Easier UVM for Functional Verification by Mainstream Users

Updated and Extended for UVM 1.0

John Aynsley
Doulos
Church Hatch, 22 Market Place
Ringwood, United Kingdom
+44 1425 471223
john.aynsley@doulos.com

ABSTRACT

This paper describes an approach to using Accellera's UVM, the Universal Verification Methodology, for functional verification by mainstream users. The goal is to identify a minimal set of concepts sufficient for constrained random coverage-driven verification in order to ease the learning experience for engineers coming from a hardware design background who do not have extensive object-oriented programming skills. We describe coding guidelines to address the canonical structure of a UVM component and a UVM transaction, the construction of the UVM component hierarchy, the interface with the design-under-test, the use of UVM sequences, and the use of the factory and configuration mechanisms.

Keywords

SystemVerilog, UVM, functional verification

1. INTRODUCTION

This paper describes an approach to using Accellera's UVM, the Universal Verification Methodology, for functional verification by mainstream users as opposed to highly skilled verification specialists. It arises from experience at Doulos in teaching SystemVerilog and functional verification methodology to engineers from a broad cross-section of the hardware design and verification community. While much of the research and development in functional verification methodology is rightly focussed on the needs of power users as they solve the hardest verification problems, we find that the majority of mainstream users have a somewhat different focus, namely, how to become productive with SystemVerilog with a minimum of delay and specialist programming expertise. SystemVerilog and UVM provide mechanisms to create verification components for checking, coverage collection, and stimulus generation, and to modify the behavior of those components for specific tests. But SystemVerilog and UVM provide more than this, so much more in fact that the learning curve can be daunting for non-specialists.

The goal of this paper is to enable engineers with experience in Verilog or VHDL to become productive in UVM by learning a small number of new coding idioms, selected to minimize the conceptual clutter they have to deal with. As users become fluent with this set of basic idioms, they can then branch out to embrace the full feature set of UVM as and when they need.

We describe coding guidelines to address the canonical structure of a UVM component and a UVM transaction, the construction of the UVM component hierarchy, the interface with the design-under-test, the use of UVM sequences, and the use of the factory and configuration mechanisms. Starting from these simple guidelines, engineers can create constrained random verification environments in an object-oriented coding style that are fully compliant with the UVM standard, and hence are interoperable with UVM verification IP from other sources.

2. EASIER UVM?

Easier UVM is not yet another verification methodology. It is UVM. The point is to somewhat restrict the range of features being used in order to make life easier for the novice. This exercise is primarily a pedagogical one. The aim is to ease the task of learning UVM, not to deny users the ability to exploit the full power of the UVM class library. The coding guidelines we give here are not the only ways of using UVM, nor are they necessarily the best approach for experienced verification engineers.

Easier UVM does not mean *easy* UVM. The set of concepts presented below is still quite extensive and very rich. It is being increasingly recognized that both system modeling and functional verification require a high level of software programming skill, and on top of that, UVM also requires a deep understanding of coverage-driven verification and transaction-level modeling.

This paper does not explicitly list every UVM feature. Many useful features have not been mentioned, some of them essential for advanced UVM usage, significant examples being report handling and the end-of-test mechanisms. *Easier UVM* merely provides a conceptual foundation on which to build a deeper knowledge of UVM.

The version of UVM current at the time of writing of the original paper was the UVM 1.0 Early Adopter release. The paper you are reading has been updated to reflect the changes made in UVM 1.0, released in Feb 2011.

3. SYSTEMVERILOG AND UVM

Over the past few years, constrained random coverage-driven verification has been increasingly adopted as the methodology-of-choice for simulation-based functional verification to the point where it is widely used on the largest ASIC projects. SystemVerilog, as the only industry standard hardware verification language supported every one of the three largest EDA vendors, has displaced its rival single-vendor solutions in many companies.

But SystemVerilog is not without its problems. Although current SystemVerilog implementations are in many ways both mature and robust, SystemVerilog remains under-specified as a language. SystemVerilog was an extremely ambitious standardization project that was undertaken prior to the development of any complete proof-of-concept implementation, and as a result the IEEE 1800

SystemVerilog Language Reference Manual still has many areas of ambiguity, and more than one of the major implementations has significant gaps. The simulation tool vendors themselves are in the unenviable position of having to invest significant engineering resources in trying to pin down and implement a very complex language against an ambiguous definition, and so take the very reasonable position of prioritizing their implementation choices according to customer demand.

Nevertheless, certain areas of the SystemVerilog language do stand out as being well-defined and consistently implemented across all the major simulators. These include:

- The concise RTL features such as ANSI-style port and parameter declarations, always_comb, always_ff, unique, priority, and the abbreviated port connection syntax
- C-like control constructs such as for, foreach, do-while, break, continue, return
- C-like data type features such as typedef, enum, struct, and the 2-valued integer types
- VHDL-like package and import features
- Classes and the features for constraints and functional coverage based thereon
- The class-based data types, namely strings, queues, dynamic arrays and associative arrays
- Interfaces and virtual interfaces sufficient for communication between classes and modules

One reason that the features from the above list have been implemented so thoroughly and consistently is that, with the exception of the concise RTL features, they have been widely used to create the libraries of base classes that underpin the functional verification methodologies AVM, URM, VMM, OVM, and now UVM. Customer demand has pushed the EDA vendors to support each others' methodology class libraries, which in turn has driven the implementations to converge on a common understanding of the features and semantics of class-based SystemVerilog.

SystemVerilog and UVM now form a virtuous circle. The class-based SystemVerilog features that support constrained random verification are sufficiently well-defined and well-implemented to allow the development of robust *and* portable verification class libraries, and the widespread use of those libraries ensures the ongoing support of the necessary language features by the tool vendors.

In addition to the Verilog-like and C-like features of SystemVerilog, we make use of classes, constraints, covergroups, packages, interfaces and virtual interfaces. UVM makes heavy use of type parameterization to classes, and fortunately all of the major simulator implementations now agree on the semantics in this area.

4. OBJECT ORIENTED CONCEPTS

Object-oriented (OO) or aspect-oriented programming concepts are key to contemporary constrained random verification methodology because they enable reuse, yet these techniques are amongst the hardest to learn. OO techniques allow verification components to be specialized to the needs of a specific test bench or test without modifying their source code and enables well-structured communication between those components using function calls. We wanted to provide some of the expressive power of the OO paradigm without getting drawn into the full set of issues involved in OO programming.

In UVM, the class is used as a container to represent components, transactions, sequences, tests, and configurations. Let's take the component. Unlike the VHDL design entity or the SystemVerilog module, a component represents an abstraction across a whole family of possible structural building blocks. A component picks out what is common across several such building blocks, but a component is not *concrete*, meaning that it is not the final once-and-for-all definition of the thing. A VHDL design entity or a SystemVerilog module can describe a family of related components, but only if the variants are anticipated in advance and are explicitly captured in the source code by means of language features such as generic parameters and generate statements. Because it is a class, a UVM component can be extended after-the-fact in arbitrary ways. An extension can add new features or can modify existing features. In particular, we require this extension capability so that a test can extend a transaction or a sequence in order to add constraints, and then use the factory mechanism to override the generation of those transactions or sequences.

5. UVM CONCEPTS

The goal of this paper is to identify a minimal set of concepts sufficient for constrained random coverage-driven verification in order to ease the learning experience for engineers coming from a hardware design background who do not have extensive object-oriented programming skills. At the same time, we did not want to strip down the conceptual framework to the point where it lost all the expressive power of the object-oriented paradigm. Some other attempts to present verification class libraries to hardware designers have ended up doing little more than re-present the semantics of VHDL or Verilog using classes, which was a pitfall we wished to avoid. In our experience, hardware designers moving to a new language for verification, SystemVerilog in this case, do indeed want to benefit from the increased expressive power and flexibility afforded by a new paradigm.

Our conceptual vocabulary is listed below. These terms are elaborated later in the paper, and expanded definitions can be found in the documentation accompanying the UVM distribution.

- Component – a structural building block, conceptually equivalent to a Verilog module
- Transaction – a bundle of data items, which may be distributed over time and space in the system, and which form a communication abstraction such as a handshake, bus cycle, or data packet
- Sequence – an ordered collection of transactions or of other sequences
- Phase – execution is subdivided into various predefined phases that permit components to agree on when to build components, connect ports, run simulation, and so forth
- Factory – a function call that returns a component, transaction, or sequence, the type of which may be overridden from a test
- Port and export – connection points for transaction-level communication between components
- Generation – the creation of components, transactions, or sequences, where the properties of each may be set deterministically or at random under the control of constraints

- Test – a top-level component, which drives generation
- Configuration – an object associated with a component which may be set or randomized by a test and which it is used to configure that component as the component hierarchy is built
- Sequencer – a component that runs sequences and that sends transactions generated by those sequences downstream to another sequencer or to a driver
- Driver – a component that receives transactions from a sequencer and that drives the signal-level interface of the Design Under Test (DUT)
- Monitor – a component that senses the signal-level interface of the DUT and that sends transactions to the verification environment
- Coverage – functional coverage information can be collected using SystemVerilog covergroups within a component
- Checking – functional correctness of the DUT can be checked using either procedural code within a component or SystemVerilog assertions within an interface

5.1. Components

Components are used to build a component hierarchy, conceptually very similar to the design hierarchy in VHDL or the module hierarchy in Verilog. In this case the component hierarchy is part of the verification environment rather than the design, and components represent stimulus generators, drivers, monitors, coverage collectors, and checkers. The component represents the reusable unit of the verification environment, so has a standard structure and conventions for how it can be customized. Components are created quasi-statically at the start of simulation.

Here is a skeleton component. The uvm_component_utils macro and the function **new** should be treated as boilerplate code and written exactly as shown.

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)

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

  ...
endclass
```

5.2. Transactions

Transactions are the basic data objects that are passed between components. In contrast to VHDL signals and Verilog wires, transactions represent communication at an abstract level. In order to drive stimulus into the DUT, a so-called driver component converts transactions into pin wiggles, while a so-called monitor component performs the reverse operation, converting pin wiggles into transactions.

Here is a skeleton transaction. Again, the uvm_object_utils macro and function new should be treated as boilerplates.

```
class my_tx extends uvm_sequence_item;
  `uvm_object_utils(my_tx)

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

  ...
endclass
```

In the case of transactions and sequences, the one-and-only constructor argument needs a default value because it can be called from contexts that do not pass in a name.

5.3. Sequences

Sequences are assembled from transactions and are used to build realistic sets of stimuli. A sequence could generate a specific pre-determined set of transactions, a set of randomized transactions, or anything in between. Sequences can run other sequences, possibly selecting which sequence to run at random. Sequences can be layered such that higher-level sequences send transactions to lower-level sequences in a protocol stack.

Here is a skeleton sequence. It is similar to a transaction in outline, but the base class uvm_sequence is parameterized with the type of the transaction of which the sequence is composed. Also every sequence contains a **body** task, which when it executes generates those transactions or runs other sequences.

```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

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

  task body;
    ...
  endtask
  ...
endclass
```

Transactions and sequences together represent the domain of dynamic data within the verification environment.

5.4. Phase

Every component implements the same set of phases, which are run in a predefined order during simulation in order to synchronize the behavior of the components. When compared with VHDL or Verilog, UVM provides rather more temporal structure within a simulation run. The standard phases are as follows:

1. build – create child component instances
2. connect – connect ports to exports on the child components
3. end_of_elaboration – housekeeping
4. start_of_simulation – housekeeping
5. run – runs simulation – decomposed into several run phases
6. extract – post-processing
7. check – post-processing
8. report – post-processing
9. final – back stop

Each phase is represented by a function within the component, except for **run**, which is a task because it alone consumes simulation

time. If a function is absent, that component will be inactive in the given phase.

As you can infer from the above list, the primary distinction amongst the phases is between the phases for building the component hierarchy, connecting the ports, and running simulation, with additional housekeeping phases pre-pended and appended to the simulation phase.

5.5. Factory

The UVM factory mechanism is an implementation of the so-called *factory pattern* described in the OO literature. The UVM factory can make components, transactions, and sequences. Use of the factory enables the choice of object type to be overridden from the test, although a given component, transaction or sequence can only be overridden with one that *extends* the class of the original. This is one of the main mechanisms by which a reusable verification component can be customized to the current environment.

Here is an example of a component creating child components during the build phase:

```
class A extends uvm_component;
  `uvm_component_utils(A)

  B b;  // Child component
  C c;  // Child component

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Factory calls to create child components
    b = B::type_id::create("b", this);
    c = C::type_id::create("c", this);
  endfunction

  ...
endclass
```

In order to instantiate the complete component hierarchy the build functions themselves are called top-down. You call **create** within the **build** function of each individual component to instantiate its children.

The same factory mechanism is also used within a sequence to create transactions:

```
my_tx tx;
tx = my_tx::type_id::create("tx");
```

The other aspect of factories is overriding the behavior of the factory from a specific test. For example:

```
my_tx::type_id::set_type_override(alt::get_type());
```

The above statement would cause all instance of transaction type **my_tx** created by the factory to be replaced with instances of the transaction **alt**. Overrides can also be made per-instance:

```
my_tx::type_id::set_inst_override(alt::get_type(),
                                  "inst", this);
```

5.6. Port and export

Ports and exports are analogous to ports in VHDL or Verilog, but are used for transaction-level communication rather than signal-level communication. A component can send out a transaction out through a port, or receive an incoming transaction through an export. Transactions are passed as arguments to function calls, which may be non-blocking (return immediately) or blocking (suspend and wait for some event before returning), which is sufficient for basic synchronization within the verification environment. All detailed timing information should be pushed down into the driver and monitor components that connect to the DUT so that the timing can be determined by the DUT interface, which is typically locked to low-level clocks and other synchronization signals. Within the verification environment, control flow radiates outward from the DUT, with drivers calling **get** to request transactions from sequencers when they are ready for further stimulus, and monitors calling **write** to distribute transactions around the verification environment for analysis. A call to **get** from a driver may block if the stimulus generator is coordinating its activities with some other part of the verification environment. A call to **write** from a monitor is not permitted to block, because the DUT cannot be stalled waiting for analysis activity.

The example below shows a component A containing two child components B and C. The **connect** function connects **p_port** of component B to **q_export** of component C.

```
class A extends uvm_component;
  `uvm_component_utils(A)

  B b;  // Child component having p_port
  C c;  // Child component having q_export

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    b = B::type_id::create("b", this);
    c = C::type_id::create("c", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    b.p_port.connect( c.q_export );
  endfunction

endclass
```

One of the most common kind of port is the so-called *analysis port*, which serves as a broadcast mechanism used to send transactions to multiple passive verification components. An analysis port may be connected to any number of analysis exports, including none at all.

Transactions are actually sent through ports using OO function calls, for example:

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)

  uvm_analysis_port #(my_tx) aport;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    aport = new("aport", this);
  endfunction

  task run_phase(uvm_phase phase);
    my_tx tx;
```

```
    tx = my_tx::type_id::create("tx");
    ...
    aport.write(tx);
  endtask
endclass
```

You may notice from the above that the type of the analysis port is parameterized using the type of the transaction **#(my_tx)**, the analysis port has to be built explicitly during the build phase, and the transaction sent through the analysis port by the call to **write** is itself built using the factory.

5.7. Generation

Generation exploits SystemVerilog randomization and constraints. Component generation occurs quasi-statically during the so-called *build phase* of UVM, when a component is able to access its configuration object (if there is one) in order to control the generation of lower-level components. This is analogous to the elaboration phase in VHDL or Verilog. Sequence generation occurs dynamically. Control over the precise sequence of transaction that finally arrives at the DUT can be distributed across pre-defined sequence generation components and the test, which is able to extend and constrain existing sequences.

Here is an example showing transaction generation within the **body** task of a sequence:

```
class my_seq extends uvm_sequence #(my_tx);
  ...
  task body;
    my_tx tx;

    tx = my_tx::type_id::create("tx");
    start_item(tx);
    assert( tx.randomize() with { cmd == 0; } );
    finish_item(tx);
    ...
  endtask
endclass
```

In the example above, the transaction is being constrained as it is randomized in order to set the value of the command field to a specific value before sending the transaction downstream. The start_item and finish_item functions synchronize with the component that is pulling transactions from the sequencer, which could be a driver or another sequencer. start_item waits for the downstream component to request the transaction, finish_item waits for the downstream component to indicate that it has finished with the transaction. For its part, the downstream component calls **get** to fetch the transaction and may call **put** if it needs to send back a response.

The same mechanisms can be used to generate sequences themselves and hence to nest sequences within sequences:

```
task body;
  repeat(n)
  begin
    my_seq seq;
    seq = my_seq::type_id::create("seq");
    start_item(seq);
    assert( seq.randomize() );
    finish_item(seq);
  end
endtask
```

The body function of a sequence is just a regular function that executes procedural code. It is often useful to have the sequence read variables declared in the sequence class (e.g. **n** in the example above), which can be set or randomized externally.

5.8. Tests

A test is a top-level component used to control generation, that is, to customize the quasi-static behavior of the components that comprise the verification environment and the dynamic behavior of the transactions and sequences that pass amongst those components. A test can:

- Set the contents of configurations, which are then used to control the generation of the component hierarchy from the **build** function
- Override components with extended components in order to modify some aspect of their structure or functionality
- Override transactions and sequences with extended transactions and sequences that typically add constraints
- Start sequences on specific components called sequencers

A transaction or sequence can be customized in two ways: by using an in-line constraint when calling the randomize function (as shown above), or by declaring an extended class that adds or overrides constraints or functions from the original class. For example, an extended transaction:

```
class alt_tx extends my_tx;
  `uvm_object_utils(alt_tx)

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

  constraint my_constraint { data < 128; }
endclass
```

A user-defined test is created by extending a specific class uvm_test, as shown below:

```
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  ...
  my_env env;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Factory override replaces my_tx with alt_tx
    my_tx::type_id::set_type_override(
                           alt_tx::get_type() );
    env = my_env::type_id::create("env", this);
  endfunction

  task run_phase(uvm_phase phase);
    my_seq seq;
    seq = my_seq::type_id::create("seq");
    assert( seq.randomize() with { n = 22; } );
    seq.start( env.sequencer );
  endtask
endclass
```

Class uvm_test itself extends class uvm_component. This is one amongst several examples of built-in classes that are themselves components, including uvm_sequencer, uvm_driver, uvm_monitor, uvm_subscriber and uvm_env. It is a good idea to use these so-called *methodology base classes* rather than using raw uvm_components, because doing so makes the user's intent clearer.

Note how the test uses the factory to create a component instance containing the fixed part of the verification environment **env**, which is not itself test-specific but will be customized from the **my_test** class. Each test only need contain the few specific modifications to the verification environment that distinguish this test from the default situation. This particular test uses the factory to create an instance of the sequence **my_seq** and starts that sequence on a specific component within the verification environment named **env.sequencer**.

A particular test is run from a process within the top-level module:

```
initial
  run_test("my_test");
```

The test name can be set as shown or can be passed as a command line argument in order to select a test without any need for recompilation.

5.9. Configuration

A configuration is an object or *descriptor* associated with a specific component instance. The configuration is populated by the test during generation, and is then inspected by components in the component hierarchy during the *build phase*. Components in the hierarchy can also create local configurations for use by their children alone. A single configuration class type may be common to many component instances, or each component instance could have its own unique configuration instance if required. Every component instance need not have its own unique configuration object; a component can inspect the configuration object of one of its ancestors.

The definition of a configuration has some commonality with a transaction:

```
class my_config extends uvm_object;
  `uvm_object_utils(my_config)

  rand bit param1;
  rand int param2;
  string   param3;
  // Other configuration parameters

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

A configuration object would typically be populated from a test:

```
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  ...
  my_env env;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    begin
      my_config config = new;
      // Can randomize the configuration
      assert( config.randomize() );
      // Can set individual members
      config.param2 = 3;
      config.param3 = "filename";
      uvm_config_db #(my_config)::set(
          this, "*.*producer*", "config", config);
    end
    env = top::type_id::create("env", this);
  endfunction
endclass
```

In the example above, the configuration object is instantiated and populated during the build phase of the test *before* using the factory to create the remainder of the verification environment, which is thus able to be generated according to the values set in the configuration object. The configuration is then stored in the UVM configuration database using the call to **set**. The argument "*.*producer*" identifies the instances to which this configuration applies using an instance name pattern containing wildcards.

The configured component should inspect the configuration object during the build phase:

```
class producer extends uvm_component;
  `uvm_component_utils(producer)
  ...
  my_config config;

  // Configuration parameters
  bit    param1 = 0;
  int    param2 = 0;
  string param3;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    begin
      if ( uvm_config_db #(my_config)::get(
               this, "", "config", config) )
      begin
        param1 = config.param1;
        param2 = config.param2;
        param3 = config.param3;
      end
      ...
    end
  endfunction
  ...
endclass
```

In the example above, a component calls **get** to retrieve the configuration object identified by the string "config" (as set by the call to **set**) from the UVM configuration database and extracts the parameter values from the configuration. These parameters can be used to control generation locally within that component.

This configuration mechanism plays a similar role to generics in VHDL and parameters in Verilog, but it afford a lot more flexibility because firstly the configuration can be randomized using constraints, secondly a single configuration can be shared by many different component instances (rather than having to be set explicitly for each individual instance), and thirdly a component can use the values from a parent configuration to set the configuration for its own children.

5.10. Sequencer

A sequencer is a variety of component that runs sequences and sends them downstream to drivers or to other sequencers. At its simplest a sequencer looks like any other component, except that it has an implicit transaction-level export for connection to a driver.

We can now get a little more ambitious and show an example including sequence, sequencer, and launching the sequence from the test:

```
// A SEQUENCER is a component
class my_sqr extends uvm_sequencer #(my_tx);
```

```
  `uvm_component_utils(my_sqr)

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

// A SEQUENCE is generated dynamically
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

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

  task body;
    my_tx tx;
    tx = my_tx::type_id::create("tx");
    start_item(tx);
    assert( tx.randomize() );
    finish_item(tx);
  endtask
endclass

class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  ...
  my_env env;
  ...
  task run_phase(uvm_phase phase);
    my_seq seq;
    // Create the sequence
    seq = my_seq::type_id::create("seq");
    // randomize it
    assert( seq.randomize() );
    // and start it on the sequencer
    seq.start( env.agent.sqr );
  endtask
endclass
```

A sequencer can also receive transactions from other sequencers. Here is an example of one that does just that:

```
class ano_sqr extends uvm_sequencer #(ano_tx);
  `uvm_component_utils(ano_sqr)

  uvm_seq_item_pull_port #(my_tx) seq_item_port;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    seq_item_port = new("seq_item_port", this);
  endfunction
endclass
```

This sequencer can be connected to the previous sequencer, which sends it transactions of type **my_tx** through a port-export pair.

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

  my_sqr  sqr1;
  ano_sqr sqr2;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    sqr1 =  my_sqr::type_id::create("sqr1", this);
    sqr2 = ano_sqr::type_id::create("sqr2", this);
    ...
  endfunction

  function void connect_phase(uvm_phase phase);
    sqr2.seq_item_port.connect(
                                    sqr1.seq_item_export );
    ...
  endfunction
endclass
```

As discussed earlier, communication between the two sequencers will be accomplished by making transaction-level calls between the two sequencers, in the case with **sqr2** pulling transactions from **sqr1**. The actual sequence to be run on **sqr2** is shown below:

```
class ano_seq extends uvm_sequence #(ano_tx);
  `uvm_object_utils(ano_seq)
  `uvm_declare_p_sequencer(ano_sqr)
  ...
  task body;
    ...
    my_tx tx_from_1;
    p_sequencer.seq_item_port.get(tx_from_1);
    ...
```

Conceptually, all that is happening here is that a sequence, running on a sequencer, is pulling in transactions through a port on that sequencer. In order to do so, it needs direct access to the sequencer object that it is running on, which is provided by the predefined variable p_sequencer. Through p_sequencer, a sequence can refer to variables declared within the sequencer class, including ports and references to other external components. It turns out that this coding trick is all that is needed in order to have a sequence start child sequences on *another* sequencer, that is, on a sequencer other than the one it is itself running on. Such a sequence is called a *virtual sequence* because it does not itself generate transactions but instead controls the execution of other sequences, running on other sequencers.

5.11. Driver

A driver is a variety of component that always sits downstream of a sequencer. The driver pulls transactions from its sequencer and controls the signal-level interface to the DUT. The transaction-level interface between the sequencer and the driver is a fixed feature of UVM, and is unusual in the sense that both the port and the export required for TL- communication are implicit.

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

  virtual dut_if dut_vi;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    begin
      my_config config;
      uvm_config_db #(virtual dut_if)::get(
          this, "", "dut_vi", dut_vi);
    end
  endfunction

  task run_phase(uvm_phase phase);
    forever
    begin
      my_tx tx;
      seq_item_port.get(tx);

      // Wiggle pins of DUT
      dut_vi.cmd  = tx.cmd;
      dut_vi.addr = tx.addr;
```

```
      dut_vi.data = tx.data;
    end
  endtask
endclass
```

In the example above, the driver communicates with the sequencer by means of the call **get(tx)** through the implicit seq_item_port. Having got a transaction, it then drives the signal-level interface to the DUT by making assignments to the members of a SystemVerilog interface, which is done through the virtual interface **dut_vi**. The virtual interface is the SystemVerilog language mechanism that is used to pass data between structural Verilog modules and the class-based verification environment.

The example above all shows how a configuration object can be used to pass the virtual interface down to the driver during the build phase.

5.12. Monitor

A monitor is a variety of component that is confined to having passive access to the signal-level interface of the DUT. The monitor monitors traffic going to and from the DUT from which it assembles transactions which are distributed to the rest of the verification environment through one or more analysis ports.

All of the necessary concepts have already been discussed above. Here is an example:

```
class my_monitor extends uvm_monitor;
  `uvm_component_utils(my_monitor)

  uvm_analysis_port #(my_tx) aport;

  virtual dut_if dut_vi;
  ...
  task run_phase(uvm_phase phase);
    forever
    begin
      my_tx tx;

      // Sense the DUT pins on a clock edge
      @(posedge dut_vi.clock);
      tx = my_tx::type_id::create("tx");
      tx.cmd  = dut_vi.cmd;
      tx.addr = dut_vi.addr;
      tx.data = dut_vi.data;

      aport.write(tx);
    end
  endtask
endclass
```

5.13. Coverage and checking

SystemVerilog itself has many language features in support of the collection of functional coverage data and checking for functional correctness. In UVM, such code would typically be placed in a verification component that receives transactions from a monitor. The coverage and checking are best kept separate from the monitor in order that each may be reused more easily; monitor components are usually specific to particular protocols but are independent of the application. In contrast, functional coverage and checking code is usually highly application-specific but may in some cases be independent of the protocols used to communicate with the DUT.

An analysis port can be unconnected or can be connected to one or more analysis exports. A *subscriber* is a variety of component that has one built-in analysis export ready-for-use:

```
class my_subscriber extends uvm_subscriber #(my_tx);
  `uvm_component_utils(my_subscriber)

  // Coverage registers
  bit cmd;
  int addr;
  int data;

  covergroup cover_bus;
    coverpoint cmd;
    coverpoint addr;
    coverpoint data;
  endgroup
  ...
  // Function called through analysis port
  function void write(my_tx t);
    cmd  = t.cmd;
    addr = t.addr;
    data = t.data;
    cover_bus.sample();
  endfunction
endclass
```

The monitor and the subscriber can be generated at the next level up in the component hierarchy using the factory and the port-export connection made in the connect phase:

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

  my_monitor    monitor;
  my_subscriber subscriber;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    monitor    = my_monitor::type_id::create(
                                 "monitorh" , this);
    subscriber = my_subscriber::type_id::create(
                                 "subscriber", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    monitor.aport.connect(
                      subscriber.analysis_export );
  endfunction
endclass
```

Note the reference to the implicit analysis_export of the subscriber.

Sometimes a checking component may need to receive two or more incoming transactions streams, in which case the single implicit analysis export of the uvm_subscriber is insufficient, and it is necessary to declare analysis_exports explicitly:

```
class A extends uvm_component;
  ...
  uvm_analysis_imp #(tx1, A) analysis_export;
  ...
  function void write(tx1 t);
    ...

class B extends uvm_component;
  ...
  uvm_analysis_imp #(tx2, B) analysis_export;
  ...
  function void write(tx2 t);
    ...

class my_checker extends uvm_component;
```

```
  ...
  // Two incoming transaction streams
  uvm_analysis_export #(tx1) tx1_export;
  uvm_analysis_export #(tx2) tx2_export;
  ...
  A a;
  B b;
  ...
  function void connect_phase(uvm_phase phase);
    // Bind exports to two separate children
    tx1_export.connect( a.analysis_export );
    tx2_export.connect( b.analysis_export );
  endfunction
```

SystemVerilog does not support function overloading so it is only possible for a class to have a single **write** function. Hence ultimately each analysis export needs to be associated with its **write** function in a separate class, classes A and B in the example above.

5.14. Transaction operations

It is common to need to perform operations on transactions, operations such as printing out the contents of a transaction, making a copy of a transaction, or comparing two transactions for equivalence. For example, the subscriber described above may wish to log the data from a transaction or compare a transaction from the DUT with another transaction representing the expected behavior. UVM provides a standard set of functions for purposes such as these, as illustrated below:

```
function void write(my_tx t);
  ...
  my_tx tx;
  tx.copy(t)
  history.push_back(tx);
  if ( !t.compare(expected))
    `uvm_error("mismatch",
               $sformatf("Bad transaction = %s",
               t.convert2string()));
endfunction
```

Firstly, the **copy** function takes a complete copy of the transaction passed as an argument, in this case storing the copy in a queue of past transactions. Secondly, the **compare** function compares two different transactions for equivalence. Finally, the **convert2string** function returns a string representing the contents of the transaction in printable format.

There is more to a transaction than meets the eye. As well as containing data fields representing properties of the protocol being modeled, a transaction object may contain housekeeping information such as timestamps, logs, and diagnostics. This secondary information typically needs to be treated differently when performing **copy**, **compare**, or **convert2string** operations, and such differences need to be accounted for by the user when declaring transaction classes. For example:

```
class my_tx extends uvm_sequence_item;
  `uvm_object_utils(my_tx)
  rand bit cmd;
  rand int addr;
  rand int data;
  ...
  function string convert2string;
    return $sformatf(...);
  endfunction

  function void do_copy(uvm_object rhs);
    my_tx rhs_;
    super.do_copy(rhs);
    $cast(rhs_, rhs);
    cmd  = rhs_.cmd;
    addr = rhs_.addr;
    data = rhs_.data;
  endfunction

  function bit do_compare(uvm_object rhs,
                          uvm_comparer comparer);
    my_tx rhs_;
    bit status = 1;
    status &= super.do_compare(rhs, comparer);
    $cast(rhs_, rhs);
    status &= comparer.compare_field("cmd",  cmd,
                      rhs_.cmd,  $bits(cmd));
    status &= comparer.compare_field("addr", addr,
                      rhs_.addr, $bits(addr));
    status &= comparer.compare_field("data", data,
                      rhs_.data, $bits(data));
    return(status);
  endfunction
endclass
```

Note that the behavior of **copy** and **compare** are overridden by providing functions named **do_copy** and **do_compare**, respectively, as part of the transaction class. Each function should exclude any housekeeping or diagnostic fields.

5.15. Reporting

The `uvm_error macro used above is one of four standard macros for message reporting, the others being `uvm_info, `uvm_warning, and `uvm_fatal. The first macro argument is the message type, which can be used to categorize the report when customizing the behavior of the report handler. The second macro argument is the text of the report. In the particular case of an information report, there is a third macro argument which specifies a verbosity level, e.g:

```
`uvm_info("message_type", "message", UVM_LOW)
```

Reports with a verbosity level greater than a certain maximum level will be filtered out.

5.16. End-of-test mechanism

The end-of-test mechanism was cleaned up between the UVM Early Adopter release and the UVM final release. Detecting when the test has ended now relies on every component and sequence raising and dropping so-called *objections*. A component or sequence should raise an objection whenever it becomes busy generating or processing some activity, and should drop the objection whenever it finishes what it is doing, before waiting on the next transaction or event.

Here is an example of raising and dropping objections within a sequence:

```
task body;
  uvm_test_done.raise_objection(this);

  repeat(n)
  begin
    ...
  end

  uvm_test_done.drop_objection(this);
endtask
```

uvm_test_done is a global object that can be used to end tests. It is also possible to raise and lower objections to ending individual phases, for example:

```
task run_phase(uvm_phase phase);
  phase.raise_objection(this);
  ...
```

The current phase will finish after the last objection has been dropped, even if there are other active components such as clock generators still running. The only catch is that **raise_objection** must be called *before* the first non-blocking assignment region of the scheduler, otherwise the run phase will finish with simulation time still at zero.

6. CONCLUSION

UVM is a rich and capable class library that has evolved over several years from much experience with real verification projects large and small, and SystemVerilog itself is a large and complex language. As a result, although UVM offers a lot of powerful features for verification experts, it can present a daunting challenge to Verilog and VHDL designers who want to start benefitting from test bench reuse. The guidelines presented in this paper aim to ease the transition from HDL to UVM.

7. REFERENCES

[1] IEEE Std 1800-2009 "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language", http://dx.doi.org/10.1109/IEEESTD.2009.5354441

[2] Universal Verification Methodology (UVM) 1.0 Class Reference, Feb 2011

[3] Universal Verification Methodology (UVM) 1.0 User's Guide, February 23, 2011

[4] On-line resources from http://www.accellera.org/activities/vip

[5] On-line resources from http://www.uvmworld.org/

[6] On-line resources from http://www.doulos.com/knowhow/sysverilog/uvm/