and copies the virtual interface into UVM's configuration data base. The driver retrieves the virtual interface using the static `uvm_config_db::get()` method. This method was discussed previously, in Section 2.3.

3.4.4 The Driver's Run Phase

The real work of a driver takes place in the UVM ***run_phase***. Typically, a UVM driver is written as an infinite loop, which can be done using the SystemVerilog `forever` loop or a `while(1)` loop. Although the driver is coded as an infinite loop, the *run_phase* itself is not left to run for all of infinity. The ***Test Writer*** uses UVM's objection flags to control how long the *run_phase* actually runs, as discussed early in this paper.

Within the driver's *run_phase*, the driver will synchronize with the DUT clocks and with the UVM sequence that is generating stimulus values. This double synchronization is discussed in the following paragraphs.

3.4.5 Driver/DUT Synchronization and Race Avoidance

In order to coordinate the transfer of values from the driver to the interface variables, the driver synchronizes with DUT using a clock signal available in the interface. This is the same clock used by the DUT for the specific bus protocol handled by the UVM agent.

In all digital verification, regardless of the verification methodology employed, the testbench must avoid race conditions with the DUT. Simply stated, the testbench must drive stimulus early enough before the DUT's clock edge so that the stimulus is stable when the DUT registers trigger and store the input values (setup time). The SystemVerilog methodology expects – but does not mandate – that this synchronization between the testbench and the DUT be handled by the interface that sits between the UVM driver and the DUT. Moving this synchronization to the interface has important advantages:

- The driver code is significantly simplified – all the driver needs to do is assign values to the interface signals, and leave it up to the interface to avoid race conditions with the DUT.
- Variations of a driver can be extended from the user-defined driver class, without having to duplicate complex synchronization timing.

The monitor will also need to avoid race conditions with the DUT, and only observe DUT outputs at times in which those outputs are stable.

Note that the driver and monitor may be written to access signals in the virtual interface directly, or they may be written to call tasks in the interface to further simplify the testbench and leave the details to the interface writer [1]. A SystemVerilog *clocking block* may be used to help with race avoidance, but this construct is generally incompatible with emulation, which should be considered when designing drivers and monitors. Having these components call interface tasks allows the choice of the synchronization mechanism to be deferred to the interface as well.

3.4.6 Driver/Sequence Synchronization

The values that a UVM driver will drive into the DUT are stored in a UVM sequence_item object. This object is created, and its values generated, by a UVM sequence. The driver needs to synchronize with the sequence for when the driver obtains sequence_item handles. This synchronization is done with a pair of methods, `get_next_item()` and `item_done()`. These methods are defined as part of the driver's TLM port, which is connected to a corresponding sequencer. All that the ***Environment Writer*** needs to know is that calling `get_next_item()` will block the execution of the driver and wait for a sequence to create a sequence_item object. The complex code to actually handshake with one or more sequence stimulus generators is handled completely within the UVM base class methods used by the driver and sequence. Figure 3 illustrates the basic handshaking that takes place between the driver and the sequence.

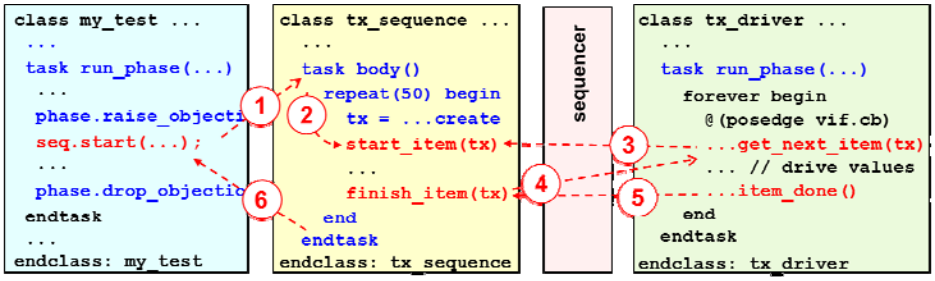

Figure 3 – Test/sequence/driver handshaking

The handshaking steps that synchronize the driver and sequence are:

1. The test class (from the ***Test Writer***) raises its objection flag and calls a sequence's `start()` method, which invokes the sequence `body()` method. The `start()` method blocks (waits at that point) until the `body()` method exits.
2. The sequence `body()` method calls a `start_item()` method. `start_item()` blocks (waits) until the driver asks for a transaction (a sequence_item object handle).

3. The driver calls the `seq_item_port.get_next_item()` method to request (pull) a transaction. The driver then blocks (waits) until a transaction is received.
4. The sequence generates the transaction values and calls `finish_item()`, which sends the transaction to the driver. The sequence then blocks (waits) until the driver is finished with that transaction.
5. The driver assigns the transaction values to the interface variables, and then calls the `seq_item_port.item_done()` method to unblock the sequence. The sequence can then repeat steps 2 through 5 to generate additional stimulus.
6. After the sequence has completed generating stimulus, the sequence `body()` exits, which unblocks the test's `start()` method. The test will then continue with its next statements, which includes dropping its objection flag and allowing the *run_phase* to end.

The beauty of UVM is that all the ***Test Writer*** needs to know is the proper usage of the `start()` method and objection flags. All the ***Sequence Writer*** needs to know is the proper usage of the `start_item()` and `finish_item()` methods, and all the ***Environment Writer*** needs to know is the proper usage of the `get_next_item()` and `item_done()` methods. Each Writer needs to know very little about the internals of what the other Writers need to code, or how the UVM base classes handle the complexities of synchronization between the classes.

3.5 UVM Monitors

A UVM monitor observes the DUT inputs and outputs for a specific interface, captures the observed values into one or more sequence_items, and broadcasts handles to those sequence_items to other UVM components (such as a scoreboard and a coverage collector). UVM is strict about what a monitor should do, but allows ample flexibility on how that functionality should be implemented – perhaps too much flexibility, which has led to a wide variety of monitor implementation styles and opinions on how to best implement monitors. The only UVM rules for monitors are: 1) Monitors are part of an agent, and therefore closely coupled with the specific bus protocol handled by that agent. 2) Monitors are strictly passive components – a monitor is only permitted to observe activity, and can never change values. 3) Monitors observe both DUT inputs and outputs. Even though the driver knows what values are being driven into the DUT, UVM requires that only the monitor communicate these input values to the scoreboard or other components. One reason for this requirement is that agents can be configured to be passive, and passive agents have a monitor, but no driver or sequencer. Example 15 shows the code for a basic UVM monitor.

```
class my_monitor extends uvm_monitor;

  `uvm_component_utils(my_monitor)

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

  virtual my_dut_interface tb_vif;  // virtual interface

  uvm_analysis_port #(my_tx) dut_inputs_port;  // analysis port for DUT inputs
  uvm_analysis_port #(my_tx) dut_outputs_port; // analysis port for DUT outputs

  function void build_phase(uvm_phase phase);
    dut_inputs_port  = new("dut_inputs_port", this);  // construct the analysis port
    dut_outputs_port = new("dut_outputs_port", this); // construct the analysis port
    if (!uvm_config_db #(virtual my_dut_interface)::get(this, "", "DUT_IF", tb_vif))
      `uvm_fatal("NOVIF", Failed to get virtual interface from uvm_config_db.\n")
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    my_tx tx_in, my_tx tx_out, tx_copy;
    fork
      // monitor DUT inputs synchronous to the interface clock
      forever @(tb_vif.clk) begin
        // create a new tx_write object for this cycle
        tx_in  = my_tx::type_id::create("tx_in");
        tx_in.operand_a = tb_vif.operand_a;
```

```
        tx_in.operand_b = tb_vif.operand_b;
        tx_in.opcode    = tb_vif.opcode;
        dut_inputs_port.write(tx_in);
      end

      // monitor DUT outputs synchronous to the cb2 clocking block
      // create a tx_tmp object to reuse each pass of the loop
      tx_out = my_tx::type_id::create("tx_out"); // tx_out is reused each loop pass
      forever @(tb_vif.clk2) begin
        tx_out.result    = tb_vif.result;
        tx_out.exception = tb_vif.exception;
        // create a copy of tx_out to send to the scoreboard
        $cast(tx_copy, tx_out.clone());
        dut_outputs_port.write(tx_copy);
      end
    join
  endtask: run_phase
endclass: my_monitor
```

Example 15 – A simple UVM monitor

3.5.1 The Monitor Declaration

A UVM monitor is extended from the `uvm_monitor` base class. This base class is not parameterized, so there is no parameter redefinition. As with all UVM testbench components, the class name is registered with the factory using the `` `uvm_component_utils() `` macro, and the class `new()` constructor is defined, with its required `name` and `parent` inputs.

3.5.2 Monitor Communication

UVM monitors use TLM ports to communicate with scoreboards and coverage collectors. TLM ports pass handles to sequence_items. The UVM methodology allows the ***Environment Writer*** to decide what types of sequence_items will be communicated through the TLM ports. The most common styles are:

- The same sequence_item type is used to capture DUT input values and DUT output values. Although the same sequence_item type is used, inputs and outputs are captured into different objects, since these values are captured at different simulation times. This is the style shown in Example 15.
- Different sequence_item types are used to capture DUT inputs and DUT outputs. For example there might be a `my_tx_in` sequence_item class that only has variables for storing the DUT inputs, and a `my_tx_out` sequence_item class that only has variables for storing the DUT outputs for a specific interface.

UVM also allows the ***Environment Writer*** to determine how many communication ports the monitor needs, in order to pass information to the scoreboard and coverage collector. A common coding style is to use two ports, one for the sequence_item object handles containing the DUT inputs and a different port for the sequence_item object handles containing the DUT outputs. The monitor shown in Example 15 uses two ports. Both ports communicate handles to a `my_tx` sequence item type, which has variables to capture both DUT input values and DUT output values. Depending on the protocol, there may alternatively be request and response item types, or whatever set of items may be appropriate.

These ports are TLM analysis ports, which permit a one-to-many connection. This allows the handle to the sequence_item object containing the DUT input values to be passed to both the scoreboard and a coverage collector. The `uvm_analysis_port` is a parameterized class, which must be specialized to work with a specific sequence_item class type (which is `my_tx`, in this example).

The ports are constructed in the *build_phase* of the monitor. Note that ports are constructed using the class `new()` constructor, instead of the factory. The `uvm_analysis_port` base class name is not registered with the factory, and therefore cannot be constructed by the factory. Nor is there a reason to do so. The purpose of the factory is to allow a ***Test Writer*** to substitute one class type for another (for example, to replace all USB 2.0 sequence_items with USB 3.0 sequence_items). It would not make sense for a test to change what type of ports a monitor uses, and therefore the factory is not needed for constructing ports.

3.5.3 Monitor/Interface Communication

The monitor observes the DUT input and output values by reading the variables in the interface that the top-level module instantiated and connected to the DUT. The monitor retrieves the virtual interface that points to the interface using the static `uvm_config_db::get()` method, in exactly the same way a driver retrieves the virtual interface.

3.5.4 The Monitor's Run Phase

The actual monitoring of DUT input and output values takes place in the UVM ***run_phase***. Typically, inputs and outputs are captured at different simulation times, and possibly on different clocks. To synchronize to different times and clocks, the *run_phase* forks off two infinite loops, which run in parallel with each other. Like the driver, the monitor is coded with infinite loops, so that monitoring will continue as long as the *run_phase* is running. The ***Test Writer*** controls how long the *run_phase* actually executes by using UVM's objection flags.

For each cycle of these loops, the monitor creates a sequence_item object, captures DUT values into that object, and then broadcasts the handle of the object out the analysis port by using a `write()` method. The actual functionality of this `write()` method will be implemented in each analysis imp export to which the analysis port is connected. We will see this implementation of the `write()` method in the code for the coverage collector (Section 3.6) and scoreboard (Section 3.7) components.

It is important that the monitor create a new sequence_item object for each pass of the monitor loop. The scoreboard will receive a handle to the sequence_item object. Problems will arise if the scoreboard saves this handle for evaluation at some future time and the monitor did not create a new object for each pass of the loop. Each pass will capture new DUT values. If these new values were saved in the same object that was used in the previous pass, the scoreboard will end up saving multiple object handles that all reference the same object, which is only storing the last set of values captured. There are two coding styles that can be used to create new sequence_item objects in each pass of the loop: using the factory `create()` method at the beginning of each pass, or using the sequence_item `clone()` method at the end of each loop. For illustration purposes, the code in Example 15 shows each of these styles, although most ***Environment Writers*** would probably use one style or the other.

3.6 UVM Coverage Collectors

Functional coverage is a vital aspect of verifying complex designs, especially when using constrained random verification. Functional coverage is an integral part of the SystemVerilog language, and includes `covergroup`, `coverpoint`, and `bin`s definitions, and a `sample()` method. UVM does not dictate where the coverage constructs should be used. They are simply SystemVerilog constructs that can be used in any SystemVerilog code. Though not mandated by the methodology, the UVM agent is an ideal location for collecting functional coverage information in a UVM testbench. The agent's monitor is observing the DUT inputs and outputs for a specific bus protocol, and coverage can be collected on the values observed for that bus.

UVM also does not specify where in the agent coverage should be collected. Some engineers prefer to make coverage an integral part of the monitor. This is a simple approach, and does not involve the use of TLM ports to communicate values to the coverage collection code. The authors, however, recommend defining a separate coverage collector component that can be instantiated in the agent. Using a separate component requires a little more coding (though not much more), but has some advantages. A coverage collector component encapsulates the coverage definitions and functionality for the agent in one place, instead of having the coverage code fragmented within the monitor. Also, a separate component can be constructed by the factory, which makes it easier to use the factory to customize coverage to meet the requirements of specific tests.

A simple coverage collector component is shown in Example 16.

```
class my_coverage_collector extends uvm_subscriber #(my_tx);

  my_tx tx;  // the transaction object on which value changes will be covered

  covergroup dut_inputs;
    option.per_instance = 1; // track coverage for each instance
    Opc: coverpoint tx.opcode;
    Opa: coverpoint tx.operand_a;
    Opb: coverpoint tx.operand_b;
  endgroup
```

```
  `uvm_component_utils(my_coverage_collector)

  function new(string name, uvm_component parent );
    super.new(name, parent);
    dut_inputs = new();  // construct the covergroup
  endfunction: new

  function void write(my_tx t);
    tx = t;  // copy transaction handle received from the monitor
    dut_inputs.sample();
  endfunction: write

  function void report_phase(uvm_phase phase);
    `uvm_info("DEBUG", $sformatf("\n\n Coverage for instance %s = %2.2f%%\n\n",
              this.get_full_name(), this.dut_inputs.get_inst_coverage()), UVM_HIGH)
  endfunction: report_phase
endclass: my_coverage_collector
```

Example 16 – A simple coverage collector

3.6.1 The Coverage Collector Declaration

A UVM coverage collector can be extended from the `uvm_subscriber` base class. This base class is a generic component type that can be used for a variety of purposes. The `uvm_subscriber` base class has a built-in TLM analysis imp export, that can be connected to the monitor's analysis port. The `uvm_subscriber` base class contains a type parameter that defines what type of sequence_item class the component will receive through the analysis imp export (`my_tx` in this example).

As with the previous examples, the coverage collector class name is registered with the factory by using the `` `uvm_component_utils() `` macro, and the class `new()` constructor is defined. However, the `new()` constructor for this coverage collector is a little different from other components we have defined. A requirement of the SystemVerilog language is that cover groups defined in a class can only be constructed in the class' `new()` constructor. In this example, the `dut_inputs` cover group is constructed.

3.6.2 The Coverage Collector write() Method

The analysis port in the monitor calls a `write()` method to broadcast out a handle to a sequence_item. Each analysis imp export connected to the port must implement this `write()` method. The implementation in this coverage collector is to copy the sequence_item handle passed from the monitor to the coverage collector's `tx` handle (the cover group's cover points are on variables within `tx`). The coverage `sample()` method is then called to actually collect coverage information.

3.6.3 The Coverage Collector Report Phase

For debug purposes, this coverage collector example reports the coverage percentage achieved for this specific collector. This is reported in UVM's ***report_phase***. The informational message generated by the `` `uvm_info `` macro uses a `UVM_HIGH` verbosity filter, which means the debug message will only be reported if simulation is run in a highly verbose mode. In a normal simulation, this message would not appear, and coverage statistics would be analyzed using the simulator's coverage reporting tools.

3.7 UVM Scoreboards

The primary role of a scoreboard in UVM is to verify that actual DUT outputs match predicted output values. UVM dictates that this verification functionality is part of the UVM environment, but that is where the methodology ends. How expected results are predicted and DUT outputs verified is SystemVerilog programming, and left up to the creativity of the ***Environment Writer***. This flexibility allows UVM to be truly universal in its ability to handle any size and type of design, but it also has led to a great deal of variance in recommendations from UVM "experts". This paper puts forth some guidelines for UVM scoreboards that follow the well-known KISS principle – "Keep It Simple, Stupid".

The KISS guideline is that the scoreboard functionality should be encapsulated into a UVM component that is instantiated at the environment level. The environment provides connectivity to one or more agents, which gives the scoreboard component access to the DUT input and output values observed by these agents. The ***Test Writer*** can configure the environment, which includes configuring the scoreboard component to meet the needs of a specific test. When the scoreboard functionality is encapsulated into a UVM component, the scoreboard can be more easily reused, and can be extended to add test-specific functionality. The UVM factory can swap the standard scoreboard component with extended versions. This KISS guideline also allows for verifying more complex designs that utilize multiple environments to verify different portions of a design.

The functionality of a relatively simple scoreboard can be coded directly in the scoreboard component (a flat hierarchy). The flattened scoreboard component will typically contain a method to predict expected results and a method to compare the predicted results to the actual DUT outputs. The scoreboard component can also have hierarchy within it. The prediction algorithms and verification algorithms can be defined in classes that are instantiated with the scoreboard. These sub blocks can also be configurable, and can be registered with the factory in order to enable factory creation and override capabilities.

Example 17 shows simple scoreboard component with a flat hierarchy for a single agent.

```
`uvm_analysis_imp_decl(_verify_outputs)

class my_scoreboard extends uvm_subscriber #(my_tx);

  `uvm_component_utils(my_scoreboard)

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

  uvm_analysis_imp_verify_outputs #(my_tx, my_scoreboard) dut_out_imp_export;

  mailbox #(my_tx) expected_fifo = new;  // SystemVerilog mailbox for my_tx handles

  int num_passed, num_failed;  // score cards

  function void build_phase(uvm_phase phase);
    dut_out_imp_export  = new("dut_out_imp_export", this);
  endfunction

  // implement the write() method called by the monitor for observed DUT inputs;
  // predict what the DUT results should be for a set of input values
  function void write (my_tx t);
    ...
  endfunction: write

  // implement the write() method called by the monitor for actual DUT outputs;
  // compare the DUT outputs to the predicted results
  function void write_verify_outputs(my_tx t);
    ...
  endfunction: write_verify_outputs

  function void report_phase(uvm_phase phase);
    ...
  endfunction: report_phase

endclass: my_scoreboard
```

Example 17 – A simple UVM scoreboard component (flat hierarchy)

This small example uses a flattened hierarchy – both the prediction and verification algorithms are coded directly in the scoreboard class. Later sections of this paper will discuss more complex verification challenges. (*Spoiler Alert:* We will see that the added verification complexity will not require adding to the complexity of UVM constructs required for effective scoreboarding. The KISS principle for writing scoreboards also minimizes the UVM constructs that are required for effective verification!)

3.7.1 The Scoreboard Declaration

UVM provides a `uvm_scoreboard` base class, but there is no requirement to use this base class to write a scoreboard component. In this example, the `uvm_subscriber` base class is used instead. The reason to use this base class is that it has a built-in TLM analysis imp export, which can be connected to the monitor's analysis port. The `uvm_subscriber` type parameter is specialized to the `my_tx` sequence_item type.

3.7.2 Scoreboard/Monitor Connections

The monitor example shown earlier in Example 15 has two *analysis ports*, one for monitoring DUT input values and one for monitoring DUT output values. The scoreboard needs two corresponding *analysis imp exports*. The `uvm_subscriber` base class provides one of these analysis imp exports. The user-defined code that extends this base class must add the second imp export.

Each of these analysis imp exports must implement the `write()` method for its corresponding port. ***This is a problem!*** The SystemVerilog language does not have "function overloading", and does not allow two methods with the same name to exist in a class, but our scoreboard needs to implement two `write()` methods. UVM provides a solution to this dilemma in the form of a macro called `` `uvm_analysis_imp_decl() ``. The argument to this macro, which is `_verify_outputs` in this example, is used to uniquify an analysis imp export declaration and its associated `write()` method. Thus, the export that is built into the `uvm_subscriber` base class has a `write()` method, but the second analysis imp export being added to the scoreboard has been "uniquified". The export is declared with the class type `uvm_analysis_imp_verify_outputs` and the associated `write()` method is now called `write_verify_outputs()`. The monitor does not know that this name change has occurred. The monitor calls a `write()` method for its port, but when that port is connected to a uniquified (made different) export, the uniquified `write_verify_outputs()` method name is actually invoked.

3.7.3 The Scoreboard Expected Results Prediction

The monitor captures DUT input values and stores them in a `my_tx` object. The monitor then calls the port's `write()` method to send out a handle to this object. The `write()` method implementation in the scoreboard uses these input values to predict what the DUT outputs should be.

```
class my_scoreboard extends uvm_subscriber #(my_tx);
  ...
  // implement the write() method called by the monitor for observed DUT inputs;
  // predict what the DUT results should be for a set of input values
  function void write (my_tx t);
    my_tx expected_tx;
    $cast(expected_tx, t.clone()); // create new my_tx object and preserve input vals
    // calculate expected results and store in FIFO
    Case (t.opcode)
      ADD: expected_tx.result = t.operand_a + t.operand_b;
      SUB: expected_tx.result = t.operand_a - t.operand_b;
      ...
    endcase
    expected_tx.exception = ...
    expected_fifo.put(expected_tx); // save transaction handle
  endfunction: write
```

Example 18 – A simple UVM scoreboard expected results predictor (flat hierarchy)

The predicted results need to be stored. In this example, the `my_tx` sequence_item type contains variables for both the DUT inputs and the DUT outputs. The `write()` method receives a `my_tx` object handle from the monitor. The name of this handle is `t` (this name is defined in the base class for TLM analysis ports, and cannot be changed to a different name). This `t` object contains the DUT inputs values observed by the monitor.

The predictor creates a second `my_tx` object, in which to store the predicted results. This second object could be constructed using the factory `create()` method. However, in this simple example, the `my_tx` sequence_item `clone()` method is used to create this second `my_tx` object. The clone is called `predicted_tx`. By cloning the `my_tx` object received from the monitor, the scoreboard has a place to store the predicted results, and the values that led to those predicted results are copied into the clone. Later on, should the DUT outputs be found to be incorrect, having access to these input values can help in debugging the incorrect results.

The `write()` method containing the prediction algorithm is called by the monitor, based on the timing used by the monitor. The actual outputs from the DUT for that set of inputs will most likely not occur until one or more clock cycles later. In the example of the monitor shown earlier (Example 15), capturing DUT input values used a different clock than capturing DUT output values. We don't know which clock is faster, and the monitor and scoreboard should be coded so that it does not matter. In this example scoreboard, the prediction routine stores a handle to the `predicted_tx` sequence_item into a SystemVerilog mailbox, which serves as a FIFO.

3.7.4 The Scoreboard Results Verification

The portion of the monitor that captured the actual DUT outputs is also called a `write()`method. Through port/export connections and the `` `uvm_analysis_imp_decl() `` macro, the call invokes the `write_verify_outputs()` method. This method can get the `predicted_tx` handle back from the FIFO when the DUT output is available. If the FIFO is empty, the call to `get()` will block and wait for a predicted result. With a handle to a `my_tx` object that contains the actual DUT output values, and a handle to a `my_tx` object that contains the expected values, the `write_verify_outputs()` can verify that the DUT outputs are correct.

```
`uvm_analysis_imp_decl(_verify_outputs)

class my_scoreboard extends uvm_subscriber #(my_tx);
  ...
  // implement the write() method called by the monitor for actual DUT outputs;
  // compare the DUT outputs to the predicted results
  function void write_verify_outputs(my_tx t);
    my_tx expected_tx;
    expect_fifo.get(expected_tx);  // blocks if FIFO is empty
    if (t.result !== expected_tx.result || t.exception !== expected_tx.exception)
      begin: mismatch
        `uvm_error("SCBD_ERR:", "DUT outputs did not match expected results")
        `uvm_info("DUT OUT:", t.convert2string(), UVM_HIGH)
        `uvm_info("EXPECTED:", expected_tx.convert2string(), UVM_HIGH)
        num_failed++;
      end: mismatch
    else num_passed++
  endfunction: write_verify_outputs
```

Example 19 – A simple UVM scoreboard actual output verifier (flat hierarchy)

3.7.5 Reporting the Scoreboard Score

Scorecard counters keep track of how many evaluations passed and how many failed. The totals for these counters are reported at the end of simulation in the ***report phase***.

```
class my_scoreboard extends uvm_subscriber #(my_tx);
  ...

  function void report_phase(uvm_phase phase);
    `uvm_info("Scoreboard:", $sformatf("\n passed=%0d  failed=%0d\n",
      num_passed, num_failed), UVM_NONE)
  endfunction: report_phase
```

Example 20 – A simple UVM scoreboard scorecard reporter (flat hierarchy)

4.0 Sequence Writer

The ***Sequence Writer*** is responsible for supplying the ***Test Writer*** with a set of sequences that define stimulus and response functionality to be used in a particular test case. In keeping with the modularity goals of UVM, the ***Test Writer*** need not know the inheritance hierarchy, or even the details of what is in the sequence. All the ***Sequence Writer*** needs to provide is a list of sequence types and enough information so that the ***Test Writer*** knows on which sequencer(s) to start them.

4.1 Define a sequence_item

The basic unit of communication in a UVM testbench is the `uvm_sequence_item`, also referred to as a *transaction*. It is derived from a `uvm_object`, and encapsulates the fields and methods necessary to pass information between the sequence and the driver. The UVM infrastructure depends on sequence items having certain methods implemented, so it is always the case that your sequence items will be extended from the `uvm_sequence_item` base type, and you must supply implementations of the following methods [3]:

- `do_copy()`
- `do_compare()`
- `convert2string()`[3]
- `do_record()`
- `do_pack()` and `do_unpack()`

Example 21 illustrates a simple sequence_item definition. The implementation of the `do_xxx` methods (where "xxx" represents any of the methods listed above) is not shown in this example. It is beyond the scope of this paper to discuss how to write the implementation of these `do_xxx` methods. The code is primarily SystemVerilog programming, and does not require more than a couple additional UVM constructs beyond those discussed in this paper. Good information on implementing the `do_xxx` methods can be found in [3], [6] and [7].

```
class my_tx extends uvm_sequence_item;
  rand   bit   [23:0] operand_a;
  rand   bit   [23:0] operand_b;
  randc  opcode_t     opcode;
         logic [23:0] result;

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

  virtual function void do_copy(uvm_object rhs);
    ... // implementation not shown
  endfunction

  virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    ... // implementation not shown
  endfunction

  ... // implementation of remaining do_xxx methods
endclass: my_tx
```

Example 21 – A UVM sequence_item (aka "transaction")

These `do_xxx` methods can be implemented and populated using `` `uvm_field_xxx `` macros, but the resultant code is inefficient, hard to debug, and can be prone to error. The authors of this paper differ in opinion on whether it is preferred to use field macros or manually write the implementation of these methods. We agree, however, that manually coded `do_xxx` methods will improve both simulation performance and simulation memory footprint. Sutherland feels that, most of the time, the ease of use of the field macros is worth the trade-off of performance, and that manually writing the methods is only worthwhile if performance profiling shows a specific field macro code is causing a performance problem. Fitzpatrick feels the impact on performance can be significant on large verification projects, and it is, therefore, worth investing the time up front to manually write the implementation of the `do_xxx` methods.

Once the sequence_item is defined, it gets created in a sequence using the factory, and is subject to factory overrides as any other UVM object, as seen in the next section.

[3] UVM also provides `print()` and `sprint()` methods, but we recommend calling `convert2string()` directly in order to get a string representation of the contents of the sequence_item.

4.2 Sequence Body Method

The heart of a UVM sequence is the `body()` method, which is a user-defined method that defines the behavior of the sequence.

```
class tx_sequence extends uvm_sequence#(my_item);
  `uvm_object_utils(tx_sequence)
   ...
  task body();
    repeat(50) begin
      tx = my_seq_item::type_id::create("tx");
      start_item(tx);
      ...
      finish_item(tx);
    end
  endtask
endclass:tx_sequence
```

Example 22 – A UVM sequence body() method

4.3 Virtual Sequence

As previously mentioned in Section 2.4, the *virtual sequence* is used to coordinate the execution of other sequences on specific sequencers.

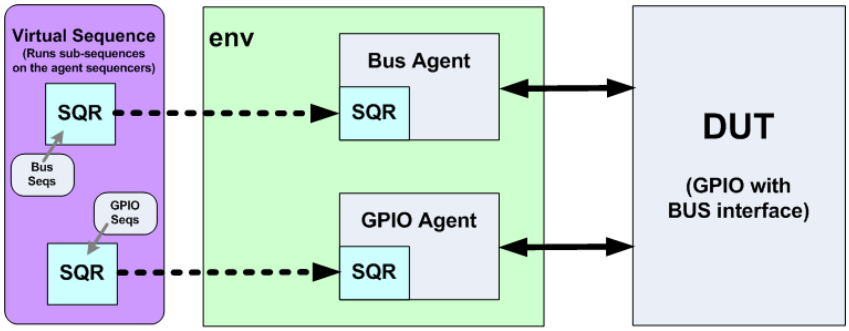

Figure 4 – The Virtual Sequence

The virtual sequence includes sequencer handles that will be set to point to the specific sequencers in the environment. Because the virtual sequence is a `uvm_object`, and not a `uvm_component`, we defer its creation to the *run_phase*. The sequencer pointers must be initialized in the virtual sequence before starting it.

```
typedef uvm_sequence #(uvm_sequence_item) uvm_virtual_sequence;

  // Virtual sequence example:
  class my_vseq extends uvm_virtual_sequence;
  ...
  // Handles for the target sequencers:
  bus_sequencer_t bus_sequencer;
  gpio_sequencer_t gpio_sequencer;

  virtual function void init(uvm_sequencer bus_seqr,
                             uvm_sequencer gpio_seqr);
    bus_sequencer = bus_seqr;
    gpio_sequencer = gpio_seqr;
  endfunction

  task body();
    ...
    // Start interface specific sequences on the appropriate target sequencers:
    aseq.start( bus_sequencer , this );
    bseq.start( gpio_sequencer , this );
  endtask
```

```
endclass
```

Example 23 – A UVM virtual sequence

How you choose to initialize your virtual sequence is a matter of preference[8]. The key is that the ***Sequence Writer*** and the ***Test Writer*** agree on the mechanism. After that, there isn't anything UVM-specific to do. It's a simple matter of programming.

5.0 Advanced Examples

The preceding sections have outlined everything you need to know to be able to use UVM to verify the vast majority of designs you will encounter. However, there are some situations that may arise which may require the use of a few more UVM constructs.

5.1 The `phase_ready_to_end` Method

As discussed earlier, phase execution is controlled by raising and dropping objections. As long as an objection is raised, the phase in which it was raised will continue to execute until all phase objections are dropped. The ***Test Writer*** is responsible for raising and dropping objections before and after sequences are executed, respectively, in the test's *run_phase*. Usually, this is sufficient to ensure that all transactions complete, but it is sometimes the case that a component needs to keep the phase executing until it has completed processing the last transaction. This may require additional time after the sequence that generated the transaction has completed. For performance reasons, we do not want this component to raise and drop an objection for every transaction, but there still must be a way for a component, such as an end-to-end scoreboard, to delay the termination of the phase. To accomplish this, we use the `phase_ready_to_end()` method.

```
function void my_component::phase_ready_to_end( uvm_phase phase );
  if( !is_ok_to_end() ) begin
    phase.raise_objection( this , "not yet ready to end phase" );
    fork begin
      wait_for_ok_end();
      phase.drop_objection( this , "ok to end phase" );
    end
    join_none
  end
endfunction : phase_ready_to_end
```

Example 24 – Implementing `phase_ready_to_end()`

The `phase_ready_to_end()` method is called automatically for each component when all objections to the current phase have been dropped, giving the component an opportunity to raise an objection again, in order to prevent the phase from ending (in this example, until the `wait_for_ok_end()` task returns).

5.2 Pipelines

In a pipelined bus protocol[4], a data transfer ("transaction") is broken down into two or more stages which are executed one after the other, often using different groups of signals on the bus. This type of protocol allows several transfers to be in-progress at the same time, with each transfer occupying one stage of the pipeline. In order to model a pipelined protocol successfully, the driver needs to have multiple threads running, each of which takes a sequence item and runs it through each of the pipeline stages. To keep the pipeline full, and thus to stress the design as much as possible, the driver needs to unblock the sequencer to get the next sequence item, once the first pipeline stage is complete, but before the current item has fully completed its execution. Also, the sequence itself needs to have separate generation and response threads, so that the generation thread can send items to keep the pipeline full and the response thread can handle responses when they come in.

Because the response will necessarily come back separately from the request, it is necessary to add one small piece of functionality to the sequence.

```
class my_pipelined_seq extends uvm_sequence #(my_seq_item);
  `uvm_object_utils(my_pipelined_seq)
  ...
  task body();
    my_seq_item req = my_seq_item::type_id::create("req");
```

```
    use_response_handler(1);
    ...
    start_item(req);
    ...
    finish_item(req);
    ...
  endtask

  function void response_handler(uvm_sequence_item response);
    ...
  endfunction
endclass: my_pipelined_seq
```

Example 25 – Pipelined UVM Sequence

The `response_handler()` function is automatically called when the driver returns a response transaction. This provides the separate response thread that allows the main generation thread in the sequence to keep the pipeline as full as possible. In order to facilitate this, we use a slightly different protocol in the driver.

```
class my_pipelined_driver extends uvm_driver #(my_seq_item);
  `uvm_component_utils(my_pipelined_driver)
  ...
  semaphore pipeline_lock = new(1);

  task run_phase(uvm_phase phase);
    ...
    fork
      do_pipelined_transfer;
      do_pipelined_transfer;
    join
  endtask

  task do_pipelined_transfer;
    my_seq_item req;
    forever begin
      pipeline_lock.get(); //unblock when semaphore is available
      seq_item_port.get(req); // instead of get_next_item(req);
      ...// execute first pipeline phase
      pipeline_lock.put(); // unlock semaphore
      ...// execute second pipeline phase
      seq_item_port.put(req);
    end
  endtask
endclass
```

Example 26 – Pipelined UVM Driver

The `get()` call in the driver unblocks the `finish_item()` call in the sequence, and the `put()` call sends the transaction back to the sequence to be processed by the `response_handler()` method. The UVM includes built-in functionality to be able to match the response transaction with the original request transaction, if necessary.

6.0 UVM Features to Avoid

6.1 Phase Jumping

The inclusion of the *reset_phase*, *configure_phase*, *main_phase* and *shutdown_phase* in UVM, which execute sequentially (along with their pre- and post- variants) in parallel with the *run_phase*, was perhaps one of the most unfortunate results of the inevitable "design by committee" approach used in developing UVM. While there may be some scenarios in which advanced UVM users find the use of the phases to be necessary, the vast majority of users will never need to use them. These phase methods should never be implemented in components like drivers and monitors and, when used, should only be used in tests to determine which sequence(s) to run at particular times in the simulation. This is what virtual sequences do.

The other reason not to use these phase methods is that it invites the user to experiment with jumping between phases, which is a use-model that is extremely complicated, with side effects that can lead to difficult-to-debug scenarios that ultimately make the use of these phases not worth considering.

6.2 Callbacks

Another feature of UVM that should be avoided is the use of callbacks. This mechanism of customizing behavior without using inheritance requires many convoluted steps to register and enable the callbacks, which have a non-negligible memory and performance footprint. In addition, this macro-driven feature suffers from a lack of control of the ordering in which various callbacks may be called.

Instead of callbacks, standard object-oriented programming (OOP) practices should be used. One OOP option would be to extend the class that you want to change, and then use the UVM factory to override which object gets created (which is what the UVM factory is for). Another OOP option would be to create a child object that gets functionality delegated to it from within the parent object. The functionality could be controlled via a configuration setting or a factory override.

6.3 UVM 1.2 Features

In addition to several features that have been in UVM for some time, there are a number of new features in UVM 1.2 that should also be avoided[5]. Among these are the `uvm_coreservice_t` class and the extended messaging macros (e.g. `` `uvm_info_begin ``/`` `uvm_info_end ``). These new features are not necessary for the verification of nearly all types of designs, and only make learning UVM and maintaining UVM testbenches more difficult.

7.0 Conclusion

The UVM Committee is to be congratulated for producing such an extensive and useful implementation of the UVM class library. UVM delivers on its goals of enabling the creation of modular reusable verification components, environments and tests. However, it is important to realize that when using UVM, most of the implementation of the library is dedicated to infrastructure and support of the relatively small set of features that ***Test Writers***, ***Environment Writers*** and ***Sequence Writers*** will actually use. Indeed, ***the set of features identified in this paper consists of 10 classes, 30 methods and 7 macros that users need to be familiar with***, in order to use UVM (in addition to proper SystemVerilog coding and general use-model approaches). When compared to the 357 classes and 1037 unique methods (938 functions and 99 tasks) that comprise UVM1.2, this means that ***UVM users really only need to learn 3% of UVM (2% of classes and 3% of methods) to be productive***.

Given the anecdotal feedback from many current and prospective UVM users that UVM is too complicated and too hard to learn, this paper should serve to allay those fears to some degree. Those feelings surely are exacerbated when one considers that the UVM Reference Manual, which is the "official documentation" for UVM, includes documentation for such a large number of classes and methods that will never (and in reality were never meant to) be deployed by UVM users. Perhaps the Accellera UVM Working Group could use this as an opportunity to streamline the UVM Specification to focus on including only the "user-facing" API in the version that it contributes to the IEEE.

8.0 References

[1] van der Schoot, Saha, Garg and Suresh, "Off To The Races With Your Accelerated SystemVerilog Testbench", http://events.dvcon.org/2011/proceedings/papers/05_3.pdf

[2] Arunachalam, "Controlling On-the-Fly-Resets in a UVM-Based AXI Testbench", http://www.mentor.com/products/fv/verificationhorizons/volume10/issue3/on-the-fly-resets

[3] Mentor Graphics, Online UVM Cookbook, "Transactions/Methods", https://verificationacademy.com/cookbook/transaction/methods

[4] Mentor Graphics, Online UVM Cookbook, "Driver/Pipelined", https://verificationacademy.com/cookbook/driver/pipelined

[5] Fitzpatrick, T. and Rich, D., "Of Camels and Committees: Standards Should Enable Innovation, Not Strangle It", DVCon 2014 Proceedings, http://events.dvcon.org/2014/proceedings/papers/08_1.pdf

[6] Erickson, "Are OVM & UVM Macros Evil? A Cost-Benefit Analysis", DVCon 2011 proceedings. Alternate url: http://www.mentor.com/products/fv/verificationhorizons/horizons-jun-11

[7] Cummings, "UVM Transactions: Definitions, Methods, and Usage", SNUG Silicon Valley 2014 proceedings. Alternate url: http://www.sunburst-design.com/papers/CummingsSNUG2014SV_UVM_Transactions.pdf

[8] Mentor Graphics, Online UVM Cookbook, "Sequences/Virtual", https://verificationacademy.com/cookbook/sequences/virtual

9.0 Appendix

This appendix contains a summary of the practical subset of UVM recommended in this paper. These are the relatively small number of UVM constructs that end users of UVM need to know, in order to write effective UVM tests, environments and sequences.

UVM Base Classes:

1) `uvm_test`
2) `uvm_env`
3) `uvm_agent`
4) `uvm_sequencer`
5) `uvm_driver`
6) `uvm_subscriber`
7) `uvm_sequence`
8) `uvm_sequence_item`
9) `uvm_analysis_port`
10) `uvm_analysis_imp_export` (or uniquified version)

Method Definitions:

1) `function new() (for uvm_objects and uvm_components)`
2) `function void build_phase()`
3) `function void connect_phase()`
4) `function void report_phase()`
5) `task run_phase()`
6) `task body()`
7) `function void write()` (or uniquified version)
8) `function void my_component::phase_ready_to_end()`
9) `function void response_handler()`

Method Calls:

1) `run_test();`
2) `uvm_config_db #(<type>)::set()`
3) `uvm_config_db #(<type>)::get()`
4) `obj_type::type_id::create()`
5) `component_type::type_id::create()`
6) `component_type::type_id::set_type_override()`
7) `component_type::type_id::set_inst_override()`
8) `super.build_phase()`
9) `sequence.start()`
10) `virtual_sequence.start()`
11) `phase.raise_objection()`
12) `phase.drop_objection()`
13) `dut_inputs_port.connect()`
14) `seq_item_port.get_next_item()`
15) `seq_item_port.item_done()`
16) `seq_item_port.get()`
17) `seq_item_port.put()`

18) `analysis_port.write()`
19) `start_item()`
20) `finish_item()`
21) `use_response_handler()`

Macros:

1) `` `uvm_object_utils() ``
2) `` `uvm_component_utils() ``
3) `` `uvm_fatal() ``
4) `` `uvm_error() ``
5) `` `uvm_warning() ``
6) `` `uvm_info() ``
7) `` `uvm_analysis_imp_decl() ``

Miscellaneous:

1) `import uvm_pkg::*;`

~~**9.1**~~