Making the most of SystemVerilog and UVM: Hints and Tips for new users

Dr David Long

Doulos
Ringwood, UK

www.doulos.com

ABSTRACT

In the two years since UVM 1.0 was released by Accellera, Doulos has seen a big increase in the number of customers wanting to learn SystemVerilog: UVM has become the new standard verification environment for many companies. However, engineers often find that the size and complexity of the SystemVerilog language and the UVM class library make it hard to learn how to make effective use of their many features. This paper is intended to give guidance to engineers about some useful features of both SystemVerilog and UVM that are often overlooked, used incorrectly or simply avoided because they are perceived as being too hard to understand. The first part identifies the most common novice-user mistakes and sets out rules to avoid them. The second part discusses useful enhancements to verification capabilities that can be obtained from a deeper understanding of several SystemVerilog and UVM features. It provides simple examples and guidelines to show how each enhancement can be achieved.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

In the two years since version 1.0 of the Universal Verification Methodology (UVM) was released by Accellera Systems Initiative, Doulos has seen a big increase in the number of customers wanting to learn SystemVerilog: Many companies have made a decision to move away from the languages and methodologies that they traditionally used and to adopt SystemVerilog and UVM as their standard verification environment. In many East-Coast and European companies, this often requires engineers with experience of writing RTL style VHDL to learn a language and methodology that looks very different. Furthermore, the size and complexity of the SystemVerilog language and the UVM class library make it hard to learn how to make effective use of their many features. This paper is based on our observations of the difficulties faced by typical engineers as they migrate to UVM. It is intended to provide guidance for novice UVM users to help them to avoid common mistakes and to suggest enhancements that might otherwise be overlooked, used incorrectly or simply avoided because they are perceived as being too hard to understand.

Section 2 of this paper identifies some of the most common mistakes in SystemVerilog testbenches made by novice users. Section 3 explores common mistakes made by UVM novices and offers some suggestions and guidelines to help avoid them. Section 4 discusses some useful enhancements to verification capabilities that can be obtained from a deeper understanding of several SystemVerilog and UVM features. It provides simple examples and guidelines to show how each enhancement might be achieved.

2. Common Mistakes of SystemVerilog Novices

Errors made by novice SystemVerilog and UVM users may be due to one or more of the following factors:

Lack of familiarity with the SystemVerilog language
Lack of understanding about how SystemVerilog classes work
Lack of familiarity with the UVM base class library
Confusion about the architecture of a UVM-based verification environment
Uncertainty about the code that must be written in order to run a test using UVM

We believe that all of the issues mentioned above can be addressed by following a well-designed training programme, however, project timescales and budget limitations often result in engineers having to teach themselves the core elements of SystemVerilog and UVM on-the-job. Mistakes in the code written by inexperienced and self-taught users may not always be revealed when the code is compiled, leading to frustration and more often than not, a lot of time wasted debugging testbenches that do not perform as expected. Examples of mistakes that we commonly see are described in more detail in the following sections.

2.1 Fundamental SystemVerilog Errors

Engineers with a VHDL background do not always appreciate the differences between the use and semantics of various features in SystemVerilog. One of the most common problems is not understanding the correct use of blocking and nonblocking assignments (e.g. race conditions

caused by blocking assignments in multiple clocked processes, incorrect use of nonblocking assignments for task output variables, assignments to class members, and so on). Another common issue is failing to understand the rules used to set the word length of operands and the result value in expressions (e.g. automatic truncation and extension of operands, self-determined versus context-determined expressions). Fundamental errors such as these will not be explored further since there is little we can suggest that will prevent them, other than reading a good Verilog/SystemVerilog text book! (e.g. [1][2])

2.2 Packages

The use of packages is encouraged in SystemVerilog as a mechanism for code reuse (note that the UVM base class library is a good example since it is accessed by including a package). However, if used incorrectly, packages can lead to some unexpected errors.

Unlike VHDL, in SystemVerilog user-defined packages are often created by using `` `include `` to pull in declarations from a list of other source files. Where source files `` `include `` other source files, there is the danger that declarations might end up being included in a single package multiple times, or in two different packages that are both imported into the same scope – these would both generate compile time errors. Careful use of macros (`` `define ``) and conditional compilation directives (`` `ifdef `` or `` `ifndef ``) around declarations is recommended to prevent these errors. However, if packages are compiled as separate "compilation units", any compilation directives set when one package was compiled would no longer be defined when subsequent packages were compiled and so would not stop errors due to `` `include `` of the same file in multiple packages. The only solution in this case is to avoid the use of wildcards if multiple packages must be imported into the same scope.

```
package pkg1;
  typedef struct {  int a;   int b;   } my_struct;
endpackage: pkg1

package pkg2;
  typedef struct {int a; int b; } my_struct;
endpackage: pkg2

module top();
  import pkg1::*; //pkg1::my_struct potentially locally visible
  import pkg2::*; //importing name from 2 packages is illegal

  my_struct str;
endmodule: top
```

Figure 1 Illegal SystemVerilog Package Imports

2.3 Interfaces, Virtual Interfaces and Modports

SystemVerilog interfaces are generally welcomed by new users as a convenient mechanism to encapsulate all of the connections to a bus in their designs and testbenches. However, the use of virtual interfaces is something that often causes confusion since there is no equivalent construct in other hardware description languages. Common errors include omitting the `virtual` keyword when declaring a virtual interface variable or parameter and getting the interface type name and interface instance name mixed up.

The situation gets even more confused if modports are involved! As a matter of principle, we would recommend the use of modports with virtual interfaces since they impose a set of rules to control access to the interface members and should therefore help to prevent unexpected situations from arising accidentally (e.g. writing to a variable that is intended to be read-only). Unfortunately, modports are often used in an ad-hoc manner and even when present, they can easily be bypassed (deliberately or inadvertently). Care is also required when specifying a virtual interface that includes modports – older versions of some SystemVerilog simulators had restrictions on the types of operation that could be performed on interface members via a modport on a virtual interface.

```
interface intf(input logic clk, reset);
  import types::*;
  addr_t addr;
  rdata_t rdata;
  wdata_t wdata;
  logic RE;
  logic WE;
  modport mport(input clk, reset, output addr, input rdata,
                output wdata, RE, WE);
endinterface: intf
```

Figure 2 SystemVerilog Interface with Modport

2.4 Differences between Modules and Classes

The use of classes in SystemVerilog often unsettles engineers more used to writing RTL code, especially if they have no experience of class-based software languages such as C++ or Java.

One of their first concerns is often trying to understand when and how objects get created. This is further compounded by there being two very different types of object in class-based verification environments such as UVM: Classes representing components (e.g. drivers and monitors) and classes representing transactions that are passed between such components. Both types of object must be created by calling their constructor (new) but at different points in the simulation. Components are normally created at time zero (just as module instances are created during the elaboration stage) while transactions are usually created throughout the simulation. Attempting to access an object that has not yet been created is a very common error that is usually only picked up when the simulation crashes! Another common error related to object creation occurs when the constructor is called with the wrong arguments. This can occur if the code writer does

not know what arguments are expected by a call to new. Alternatively, it can happen if the author of a custom class is not aware of the arguments that will be automatically passed to `new` (e.g. by a UVM `create` function) .

```
class my_class;  //a simple class with constructor
  int i; real r;
  function new(int iv, real rv); i = iv; r = rv; endfunction
endclass: my_class

module test();
initial begin
  my_class c1,c2,c3; //handles used to point to object
  c1.i = 10;         //run-time error - c1 points to null!!
  c2 = new(10,1.2);  //creates new object (allocates memory)
  c2.i = 20;         //OK - c2 points to valid object
  c1 = c2;           //does not create a copy of object
  c1.i = 100;        //modifies object also pointed to by c2
  $display(c2.i);    //100
  c3 = new c2;       //creates new object, values copied from c2
...
```

Figure 3 Issues when accessing and copying objects

Misunderstandings about how transaction objects are copied are also common. Novices often (wrongly) believe that passing an object to a task or returning one from a function will automatically create a copy, when in fact, it is simply passing a reference to the original object. In many cases, the object reference will be used immediately so this does not generate an error. However, if an object must be stored until some future time step (e.g. in a FIFO buffer that holds multiple, pending transactions), the results could be very different. In the worst case, all stored object references could point to the same transaction object, with each new write to the transaction modifying the content seen through the other references. The correct approach in this case, should be to make an explicit copy of the transaction (by calling new) either before each call to the task it is passed to, or before the function returns, or wherever it is received and stored.

The use of classes in a verification environment also exposes users to object-oriented characteristics which they are often unfamiliar with. One of the questions that novices ask most often when creating their own classes is "how should I use inheritance properly?" For the component classes in a typical UVM environment, this is usually an easy question to answer since most component classes will be directly derived from an appropriate UVM base class and it will not be necessary to create further derived classes. The situation for transaction and sequence classes is not quite so clear. Tests often require multiple classes to be created that each correspond to slightly different kinds of stimulus. In these cases, a hierarchy of classes with multiple levels of inheritance can provide an efficient mechanism to create each of variety of stimulus while minimising the number of lines of code that must be written. However, the improper use of inheritance can lead to code that runs inefficiently and is hard to understand, maintain and reuse. A base class might contain code that is not required in, or relevant to, derived classes: In extreme cases, there may be members of a base class that must be disabled or hidden in a derived class. Both of these indicate a poorly designed class hierarchy.

3. Common Mistakes in UVM and How to Avoid Them

The first thing that strikes many engineers when they first encounter UVM is the size and complexity of the class library and the number of concepts that must be absorbed. Those who have a background in RTL verification are often unfamiliar with the operation of transaction level testbenches and "bus functional models" (BFMs). These are both factors that increase the likelihood of mistakes being made when switching from RTL to UVM based verification.

3.1 Simulation Phases

The UVM base classes provide the communication and synchronisation mechanisms required to build a powerful and flexible verification environment, thereby saving testbench developers a lot of work. On the other hand, if the use of these mechanisms is not properly understood, it often leads to a situation where the testbench does not run properly, if at all! The phase synchronisation methods are inherited by every UVM component class from their `uvm_component` base class. The `build_phase` and `connect_phase` virtual methods should be overloaded in derived component classes (where appropriate) to ensure child components are built and connected together in a controlled manner. If new component classes are derived from existing component classes, it is generally required to call `super.build_phase` or `super.connect_phase` to ensure any building or binding performed in the base class is still applied even though the base class has been replaced by a derived alternative. It is also worth pointing out that users migrating from the Open Verification Methodology (OVM) might accidentally use the similarly named OVM phase methods rather than the official UVM ones, e.g. function `build` in OVM should be replaced by function `build_phase` in UVM – there are some subtle differences between the two (a call to `super.build_phase` will also call `super.build`, however a call to `super.build` will not call `super.build_phase`).

A source of confusion for many new users of UVM is the operation of components such as monitors which may behave quite differently from those in traditional RTL testbenches. An RTL monitor will be a design entity/module that typically contains a single VHDL process or Verilog always block which samples its input signals on every active clock edge. A monitor in UVM will most likely be an instance of a class derived from `uvm_monitor`. Classes do not contain (static) processes. Instead, a UVM monitor will include one or more member tasks (usually just `run_phase`) that define its operation. Unlike processes, tasks are not started automatically at the start of simulation: they must be called explicitly from a running process once the simulation is running. The UVM phase scheduler ensures that the `run_phase` task is called automatically, at the correct point in the simulation, for every class derived from `uvm_component`. It is the responsibility of the testbench author to ensure the code they write within the `run_phase` task causes it to wake up whenever input signals are required to be sampled, to suspend between sampling events and to continue running until the end of the phase. Common mistakes include failing to place actions within a suitable loop (results in a monitor that only reads its signals once) and not waiting for the correct signals (results in values not being sampled at the correct time).

```
class my_mon extends uvm_monitor;
...

  task run_phase(uvm_phase phase);
    forever begin  //infinite loop typically required
      @(vi.cb);  //wait for clocking block event
      ... //read inputs
      #settling_time;
      ... //read outputs and send transaction to checker
    end
  endtask

  ...
endclass
```

Figure 4 Typical UVM Monitor Class run_phase

It is common for a monitor in an RTL testbench to perform checks of some kind. These checks could be for adherence to a particular bus protocol or could test the observed data against expected values. They might also involve more formal checks that make use of SystemVerilog Assertions (SVA). It is common for novices to attempt to implement all of these functions in the `run_phase` of a UVM monitor but this would be wrong on two accounts. Firstly, UVM monitors are generally written as part of a reusable component for a particular type of interface (e.g. a standard bus such as APB) and so should not contain any checks that are related to the functionality of a particular device under test. Instead, whenever they detect significant activity on their interface, they should assemble an appropriate transaction object and send it to a separate component that is responsible for testing the device functionality. The connections between UVM components are designed to work at this "transaction level" rather than responding to individual signal events. Secondly, it is illegal to place SVA (`assert property`) statements within a class. SVA properties are a convenient approach for checking bus protocol but are best placed within a SystemVerilog interface since they are not allowed within the UVM environment.

3.2 Use of Macros

UVM includes a set of macros that are intended to make it easier and quicker to create components and sequences. There are negative aspects to using macros (such as making the resultant code unnecessarily large and creating issues with debugging) that have led some experts to suggest that certain UVM macros should be avoided [3]. Novice users are often not properly aware about which macros are "good" or "bad" and may wrongly believe that macros should never be used. We have found that even when new users are aware of the macros that they should use, they often make mistakes because they do not know what the macros actually do and what might happen if they use the wrong macro. This section looks at some of the issues that we have seen when macros are used incorrectly or where they do not work as users expect.

UVM component and transaction objects are generally created using the UVM "factory". This mechanism enables the types of object created to be overridden for a particular test without making any changes to the source code. UVM provides a macro that adds the code required to create

objects using the factory. We strongly recommend that this macro is added to the component and transaction class declarations, otherwise the factory will not work! Unfortunately, there are different macros for transactions (derived from `uvm_object`) and components (derived from `uvm_component`). New users often forget to include this macro or include the wrong one (e.g. using `` `uvm_object_utils `` inside a component). Both of these mistakes will generate rather obscure error messages when the code is compiled! Examples showing the correct use of these macros are provided later (see Figure 11 and Figure 12). UVM provides "field automation macros" (e.g. `` `uvm_field_int ``) that may be called to add class members to automatically generated `print`, `copy`, `clone`, etc. functions. These fall into the group of "bad" macros mentioned above so we will not discuss them further here, other than to mention the importance of reading the UVM documentation to establish how these functions behave. A common mistake when using the field automation macros, is to forget to replace the `` `uvm_object_utils `` macro with `` `uvm_object_utils_begin ... `uvm_object_utils_end `` (and similarly for the component macro).

There are nearly twenty different macros in UVM that may be called as part of a sequence to create and send sequence items or child sequences. There is considerable overlap between these macros. We have found that new users are often confused about the differences between the macros and have doubts about which one(s) to use. These users are generally wanting prescriptive directions to help them write sequences and (unlike Perl programmers) do not appreciate "There's More Than One Way To Do It"! Since the macros only contain a few simple task/function calls, we would suggest that it is easier for new users to avoid the sequence macros completely (for an example, see Figure 14).

UVM provides macros to report information, warning and error messages. Even though these macros are very simple, it is recommended that they should always be used instead of calling the corresponding `uvm_report` functions directly. The rationale for the macros is to test the verbosity settings for the report and only call the report handler when the verbosity is lower than the current threshold. New users who are not aware of this recommendation and call the report functions directly may find this has a significant detrimental effect on the simulation performance. Users sometimes complain that using the report macros to generate messages from inside sequences are prefixed with an unwanted string that lists the full hierarchical path name of the sequencer that is running the sequence plus the name of the sequence object itself. This can be avoided by using an alternative macro `` `uvm_info_context `` and setting the context (the last argument) to `uvm_top`.

```
start_item(req);
void'(req.randomize());
`uvm_info_context("Serial Seq",req.convert2string(),
                  UVM_LOW,uvm_top)
finish_item(req);
```

Figure 5 Use of uvm_info_context within a Sequence

3.3 The Configuration Database

UVM includes a configuration database that provides a powerful and flexible mechanism to adjust the environment for specific tests without needing to modify any code. However, novice users frequently make mistakes when writing entries into the database and when attempting to extract the required values from the database. The situation is further complicated by UVM including two related databases that new users can easily mix up. The configuration database allows named values to be stored and retrieved by calling `set` and `get` functions respectively with a component's hierarchical path name as one of the arguments. The resource database uses a "scope" string plus a name to access values. In fact, the configuration database is simply a wrapper around the resource database that sets the scope parameter to the full hierarchical path of a component. We recommend that new users should use the configuration database in preference to the resource database, even though the resource database might appear to have a simpler user interface.

Part of the confusion in using the configuration database is due to the support for wildcards and regular expressions in the component path name and the rules that determine how this gets matched. The most common mistake is providing a search expression for an invalid path name, either due to misunderstandings about how the component hierarchy is created or about the starting point for relative searches.

Another source of confusion lies in the mechanism to override configuration settings as the component hierarchy is built. This is explored further in [4].

3.4 Objections

The UVM objection mechanism is used to ensure that the run-time phase tasks of components do not advance to the next phase until the current phase of every component is ready. Novice users often fail to understand how objections should be used. Common errors are premature end of simulation due to a failure to raise objections or simulations that never complete due to raised objections never getting dropped. This section looks at how these issues occur and how they might be avoided.

A very common error when users attempt to write their first UVM test is to forget to raise an objection at the start of the run phase. If no pre-existing components or sequences raise an objection during the run phase, the simulation will terminate immediately (unlikely to be the desired behaviour). Raising an objection near the start of the test's `run_phase` method will ensure the simulation can run beyond time 0. This brings us on to the second common error: forgetting to drop the objection once the test is over! If an objection is never dropped, simulation will continue until the simulation time reaches the "timeout" threshold – this could take a very long time if the test bench includes clock generators that run until the simulation is forcibly stopped! Forgetting to drop an objection will result in a message at the end of simulation along the lines of Figure 6.

```
# UVM_ERROR verilog_src/uvm-1.1b/src/base/uvm_phase.svh(1210) @
9200000000000ns: reporter [PH_TIMEOUT] Default phase timeout of
9200000000000ns hit. All processes are waiting, indicating a
probable testbench issue. Phase 'run' ready to end
# UVM_WARNING @ 9200000000000ns: run [OBJTN_CLEAR] Object 'com-
mon.run' cleared objection counts for run
```

Figure 6 Error Due to not Dropping Objection

To ensure at least one objection is raised and dropped correctly, we would recommend the template given in Figure 7 for the `run_phase` method of the test.

```
task run_phase(uvm_phase phase);
    master_sequence seq;  //top-level sequence
    seq = master_sequence::type_id::create();
    phase.raise_objection(this);
    //start top-level sequence and wait until it completes
    seq.start(m_env.m_sequencer);
    phase.drop_objection(this);
  endtask: run_phase
```

Figure 7 Template for run_phase with Objection

The other main error that new users make with objections is raising and dropping them unnecessarily. A typical example might be a driver that raises an objection whenever it pulls a new transaction from a sequencer and drops it once the transaction has been sent to the bus. A better solution if the transactions are generated within a finite-length sequence, would be to raise a single objection at the start of the sequence and drop it at the end (the driver should indicate to the sequencer once it has finished processing each transaction).

3.5 Synchronisation and Timing

Novice users are often unsure how to represent delays in components that are generating stimulus for, or monitoring signals around, the device under test (DUT). This applies to both delays between transactions and within pin-level protocols. It often leads to errors in the synchronisation of sequences, sequencers and drivers.

A typical RTL testbench needs to "wiggle" the pins connected to the DUT. If the DUT connections are part of a standard bus, it has become common to create a "bus functional model" (BFM) that translates requests from the testbench for read and write operations into the corresponding pin-level events across multiple bus clock cycles. The BFM could be implemented as a task but is most likely to be a module with one or more tasks or processes, triggered on the appropriate clock edge. A SystemVerilog testbench is likely to use a class to represent transactions and may also replace the stimulus generator and the BFM modules with a class-based components. In a UVM testbench, the BFM will be an instance of a class derived from `uvm_driver` and the stimulus generator will be an instance of `uvm_sequencer` (or a class derived from `uvm_sequencer`). The UVM driver component will contain one or more tasks to talk to both sequencer and the DUT. An example Verilog BFM is given in Figure 8 and a suitable testbench in Figure 9. A corresponding UVM driver is given in Figure 10.

```
module bfm (input clock, output reg drive, input ack);

  reg [7:0] target;
  event start;

  initial forever begin
    @start;
    @(posedge clock) drive <= 0;
    repeat (8) @(posedge clock) begin
      drive <= target[7];
      target <= {target,1'b0};
    end
    @(posedge clock) drive <= 1;
  end

  task move_forward(input [7:0] target);
  begin
    target = target;
    ->start;
    wait(ack == 1);
  end
  endtask

  task reset;
  ....   /// plus tasks for other operations

Endmodule
```

Figure 8 Verilog BFM

```
module tb();

...

  bfm driver ( .clock(clock),
     .drive_l(drive_l),
     .drive_r(drive_r),
     .ack_l(ack_l),
     .ack_r(ack_r) );

  initial begin
    driver.reset;
    driver.move_forward(8'hFE);
    driver.turn_left;
    driver.move_forward(8'b20);
```

```
    ...
    $finish;
  end
```

Figure 9 Verilog Testbench calling BFM

```
class robot_driver extends uvm_driver #(robot_transaction);

  `uvm_component_utils(robot_driver)

  virtual robot_if.master mi;  //virtual interface

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

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

  task run_phase(uvm_phase phase);
    robot_transaction tr;
    forever
    begin
      seq_item_port.get(tr); //get sequence item from sequencer
      case(tr.op)
        rst:      reset();
        move_fwd: move_forward(tr.target);
        left: turn_left()
        ...
      endcase
    end
 endtask: run_phase

//tasks to drive interface signals (via clocking block)
task reset();
  @(mi.cb) mi.cb.drive <= 1;
endtask: reset

task move_forward(input logic [7:0] target);
  logic target_l = target;
  mi.cb.target <= target_l;
  @(mi.cb) mi.cb.drive <= 0;
    repeat (8) @(mi.cb) begin
      mi.cb.drive <= target_l[7];
      targe_lt <= {target_l,1'b0};
    end
```

```
  @(mi.cb) mi.cb.drive <= 1;
  wait(mi.cb.ack == 1);
endtask

... // plus tasks for other operations

endclass: serial_driver
```

Figure 10 UVM Driver

3.6 Recommended Templates

Doulos has developed "Easier UVM": a set of templates and recommendations to simplify adoption of UVM for new users and to help avoid some of the issues mentioned above. This section summarises some of the main points of Easier UVM. [5]

Easier UVM suggests two general templates that should followed when writing code: Pattern1 for components (see Figure 11) and Pattern 2 for sequences (see Figure 12). There is also a similar template for sequence items.

```
class my_comp extends uvm_component; //or uvm_test/uvm_env/etc
  `uvm_component_utils(my_comp)

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

  ...
Endclass
```

Figure 11 Easier UVM Pattern 1

```
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
```

Figure 12 Easier UVM Pattern 2

The Easier UVM testbench components should be implemented by overriding the phase methods given in Table 1.

build	create child component instances
connect	connect ports to exports on the child components
end_of_elaboration	housekeeping
start_of_simulation	housekeeping
run	runs simulation – may be decomposed into several separate run phases
extract	post-processing
check	post-processing
report	post-processing
final	back stop

Table 1 UVM Phase Methods

The UVM factory should always be used to create components as shown in Figure 13.

```
  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
```

Figure 13 Easier UVM build_phase

The behaviour of a UVM sequence is defined by its body task. This should call `start_item` or `start` to execute sequence items and child sequences respectively.

```
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
```

Figure 14 Easier UVM Sequence Pattern

A user-defined test class overrides the factory (if necessary); creates a configuration object and writes it to the configuration database; creates the UVM environment and finally starts the top-level sequence. A typical test is given in Figure 15.

```
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() );
    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);
    env = my_env::type_id::create("env", this);
  endfunction

  task run_phase(uvm_phase phase);
    //follow template in Figure 7
  endtask
endclass
...
```

Figure 15 Easier UVM Test Class

4. Use of Lesser Known Features to Enhance Verification Capabilities

This section looks at some features of SystemVerilog and UVM that are often overlooked but can nevertheless still be very useful, and are therefore worth exploring further.

4.1 Improvements to random number generation

Repeatability of tests is a key requirement for most users: either they require the sequence of input transactions to be exactly the same every time a test is run or they require a completely different sequence of input transactions for each test run! UVM provides a mechanism to ensure that the random number generator (RNG) of every component gets a seed value that is based on its instance (hierarchical path) name and the initial global seed. This ensures repeatability for the RNG in particular component instances, even if changes are made to other parts of the hierarchy. Sequences are not part of the UVM component hierarchy and are often created without a specified instance name. Repeatability can be ensured by taking care in how sequences are created and providing the seed manually. A simple example starting sequences from a test class is shown in Figure 16. For a more in-depth discussion of the behaviour of the SystemVerilog random number generator see [6]. Note that the random number generator of a UVM sequence item is automatically given a repeatable seed when `create_item` or `start_item` is called by a

sequence (it is based on the hierarchical path name of the sequencer the sequence is running on, including any name given to the sequence instance when it was created).

```
  task run_phase(uvm_phase phase);
    serial_sequence1 seqA, seqB;
    //other objects might be created here
    ...
    seqA = serial_sequence1::type_id::create();
    seqB = serial_sequence1::type_id::create();
    //calling reseed here ensures sequence seeds do not
    //depend on the order in which create is called above
    seqA.reseed();   //or call e.g. seqA.srandom(1234)
    seqB.reseed();
    //randomize control variables in sequences
    void'(seqA.randomize());
    void'(seqB.randomize());
    phase.raise_objection(this);
    fork
      seqA.start(m_env.m_sequencerA);
      seqB.start(m_env.m_sequencerB);
    join
...
```

Figure 16 UVM Sequences with Manual Seed

4.2 Using UVM regular expressions in a checker

The UVM configuration database allows a string containing a regular expression to be used as the path in calls to `get()`. The string matching mechanism that this requires can also be accessed in a user's code: it is made available by the global `uvm_is_match` function. Unfortunately, the UVM documentation does not give many details about how this function should be used so we will explore it further here.

The string matching provided by `uvm_is_match` is based on Posix regular expressions (it uses DPI to call a function from the GNU `regex.h` header). The rules for matching string syntax are summarised below:

- String must begin and end with a forward slash character '/'
- Character classes are surrounded by square brackets (e.g. "[A-Z]" matches any upper case character)
- There are several pre-defined character classes as shown in Table 2 (Note that classes such as Perl's \w, \d, etc are not supported)

[[:upper:]]	upper case	[^[:upper:]]	not upper case
[[:digit:]]	digit	[^[:digit:]]	not digit
[[:lower:]]	lower case	[^[:lower:]]	not lower case
[[:space:]]	space	[^[:space:]]	not space
[[:word:]]	word	[^[:word]]	not word

Table 2 Posix Regular Expression Classes

- Dot character '.' is a wildcard
- Characters may be grouped together within parenthesis "(...)"
- Alternatives within a group are separated by '|'
- Characters or groups may be followed by one of the quantifiers given in Table 3.

{3}	exactly 3
{1,3}	between 1 and 3
{3, }	3 or more
*	0 or more
+	1 or more
?	0 or 1

Table 3 Posix Regular Expression Quantifiers

A search starts at the left-hand side of a string and is "greedy" (if there are wildcards it tries to match as many characters as possible). There are some limitations compared to Perl regular expressions: The most significant is the lack of support for "back-references" within an expression.

Regular expressions can be useful when searching for a particular pattern within a string. Given that it is usual for UVM transactions to define the `convert2string` function (see Figure 17), this opens up interesting possibilities for checkers as demonstrated in Figure 18.

```
class bus_transaction extends uvm_sequence_item;

  typedef enum {BUS_RESET, BUS_WRITE, BUS_READ,
                BUS_FETCH} bus_kind_t;
  function new ( string name = "");
    super.new(name);
  endfunction: new

  // Transaction data contents
  //
  rand logic [ww-1:0] addr;
  rand logic [ww-1:0] data;
  rand bus_kind_t     kind;

  function string convert2string();
    return $sformatf (
      "%s addr=%0d'h%h, data=%0d'h%h",
      kind.name(), ww, addr, ww, data
```

```
    );
  endfunction : convert2string

  //constraints and other member functions
  ...
endclass: bus_transaction
```

Figure 17 Transaction Class with convert2string function

```
class cpu_scoreboard extends uvm_scoreboard;
  ...
  task run_phase(uvm_phase phase);
    bus_transaction tx_cpu, tx_iss;
    forever begin
      iss_fifo.get(tx_iss);
      cpu_fifo.get(tx_cpu);
      if ( uvm_is_match( "/BUS_READ|BUS_FETCH/",
                         tx_iss.convert2string()
                       ) )
      begin
        tx_iss.print();
        if (tx_iss.addr == tx_cpu.addr)
          m_matches++;
        else
          m_mismatches++;
      end
    end
  endtask: run_phase
  ..
endclass: cpu_scoreboard
```

Figure 18 Checker using uvm_is_match

4.3 Making use of C++ Boost library – e.g. enhanced regular expressions using DPI

The `uvm_is_match` function calls a C++ function using the SystemVerilog Direct Programming Interface (DPI). It is easy to enhance the capabilities of UVM by importing other useful C and C++ functions. We will use the Boost regular expression library as an example, This provides enhanced regular expression and match operations that are similar to Perl and include some features that are not supported by `uvm_is_match`.

Calling a C (or C++) library function using DPI may require a C/C++ wrapper function to be defined. This is the case with the `boost::regex_search` function that requires arguments of type `std::string` and `boost::regex`. Fortunately, the `const char*` values

passed from DPI with SystemVerilog strings can be passed directly to the `std::string` and `boost::regex` constructors. The DPI wrapper function `regex_search` is given in Figure 19.

```
#include <string>
#include <boost/regex.hpp>

extern "C"
bool regex_search(const char* re, const char* str) {

  const boost::regex rexp(re);
  const std::string s1 = str;
  return boost::regex_search(s1,rexp);
}
```

Figure 19 C Function Calling boost::regex_search

The SystemVerilog code to work with the imported `regex_search` function is given in Figure 20. In many cases, incorporating existing C/C++ functions simply requires adding their source code files to the command line when the SystemVerilog code is compiled. However, in the case of the Boost regular expression library, this required the library to be built separately (as a shared library) and then added to the VCS command line with the `-sv_lib` option. When building a C/C++ library, it is important to use the C/C++ compiler that is provided with the SystemVerilog simulator (g++ in the case of VCS in a Linux environment).

```
import "DPI-C" function bit regex_search(string re, string str);

class cpu_scoreboard extends uvm_scoreboard;
  ...
  task run_phase(uvm_phase phase);
    ...
    if (regex_search("(\\w{3})_READ",tx_iss.convert2string()))
      ...
```

Figure 20 Calling DPI Function from UVM Checker

4.4 Using strings and UVM HDL access to read/write signals in DUT

It is often convenient in a test, to set or read the value of a variable embedded within a component, at some lower level of the component hierarchy. The exact location of the component might only be known when the test is run, not when the test code is written. UVM provides a solution to this problem with the `find` function that searches for a named component and returns a reference to it. The component name can be a string that is read from the command line. Unfortunately, this mechanism only works for components derived from the `uvm_component` base class and within the UVM environment: it does not work for HDL modules within a test harness or DUT.

The UVM register layer includes a feature that enables direct access to a register within the DUT based on its hierarchical path name (provided as a string). This mechanism may be reused to write or read any variable within the DUT from a UVM test, using a string supplied at run-time, as mentioned above. The `uvm_check_hdl_path` should be called as a first step to make certain that the string received matches an object at the path specified. If it does, `uvm_hdl_deposit` may be called to set the value of a variable or wire while `uvm_hdl_read` may be called to read its current value. Examples demonstrating the use of these functions are given in Figure 21 and Figure 22.

```
function void end_of_elaboration_phase(uvm_phase phase);
  string arg_value;
  int value_i;
  string hdl_path_string;
  if (uvm_cmdline_proc.get_arg_value(
                   "+hdl_path_var_name=",
                   hdl_path_string) )
    if (uvm_cmdline_proc.get_arg_value(
                   "+hdl_path_var_value=", arg_value))
      value_i = arg_value.atoi();


  if (uvm_hdl_check_path(hdl_path_string) )
        `uvm_info("ENV",{"HDL path ",hdl_path_string,
                         " found"},UVM_HIGH)
  else
        `uvm_warning("ENV",{"HDL path ",hdl_path_string,
                         " not found!!"});

  assert( uvm_hdl_deposit(hdl_path_string,value_i));

endfunction: end_of_elaboration_phase
```

Figure 21 Using uvm_hld_deposit to configure DUT

```
task run_phase(uvm_phase phase);
  word_t rx_count;
  string hdl_path_string;
  #100;
  if (uvm_cmdline_proc.get_arg_value(
                   "+hdl_path_var_name=",hdl_path_string) )
    if ( uvm_hdl_read(hdl_path_string, rx_count) )
      `uvm_info("ENV",$sformatf("rx_count = %0b",
                                rx_count),UVM_LOW)
```

```
  endtask: run_phase
```

Figure 22 Using uvm_hdl_read in monitor

There are restrictions on what can be done with the values returned with `uvm_hdl_read` (such as not being able to use them to trigger processes). For an in-depth discussion, see [7].

4.5 Intercepting messages with UVM Report Catcher

UVM includes a highly configurable message reporting system – messages may be suppressed or their actions overridden, from a test or even the command line. However, sometimes it is useful to be able to perform additional actions when a particular message is produced, or to have greater control over which messages should be reconfigured. The `uvm_report_catcher` class makes it simple to implement these features. It is based on the standard UVM callback mechanism and is used as the base class for a customised report catcher. The derived class should override its `catch` function to determine if any action is required to modify the behaviour for generated errors. This can be applied to report handlers with a specified path or globally (by setting the first argument to the `uvm_report_cb::add` function to `null`). The example given in Figure 23 uses the `uvm_is_match` function to test if the message string is matched by the given regular expression.

```
class catch_fatals extends uvm_report_catcher;
  function new(string name="catch_fatals");
    super.new(name);
  endfunction

  function action_e catch();
    string filter;
    //This changes all BUS_RESET, BUS_READ and BUS_FETCH
    //fatal errors to an info message
    if(get_severity() == UVM_FATAL &&
                         (uvm_is_match(
                             "/BUS_RESET|BUS_READ|BUS_FETCH/",
                             get_message() )
                         )
      )
      set_severity(UVM_INFO);
    return THROW;
  endfunction
endclass: catch_fatals

class my_test1 extends uvm_test;

  catch_fatals m_catcher = new;

  function void start_of_simulation_phase(uvm_phase phase);
    // To affect all reporters, use null for the object
```

```
    uvm_report_cb::add(null, m_catcher);
  endfunction: start_of_simulation_phase

 ...
endclass: my_test1
```

Figure 23 Use of uvm_report_catcher to change severity of matched messages

Sometimes, there might be a requirement to modify the behaviour of a particular message that is not known until a simulation is run. In these situations, a regular expression could be used to check each message as it was generated. The regular expression matching string could be set dynamically every time a simulation is run, from a command-line argument, as shown in Figure 24.

```
class catch_fatals extends uvm_report_catcher;
  function new(string name="catch_fatals");
    super.new(name);
  endfunction
  function action_e catch();
    string filter;
    //This changes all messages that contain search
    //pattern in catch_fatal plusarg to info
    if (uvm_cmdline_proc.get_arg_value("+catch_fatal=",filter) )
    begin
      if(get_severity() == UVM_FATAL &&
                          (uvm_is_match(filter,get_message())) )
        set_severity(UVM_INFO);
    end
    return THROW;
  endfunction
endclass: catch_fatals
```

Figure 24 Report Catcher using Regular Expression and Command Line Argument

The report catcher can also be extended to pause a simulation whenever a particular message is detected. The example in Figure 25 reads a string as a command line "plusarg" which it tests against the message string using `uvm_is_match`.

```
class catch_stop extends uvm_report_catcher;
  function new(string name="catch_stop");
    super.new(name);
  endfunction
  function action_e catch();
    string filter;
    //This stops simulation for all messages that
    //contain search pattern in stop_on plusarg
```

```
    if (uvm_cmdline_proc.get_arg_value("+stop_on=",filter) )
    begin
      if(uvm_is_match(filter,get_message()))
        set_action(UVM_DISPLAY + UVM_STOP);
    end
    return THROW;
  endfunction
endclass: catch_stop
```

Figure 25 Report Catcher to Pause Simulation using Command Line Arguments

4.6 Uses of dynamic processes

The components in a typical Verilog or VHDL testbench often contain multiple processes, executing concurrently. UVM novices are often confused about how to implement equivalent behaviour using just the standard UVM phases. The trick is to make use of SystemVerilog dynamic processes within a UVM component that are "spawned" from the appropriate UVM run-time phase.

Figure 26 shows a version of the Verilog BFM from Figure 8 that drives two output channels in parallel (it is for a controller that has separate channels to drive two motors using a common clock but separate handshaking). The UVM driver that replaces the Verilog BFM is given in Figure 27. It spawns a dynamic process do drive each channel. There is a `wait fork` statement at the end of the loop in the `run_phase` – this waits for all spawned processes to have completed before getting the next sequence item from the sequencer.

```
module bfm ( input clock,
   output reg drive_l,
   output reg drive_r,
   input ack_l,
   input ack_r);

  reg [7:0] target_l, target_r;
  event start_l, start_r;

  initial forever begin
    @start_l;
    @(posedge clock) drive_l <= 0;
    repeat (8) @(posedge clock) begin
      drive_l <= target_l[7];
      target_l <= {target_l,1'b0};
    end
    @(posedge clock) drive_l <= 1;
  end

  initial forever begin
    @start_r;
```

```
    @(posedge clock) drive_r <= 0;
    repeat (8) @(posedge clock) begin
      drive_r <= target_r[7];
      target_r <= {target_r,1'b0};
    end
    @(posedge clock) drive_r <= 1;
  end

  task move_forward(input [7:0] target);
  begin
    target_l = target;
    target_r = target;
    ->start_l;
    ->start_r;
    fork
      wait(ack_l == 1);
      wait(ack_r == 1);
    join
  end
  endtask

  ...

endmodule
```

Figure 26 Verilog BFM with Multiple Processes

```
class robot_driver extends uvm_driver #(robot_transaction);

...

  task run_phase(uvm_phase phase);
    robot_transaction tr;
    forever
    begin
      seq_item_port.get(tr);
      case(tr.op)
        rst:      reset();
        move_fwd: move_forward(tr.target);
        left: turn_left()
        ...
      endcase
      wait fork; //wait before getting next seq item
    end
 endtask: run_phase
```

```
  task reset();
    @(mi.cb);
    mi.cb.drive_l <= 1;
    mi.cb.drive_r <= 1;
  endtask: reset

  task move_forward(input logic [7:0] target);

    logic [7:0] target_l, target_r;
    target_l = target;
    target_r = target;
    @(mi.cb);
    mi.cb.drive_l <= 0;
    mi.cb.drive_r <= 0;
    fork
    begin  //drive_l
      repeat (8) @(mi.cb) begin
        mi.cb.drive_l <= target_l[7];
        target_l <= {target_l,1'b0};
      end
      @(mi.cb) mi.cb.drive_l <= 1;
      wait(mi.cb.ack_l == 1);
    end
    begin  //drive_r
      repeat (8) @(mi.cb) begin
        mi.cb.drive_r <= target_r[7];
        target_r <= {target_l,1'b0};
      end
      @(mi.cb) mi.cb.drive_r <= 1;
      wait(mi.cb.ack_r == 1);
    end
    join_none
endtask

...

endclass: serial_driver
```

Figure 27 UVM Driver with Multiple Processes in run_phase

5. Conclusions

We have highlighted some common errors made by new users and suggested some ways to avoid them. At the end of the day, there really is no substitute for spending time to learn SystemVerilog and UVM properly, but hopefully this paper has provided some guidance to help those who need to start using SystemVerilog and UVM immediately, until they are able to follow a more

formal training course. The section on lesser-known features has hopefully given readers who have already started using UVM, an insight into some areas where they could improve their simulation environment.

6. References

[1] Chris Spear and Greg Tumbush, "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features", 3rd Edition, pub. Springer 2012, ISBN: 978-146140714-0

[2] Stuart Sutherland and Don Mills, "Verilog and SystemVerilog Gotchas: 101 Common Coding Error and How to Avoid Them", pub. Springer 2006, ISBN: 978-038771714-2

[3] Adam Erickson, "Are OVM & UVM Macros Evil? A Cost-Benefit Analysis", DVCon 2011

[4] John Aynsley, " The Finer Points of UVM: Tasting Tips for the Connoisseur", DVCon 2013

[5] John Aynsley, " Easier UVM for Functional Verification by Mainstream Users ", DVCon 2011

[6] Doug Smith, "Random Stability in SystemVerilog", SNUG Austin, 2013

[7] Jonathan Bromley, " I Spy with My VPI: Monitoring signals by name, for the UVM register package and more ", SNUG Munich, 2012