World Class Verilog & SystemVerilog Training

UVM Message Display Commands
Capabilities, Proper Usage and Guidelines

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com
www.sunburst-design.com

ABSTRACT

UVM message display commands offer great flexibility in printing of UVM messages, but their usage is frequently misunderstood. In fact, the first printings of two respected UVM texts released in 2013 as well as the UVM Users Guide and UVM Reference Manual all either incorrectly describe UVM verbosity, incorrectly use UVM verbosity settings in examples, or both.

Many new users incorrectly assume the built-in verbosity settings represent printing priority, but this is exactly backwards from reality.

With so many respected resources offering false or ill-advised guidelines regarding the use of UVM verbosity, it is time to set the record straight, give a correct description of UVM verbosity and suggest important UVM verbosity usage guidelines.

This paper details strategies and guidelines for proper usage of UVM display commands and the built-in method `convert2string()`.

Table of Contents

Table of Examples

1. Introduction

UVM verbosity settings are ***NOT*** message priority settings!

Lest anyone fell asleep while reading the above text, I repeat,

UVM verbosity settings are NOT message priority settings!

UVM verbosity settings ***are the opposite*** of message priority settings.

A **`uvm_info** message labeled with **UVM_LOW** does not mean it is a low priority message that should be rarely displayed. In fact **UVM_LOW** info-messages are some of the highest priority messages and they are very difficult to disable.

This paper will properly describe UVM verbosity theory and recommend proper usage of UVM verbosity settings.

1.1. VCS version and execution command

For this paper, all examples were run using VCS version H-2013.06-SP1, which includes UVM library version 1.1d.

Compilations with a command file named **run.f** were done using the command:

```
vcs -sverilog -ntb_opts uvm -timescale=1ns/1ns -f run.f
```

Simulations were executed using the command:

```
simv +UVM_TESTNAME=<testname> <other + options as noted>
```

2. Verilog $display

Verilog engineers have customarily used the **$display** command to show information and report results in a verification environment. Verilog engineers have frequently added additional debugging display commands surrounded by **`ifdef DEBUG** to turn on and off additional debug messages. Controlling the display and debug-display of messages has been an annoying and non-trivial task that frequently required that a design be re-compiled between simulation runs.

UVM has a rich set of message-display commands and methods to change the number and types of messages that are displayed without re-compilation of a design. The same message mechanism also includes the ability to mask or change the severity of the message to adapt to the required verification environment.

Unfortunately there is significant confusion on how the messages work and how the message capabilities should be used. The confusion has been compounded by the release of two very good UVM books in 2013 that improperly use the UVM messaging capabilities.

This paper will clarify the definitions of the UVM messaging capabilities and offer proven recommendations to properly use those capabilities.

3. UVM reporting arguments

Many of the reporting commands take one or more of the following reporting arguments: **uvm_severity**, **uvm_action** and **uvm_verbosity**. A brief description of these arguments and their legal values is shown in the following three subsections.

3.1. uvm_severity definitions

There are four **uvm_severity** reporting definitions: **UVM_INFO**, **UVM_WARNING**, **UVM_ERROR** and **UVM_FATAL**.

These severity definitions correspond to the UVM messages and macros by the same name.

3.2. uvm_action definitions

There are eight **uvm_action** reporting definitions and a description of how these actions work can be found in the UVM-1-2 Class Reference Manual. That description is shown below.

> *Defines all possible values for report actions. Each report is configured to execute one or more actions, determined by the bitwise OR of any or all of the following enumeration constants.*
>
> | | |
> |---|---|
> | *UVM_NO_ACTION* | *No action is taken* |
> | *UVM_DISPLAY* | *Sends the report to the standard output* |
> | *UVM_LOG* | *Sends the report to the file(s) for this (severity,id) pair* |
> | *UVM_COUNT* | *Counts the number of reports with the COUNT attribute. When this value reaches max_quit_count, the simulation terminates* |
> | *UVM_EXIT* | *Terminates the simulation immediately.* |
> | *UVM_CALL_HOOK* | *Callback the report hook methods* |
> | *UVM_STOP* | *Causes $stop to be executed, putting the simulation into interactive mode.* |
> | *UVM_RM_RECORD* | *Sends the report to the recorder* |

Of the above action settings, only **UVM_NO_ACTION** will be described and used in this paper. The above **uvm_action** documentation from the UVM Class Reference is included to make the reader aware of other possible actions.

3.3. UVM_VERBOSITY definitions

There are predefined UVM verbosity settings built into UVM (and OVM). These settings are included in the UVM **src/uvm_object_globals.svh** file and the settings are part of the enumerated **uvm_verbosity** type definition. The settings actually have integer values that increment by 100 as shown below.

```
UVM_NONE   = 0       (highest priority messages)
UVM_LOW    = 100
UVM_MEDIUM = 200
UVM_HIGH   = 300
UVM_FULL   = 400
UVM_DEBUG  = 500     (lowest priority messages)
```

By default, when running a UVM simulation, all messages with verbosity settings of **UVM_MEDIUM** or lower (**UVM_MEDIUM**, **UVM_LOW** and **UVM_NONE**) will print.

To display more verbose simulations and the corresponding messages with verbosity settings of **UVM_HIGH**, **UVM_FULL** and **UVM_DEBUG**, a new verbosity setting must be specified on the simulation command line or using a verbosity setting embedded within one of the test components.

The developers of OVM/UVM wisely chose verbosity settings with corresponding integer values spaced by units of 100. Incrementing by 100 between the defined verbosity settings allows future versions of UVM to add verbosity settings between the settings defined in all versions of OVM and UVM through UVM version 1.2.

4. How to change verbosity settings

There are two principal approaches to select or modify verbosity settings: (1) command line switches, and (2) coded method calls.

4.1. Verbosity command line switches

The primary advantage of using command line switches is that if the design has been compiled into a simulation executable, such as the VCS **simv** executable, the design can be re-simulated using a different verbosity setting without re-compiling the design and testbench.

The most common option is the command line switch:

+UVM_VERBOSITY=<***verbosity***>

Where <***verbosity***> is one of the following: **UVM_NONE**, **UVM_LOW**, **UVM_MEDIUM**, **UVM_HIGH**, **UVM_FULL**, **UVM_DEBUG**, and the default setting is **UVM_MEDIUM**.

After compiling the design and testbench, to run a VCS simulation with a verbosity setting that produces more messages (**UVM_HIGH** or lower), execute the command:

```
simv +UVM_VERBOSITY=UVM_HIGH
```

Additional command line options to target the verbosity of sepcific components are shown below. If the output from a simulation seems to be excessively verbose, the simulation can be re-run with command line switches to filter some of the noise that appears on the screen.

```
+uvm_set_verbosity=<comp>,<id>,<verbosity>,<phase>
+uvm_set_verbosity=<comp>,<id>,<verbosity>,time,<time>
```

For example, to change the verbosity to **UVM_LOW** for the **uvm_test_top.e.agnt.drv** (**tb_driver**) in the **run** phase, which would suppress all messages with a verbosity of **UVM_MEDIUM** or higher, execute the command:

```
simv +UVM_TESTNAME=test1
+uvm_set_verbosity=uvm_test_top.e.agnt.drv,DRIVER,UVM_LOW,run
```

You could also change the verbosity to **UVM_LOW** for all **<*id*>** values and for all components in the **tb_agent** using the command:

```
simv +UVM_TESTNAME=test1
+uvm_set_verbosity=uvm_test_top.e.agnt.*,_ALL_,UVM_LOW,run
```

Note that the **run_phase()** is specified as **run** and not **run_phase** (the latter does not work).

Quoting from the UVM-1.2 Class Reference Manual:

> *<these switches> allow the users to manipulate the verbosity of specific components at specific phases (and times during the "run" phases) of the simulation. The id argument can be either _ALL_ for all IDs or a specific message id.*

The UVM-1.2 Class Reference Manual also explains that use of wild cards to select multiple message **<*id*>** values is not supported due to concerns over tool performance.

To be clear, wild cards are supported for selecting components. All of the following are supported:

```
simv +UVM_TESTNAME=test1
+uvm_set_verbosity=uvm_test_top.e.agnt.drv,_ALL_,UVM_LOW,run

simv +UVM_TESTNAME=test1
+uvm_set_verbosity=uvm_test_top.e.agnt.*,_ALL_,UVM_LOW,run

simv +UVM_TESTNAME=test1 +uvm_set_verbosity=*.agnt.*,_ALL_,UVM_LOW,run
```

Since your tests will be recognized as **uvm_test_top**, and since you frequently use the same environment, wild card usage is often a convenient shorthand to select components under the test/environment as shown in the last simulation command above.

The UVM-1.2 Class Reference Manual further clarifies argument priorities by noting:

Settings for non-"run" phases are executed in order of occurrence on the command line. Settings for "run" phases (times) are sorted by time and then executed in order of occurrence for settings of the same time.

Another command line option to target the messages of sepcific components is shown below. If the output from a simulation seems to be excessively verbose, the simulation can be re-run with switches to filter some of the noise that appears on the screen.

```
+uvm_set_action=<comp>,<id>,<severity>,<action>
```

For example, to suppress all messages reported from the **uvm_test_top.e.agnt.drv** (**tb_driver**), execute the command:

```
simv +UVM_TESTNAME=test1
+uvm_set_action=uvm_test_top.e.agnt.drv,_ALL_,_ALL_,UVM_NO_ACTION
```

-or-

```
simv +UVM_TESTNAME=test1 +uvm_set_action=*.drv,_ALL_,_ALL_,UVM_NO_ACTION
```

As another example, to suppress all **UVM_INFO** messages reported from the **uvm_test_top.e.agnt.drv** (**tb_driver**), but still allow **UVM_WARNING**, **UVM_ERROR** and **UVM_FATAL** messages to be displayed, execute the command:

```
simv +UVM_TESTNAME=test1 +uvm_set_action=*.drv,_ALL_,UVM_INFO,UVM_NO_ACTION
```

As shown in the preceding examples, the _**ALL**_ UVM keyword can be used for all <***id***> and for all <***severity***> values.

4.2. Changing verbosity using method calls

The **uvm_component** base class includes a hierarchical reporting interface with **set_report_*** methods that are applied to a component or recursively to the specified component and all child subcomponents.

The primary advantage of using verbosity method calls is that it is relatively simple to extend a test (or another testbench component) and include verbosity method calls that selectively change verbosity settings for specific components or entire component hierarchies.

Most of the following information is taken nearly verbatim from the UVM-1.2 Class Reference Manual and more details can be found in that Reference Manual.

The following methods recursively apply the specified verbosity to messages that include the given severity, id, or severity-id pair.

```
set_report_id_verbosity_hier
set_report_severity_id_verbosity_hier
```

The following methods recursively apply the specified action to messages that include the given severity, id, or severity-id pair.

```
set_report_severity_action_hier
set_report_id_action_hier
set_report_severity_id_action_hier
```

The following methods recursively apply the specified FILE descriptor to messages that include the given severity, id, or severity-id pair.

```
set_report_default_file_hier
set_report_severity_file_hier
set_report_id_file_hier
set_report_severity_id_file_hier
```

The following method recursively sets the maximum verbosity level to messages for this component and all those below it.

```
set_report_verbosity_level_hier
```

5. UVM messages and message-macros

There are two forms of message-display commands included in UVM: built-in UVM message-methods and UVM message-macros. Although there is disagreement between simulation vendors on the usefulness and efficiency of some of the UVM macros [1], there is universal agreement that UVM-message macros should be used over UVM message methods.

The interesting fact is that UVM message-macros are more simulation efficient than the UVM message methods even though they call the built-in UVM message methods. How is this possible? The UVM message methods perform time consuming string processing on the user-specified strings, whether the message will be displayed or not. The UVM message macros first check simulation verbosity settings to determine if the string will be printed and if the message will not be printed, the message method is not called; hence, the expensive string processing is avoided.

6. UVM Message Guidelines

(1) Quit using the **`$display`** command!
(2) Use the message macros, not the message methods.
(3) Use **`` `uvm_info("id", "msg", UVM_NONE) ``** for only the most important messages that should NEVER be filtered, such as test-passing messages.
(4) Use **`` `uvm_info("id", "msg", UVM_LOW) ``** for only very important messages that should be rarely filtered. These might include block-level test-passing messages.
(5) Use **`` `uvm_info("id", "msg", UVM_MEDIUM) ``** as your new default **`$display`** command. These messages will always be displayed by default but are easily disabled.

(6) Use **`uvm_info("id", "msg", UVM_HIGH)** to display information that should only be shown occasionally.
(7) Use **`uvm_info("id", "msg", UVM_FULL)** to display design status and UVM testbench status messages.
(8) Use **`uvm_info("id", "msg", UVM_DEBUG)** to display debug messages added to the design or UVM testbench.
(9) Use the **`uvm_fatal("id", "msg")**, and **`uvm_error("id", "msg")**, macros as appropriate - consider these to be non-maskable messages.
(10) Use the **`uvm_warning("id", "msg")** macro very sparingly. Unfortunately, these warnings cannot be disabled using verbosity settings.
(11) Add the UVM-standard **convert2string()** method to all of your transaction data classes.
(12) Project and IP providers should implement an intelligent "ID" scheme to help modify severities and mask unwanted messages.

Each of these guidelines will be discussed in detail in this paper.

Who knew message guidelines could be so complex! Following these guidelines will save time and trouble later in the project since most projects eventually try to apply some form of intelligent message mechanism after it is difficult to implement a forward thinking approach.

7. Simulation reporting goals

To properly take advantage of the UVM verbosity capabilities, it is important to set goals that are related to different phases of testing and simulation. Below are suggested goals and recommendations regarding the use of UVM verbosity.

7.1. Initial testing

When executing initial testing of the design and testbench, it is sometimes useful to see proper completion of each UVM phase. These are basically status messages to show that the testbench has been properly assembled and is functioning as expected. UVM phase status messages should have been added using a verbosity setting of **UVM_FULL**, to ensure that these messages are off by default and only on when doing early testing of the verification environment by using the simulation command line switch:

```
+UVM_VERBOSITY=UVM_FULL
```

UVM_FULL is level 400 and should be used to increase report verbosity by showing UVM phase status information as well as both failing and passing transaction information.

7.2. Early testing

When doing early testing of both the design and testbench, it is often desirable to see messages about passing transactions as well as failing transactions. Viewing reports about passing transactions gives greater confidence that the design and testbench are performing as expected. Showing passing transaction information will significantly increase the volume of messages displayed during simulation, so this should not be the default mode. One should be able to turn on these messages by using the simulation command line switch:

```
+UVM_VERBOSITY=UVM_HIGH
```

UVM_HIGH is level 300 and should be used to increase report verbosity by showing both failing and passing transaction information, but does not show annoying UVM phase status information after it has been established that the UVM phases are working properly.

7.3. Routine testing

Routine testing includes messages that you would normally display using a Verilog **$display** command.

```
+UVM_VERBOSITY=UVM_MEDIUM
```

UVM_MEDIUM is level 200 and should be used to as the default $display command. If an engineer does not select a verbosity setting, these messages will print by default (roughly equivalent to a default **$display** command). This verbosity setting should not be used for any debugging messages or for standard test-passing messages.

7.4. Block-level regression testing

Any display macro with a verbosity setting of **UVM_LOW** should be considered a message that is difficult to turn off, and thus should be reserved for only very important messages. Turning off **UVM_LOW** info messages can typically only be accomplished by setting a command-line verbosity setting of 99 or less, and there is only one standard verbosity setting that meets this criteria and that is **UVM_NONE**. This is why most of the examples in existing texts that include **UVM_LOW** as a verbosity setting are so wrong! It is hard to "silence" **UVM_LOW** messages.

The other way to disable **UVM_LOW** info messages is to trap the messages and change their verbosity setting in each test, which is a tedious process.

```
+UVM_VERBOSITY=UVM_LOW
```

UVM_LOW is level 100 and should be used to reduce report verbosity and only shows important messages

7.4.1. Deep regression testing

Any display macro with a verbosity setting of **`UVM_NONE`** cannot be turned off except by intercepting the message the disabling it.

Another common misconception about UVM is related to tests extended from the uvm_test base class. Engineers often ask, "how do I run multiple tests (extended from uvm_test)?" the answer is it cannot be done. You can only run one test from the command line using **`+UVM_TESTNAME="`***testname_string***`"`**

UVM tests, extended from the uvm_test base class, should be considered regression suites. Users should also consider creating three or more levels of sequences, extended from the uvm_sequence base class, following these simplified guidelines:
(1) Level 1 sequences: Create basic tests that run a few transactions (extended from the uvm_sequence_item base class) using sequences.
(2) Level 2 sequences: Create larger tests that run Level 1 sequences. A Level 2 sequence is a sequence of sequences (Level 1 sequences). Level 2 sequences should be considered "tests" and multiple Level 2 sequences can be called from a top-level test. This is how you can run multiple tests. A Level 2 sequence is what has traditionally been considered a test by Verilog verification engineers.
(3) Level 3 sequences: Create block-level or system-level regression suites of Level 2 Sequences using a Level 3 sequence. Level 3 sequences should be considered "regression suites" and one or more Level 3 sequences can be called from a top-level test to make execution of a regression suite possible by using the standard **`+UVM_TESTNAME`** simulation switch. These regression suites should use the non-maskable verbosity setting of **`UVM_NONE`** to show pass/fail status of each Level 3 regression suite.

```
+UVM_VERBOSITY=UVM_NONE
```

`UVM_NONE` is level 0 and should be used to reduce report verbosity to a bare minimum of vital simulation regression suite messages.

8. The Verilog $display command should not be used in UVM

The Verilog **`$display`** command is simple but inadequate for proper message strategies.

One source suggested that **`$display`** only be used for table headers. I believe this is wrong. Using the **`$display`** command will indeed print table headers but the table headers cannot be turned off using verbosity settings.

Guideline: Use **`` `uvm_info() ``** to print table headers. Use the id-string **`"HDR"`** and use the same verbosity setting that is used to print the table entries.

If the table entries use a verbosity setting of **`UVM_HIGH`**, one should print the table headers using **`UVM_HIGH`**. If the table entries use **`UVM_LOW`**, print the table headers using **`UVM_LOW`**. Using the same verbosity setting for both table headers and table entries will cause headers and table entries to print or not-print in a coordinated fashion.

Guideline: Quit using the **`$display`** command in UVM testbenches.

`$display` is a non-maskable print command that requires recompilation of a design to disable or enable. The **`$display`** command should be considered a forbidden command in UVM verification environments.

9. Changing the verbosity of simulations

One of the advantages offered by UVM is the ability to re-run a simulation with more or fewer simulation messages without the need to re-compile the design or testbench. The command line option that is used to re-run a simulation without re-compiling the design is the **`+UVM_VERBOSITY=UVM_`***<verbosity_setting>*

9.1. Verbosity versus priority

With UVM messages, the user specifies verbosity, not priority! The user specifies just how verbose the message reporting should be and which messages should be included with each simulation run. Users do not directly specify message priority.

Many new UVM users mistakenly believe that **`UVM_HIGH`** means that a message has high priority and that **`UVM_LOW`** means that a message has low priority, which is exactly backwards from reality.

Verbosity refers to the number of messages that should be reported during a simulation. Highly verbose (**`UVM_HIGH`**) simulations would display a large number of messages, while minimally verbose (**`UVM_LOW`**) simulations would show only a small number of messages.

10. Message catch & throw

There are times when Verification IP includes code that issues a **`` `uvm_fatal ``** message that will abort the simulation. Although the message is both correct and useful, it may be appropriate to display the message without aborting the simulation. One example would be a test-sequence that is checking that an error occurred when certain error conditions were issued. In examples like this, the verification engineer would like to trap the fatal error and demote the message to a **`` `uvm_error ``** that would continue to report the error without aborting the test-sequence.

UVM provides a built-in callback mechanism to demote or modify specified messages. The user can define a test-demoter class that is an extension of the **`uvm_report_catcher`** class. The

extended demoter class overrides the **catch()** method of the **uvm_report_catcher** class and this method returns an enumerated variable of the **action_e** type. The **action_e** values that can be returned are either **CAUGHT** or **THROW**.

The **uvm_report_catcher** has many builtin methods and the reader should examine the UVM Class Reference Manual to discover all of the capabilities, but this paper will show examples using some of the more common **uvm_report_catcher** methods.

To test the current message state inside of the **catch()** method that is inside each registered report catcher, you can use the following methods:

```
get_severity    Returns the uvm_severity of the message that is currently
                being processed.
get_verbosity   Returns the verbosity of the message that is currently
                being processed.
get_id          Returns the string id of the message that is currently
                being processed.
get_message     Returns the string message of the message that is currently
                being processed.
```

Other message-testing methods include: **get_context**, **get_client**, **get_action**, **get_fname**, **get_line** and **get_element_container**.

Each **catch()** method can execute the following methods:

```
set_severity    Change the severity of the message to severity.
set_verbosity   Change the verbosity of the message to verbosity.
set_message     Change the text of the message to message.
set_action      Change the action of the message to action.
```

Other execution methods include: **set_id**, **set_context**, **add_int**, **add_string** and **add_object**.

There are also a number of debug and reporting methods, including: **uvm_report_fatal**, **uvm_report_error**, **uvm_report_warning**, **uvm_report_info**, **uvm_report**, **issue** and **summarize** .

The first report catcher example is a demoter that catches any **UVM_FATAL** message and uses the **set_severity()** method to change the caught message and reissue the message with **UVM_ERROR** severity.

```
class test1_demoter extends uvm_report_catcher;
  ...

  function action_e catch();
    if(get_severity() == UVM_FATAL) begin
      set_severity(UVM_ERROR);
      `uvm_info("demoter", "Caught FATAL / demoted to ERROR", UVM_MEDIUM)
    end
    //return CAUGHT;
```

```
    return THROW;
  endfunction
endclass
```

Example 1 - test1_demoter class example

As shown in the **test1_demoter** class of Example 1, the **get_severity()** command is intercepting **UVM_FATAL** messages, changing the severity using the **set_severity(UVM_ERROR)** method call, printing a "**… FATAL / demoted …**" message and then throwing the new message for display (**return THROW**). It is not required to print a message using the **`uvm_info()** macro, in which case the test-and-set command would simply take the form:

```
if(get_severity() == UVM_FATAL) set_severity(UVM_ERROR);
```

The above demoter is demoting all **UVM_FATAL** messages, but it is also possible to select fatal messages from specific components by further qualifying the **get_severity()** **if**-test by adding a **get_id()** test as shown below:

```
if(get_severity() == UVM_FATAL && get_id() == "AGENT")
        set_severity(UVM_ERROR);
```

There are many industry examples where the **demoter** is placed in the **top** module, but I prefer to add the **demoter** to extended test classes, making it possible to run the original test with the fatal-abort action, and then run the extended class that **catch**-es the **UVM_FATAL** message and performs a **throw**-action to demote the **UVM_FATAL** to a **UVM_ERROR** of the same message.

```
class test1x extends test1;
  ...
  // env e; // inherited from the test1 class
  test1_demoter demoter;
  ...

  function void build_phase(uvm_phase phase);
    demoter = test1_demoter::type_id::create("demoter");
    uvm_report_cb::add(e, demoter);
    super.build_phase(phase);
  endfunction
endclass
```

Example 2 - Extended class example with demoter enabled

As shown in the extended **test1x** class of Example 2, a **test1_demoter** **demoter** handle is declared at the top of the class, then in the **build_phase()** method, the **demoter** is factory-created followed by a call to the **uvm_report_cb::add(e, demoter)** static method to add the **demoter** to the environment using the **e** handle inherited from the **test1** class. The fully coded **test1x.sv** file is shown in Example 16 in the Appendix.

For reasons that I do not fully understand, the **demoter** in the examples of my extended test class required the creation of the demoter and a call to the **uvm_report_cb()** method BEFORE calling **super.build_phase()**. Calling **super.build_phase()** first or deleting the call to **super.build_phase()** did not work.

Perhaps Verification IP only reports warnings or errors and continues the simulation when the verification engineer would like to promote the warnings or errors to abort the simulation with a fatal error.

The second report catcher of Example 3 uses the **`get_severity()`** and **`get_id()`** state tests to catch any **`UVM_WARNING`** message in the **`tb_agent`** with <***id***> **`"AGENT"`** and uses the **`set_severity()`** and **`set_message()`** methods to change the caught message and reissue the message with **`UVM_ERROR`** severity and **`"Caught AGENT WARNING / promoted to ERROR"`** message.

```
class test_report_catcher extends uvm_report_catcher;
  ...

  // This example promotes "AGENT" warnings to error messages
  function action_e catch();
    if(get_severity() == UVM_WARNING && get_id() == "AGENT") begin
      set_severity(UVM_ERROR);
      set_message("Caught AGENT WARNING / promoted to ERROR");
    end
    return THROW;
  endfunction
endclass
```

Example 3 - test_report_catcher class example

The extended report catcher **`catch()`** method must either **`return(THROW)`**, which basically executes the message per the settings in the **`catch()`** method, or **`catch()`** must **`retun(CAUGHT)`**, which appears to show how many messages of each type (FATAL, ERROR and WARNING) are observed by the simulation, but then halts report processing with the first **`return(CAUGHT)`** command executed. At the time of this writing, I had not found a compelling reason to use the **`return(CAUGHT)`** command.

11. Using get_type_name()

It is a somewhat common practice to display messages with an <***id***> set to **`get_type_name()`**. This may be a verbose way to request information that is already provided by default with message macros.

The **`get_type_name()`** macro returns the class type name, which is often added in abbreviated form by the user in the **`` `uvm_info ``** messages, but all of the message macros already return the full test path name by default, so the **`get_type_name()`**, or the other common name-type mthod calls, **`get_name()`** and **`get_full_name()`**, offer little additional information or advantage to the verification engineer while debugging.

Consider the slightly modified **`start_of_simulation_phase()`** method from the **`env`** class shown in Example 8 of the Appendix. The modification is that four **`` `uvm_error ``** messages are

displayed, the first with a user defined <***id***>, and the remaining three with builtin "**`get_*_name()`**" method calls. The default output from this simulation run shows the four **`env`** error messages and all four already include the same information as the **`get_full_name()`** method call.

```
function void start_of_simulation_phase(uvm_phase phase);
  `uvm_error("ENV", "env error")
  `uvm_error(get_name(), "env error")
  `uvm_error(get_full_name(), "env error")
  `uvm_error(get_type_name(), "env error")
endfunction

UVM_ERROR env.sv(15) @ 0: uvm_test_top.e [ENV] env error
UVM_ERROR env.sv(16) @ 0: uvm_test_top.e [e] env error
UVM_ERROR env.sv(17) @ 0: uvm_test_top.e [uvm_test_top.e] env error
UVM_ERROR env.sv(18) @ 0: uvm_test_top.e [env] env error
```

Example 4 - get_*_name() method calls and printed output displays

Adding **`get_type_name()`** to message macros is a somewhat verbose way to request information that is largely already displayed. I typically discourage using **`get_type_name()`** as the <***id***> field of a message macro.

12. Conditional verbosity printing

A useful debugging strategy is to display the entire UVM testbench structure and factory configuration before entering the test-**`run_phase`**(s). The ideal place to display this information is after the test has been built, connected and elaborated, or in other words, in the test's **`start_of_simulation_phase()`** method. The **`start_of_simulation_phase()`** method is basically a pre-**`run`** phase that runs after the **`build_phase()`**, **`connect_phase()`** and **`end_of_elaboration_phase()`** have completed for the entire testbench construction.

The verification engineer will not want this information to be routinely displayed during regression runs, after the testbench structure is working, so it is best to only display this information when a verbosity setting of **`UVM_HIGH`** or higher is selected.

Calling **`this.print`** from the test causes an unconditional display of the entire UVM testbench structure, while the **`this.sprint`** command can be called from **`` `uvm_info``** with a corresponding verbosity setting: **`` `uvm_info("TCFG", this.sprint(), UVM_HIGH)``**

There is also a **`factory.print`** command that causes an unconditional display of the components, transactions and sequences registered with the factory, including any active overrides[3] but there is no corresponding **`factory.sprint`** command that could be put under verbosity control from a **`` `uvm_info``** command. However there is another technique to control the printing of factory information that can be controlled by verbosity settings.

A verification engineer can use a UVM method to explicitly check what the current verbosity setting is. The **`uvm_report_enabled()`** function will return a 0 or a 1 to inform the user if the

component is currently configured to print messages at the verbosity level selected from the command line. Code can then explicitly check the current verbosity setting before calling functions to display information (**factory.print()**, etc.).

```
function void start_of_simulation_phase(uvm_phase phase);
  super.start_of_simulation_phase(phase);
  ...
  if (uvm_report_enabled(UVM_HIGH)) begin
    this.print;
    factory.print;
  end
endfunction
```

Example 5 - Conditional display of test configuration and factory configuration

The code in Example 5 checks to see if **UVM_HIGH** messages should be printed. If they should be printed, then both the test configuration and the factory configuration information will be printed. I recommend using this block of code in a top-level test as shown in the fully coded **test1.sv** code of Example 14 in the Appendix of this paper.

13. transaction.print() -vs- transaction.sprint()

Calling the transaction**.print()** method is discouraged as it cannot be put under verbosity control.

Calling the transaction**.sprint()** method is a function that returns a string that can be called from the message macros. If you must call a transaction printing method, use verbosity controlled transaction.sprint() over the unconditional transaction**.print()**.

The standard **print()** methods, by default, print transactions in a rather verbose tabular format and consume significant simulation time to format. The tabular form of the standard **print()** method also consumes large amounts of display space each time it is called on a transaction, which generates a large amount of transcript data that has to be manually analyzed. In general it is better to use transaction**.convert2string()**.

14. convert2string()

The **convert2string()** standard transaction method should be implemented for all transaction classes. The **convert2string()** method is user defined and formatted, and consumes less simulation processing time and typically far fewer printed lines of code than the standard **print()** method.

Whoever is in charge of implementing the transaction class should also implement the built-in **convert2string()** method. This is a common courtesy to the rest of the verification team. See [Cummings SNUG-SV 2014] for more details and recommendations.

15. OVM libraries versus UVM libraries

A somewhat amusing exercise is to run an old OVM testbench using the verbosity setting of **OVM_DEBUG**. Using this setting will cause all user messages and a large number of stray OVM library-debug messages to be displayed.

This exercise shows that the old OVM library was debugged using verbosity settings of **OVM_DEBUG** and that some of those messages were forgotten and left in the released OVM library.

This situation actually causes me to suggest an enhancement to UVM and the current **UVM_VERBOSITY** definitions, specifically, that a new definition of **UVM_LIB_DEBUG** with integer setting of 600 be added to the standard verbosity settings. All library development messages should use the **UVM_LIB_DEBUG** verbosity setting and it may even be beneficial to leave some of those messages in the released UVM library if the information might prove useful to verification engineers.

By using **+UVM_VERBOSITY=UVM_DEBUG** the user would only see the user-added debug messages (level 500 and lower) while adding **+UVM_VERBOSITY=UVM_LIB_DEBUG** would show all of the users defined messages and any useful library debug messages during simulation (level 600 and lower).

Users would be cautioned to never use **UVM_LIB_DEBUG** in their own code. The setting, per methodology guidelines, would be reserved for UVM library development only.

The **UVM_LIB_DEBUG** verbosity setting would be fully backward compatible with existing UVM simulations since nobody currently uses a verbosity setting of greater than 500 (**UVM_DEBUG**).

16. UVM message documentation errors

Common UVM message documentation errors fall into three categories:
(1) Description and default documentation errors.
(2) Misleading or erroneous descriptions.
(3) Improper message usage in documented examples.

There were two very good UVM books that were released in 2013, but both books largely use **UVM_LOW** for verbosity settings, which are difficult to disable. **UVM_LOW** should not be the default verbosity setting. Both books also have either erroneous or misleading verbosity descriptions. It is hard to fault Cooper and Meade/Rosenberg for the improper usage because the UVM User's Guide and UVM Class Reference Manual also include the same improper usage.

Description errors or misleading text from the 2013 books include:

Cooper - pg. 28 - "The default verbosity is **UVM_LOW**." (This is wrong - should be **UVM_MEDIUM**)

Meade/Rosenberg - pg. 81 (end of 1st paragraph) - "By default, UVM will print messages with the verbosity **UVM_MEDIUM** (200) and lower. (This is correct)

Meade/Rosenberg - pg. 81 (beginning of 1st paragraph) - "The verbosity level is an integer value to indicate the relative importance of the message." (This is wrong or at least misleading).

Meade/Rosenberg - pg. 81 (Section 4.8.2) - "**UVM_LOW** -- For maskable messages that are printed only rarely during simulation. (This is VERY misleading!)

The Meade/Rosenberg description of **UVM_LOW** is attempting to tell the reader that only important messages, ones that print rarely and should almost always be displayed, should use the **UVM_LOW** verbosity setting. Unfortunately, the wording suggests that messages that you only want to be printed rarely and masked most of the time should use **UVM_LOW**. There is no mention of the importance of the messages.

In the following subsections of the 2013 books and the UVM standard documentation, I cite examples that, in my opinion, improperly use the wrong verbosity setting. Readers are left to make their own judgment about the examples cited.

16.1. Cooper - improper usage examples

All citations are for the 2013 printing of the book. The book has very good beginner content but most verbosity settings should be modified as shown below.

The following cited examples should replace **UVM_LOW** with **UVM_MEDIUM**. The examples do not merit the use of the high-priority **UVM_LOW** verbosity setting but instead should be displayed by default and easily suppressed using the command line option: **+UVM_VERBOSITY=UVM_LOW**

pp. 27, 32 and 92 - function displayAll - this is a transaction display (better yet, put this into a **convert2string()** method inside of the transaction and each call to **convert2string()** can select the appropriate verbosity setting for the transaction data being displayed). On pp. 32 and 92, this should also be a **function**, not a **task** (messages do not consume time).

The following cited examples should replace **UVM_LOW** with **UVM_HIGH** since these are passing messages or useful status messages that should be off by default and only enabled by using the command line option: **+UVM_VERBOSITY=UVM_HIGH**

pg. 21 - top of page - UVM testbench topology (off by default but easily enabled)
pg. 35 - task reset - useful runtime message (off by default but easily enabled)
pg. 37 - task reset - useful runtime message (off by default but easily enabled)
pg. 70 - In function **end_of_elaboration_phase** - UVM testbench topology (off by default but easily enabled - this should not use **UVM_DEBUG**)
pg. 93 - task reset - useful runtime message (off by default but easily enabled)
pg. 94 - task reset - useful runtime message (off by default but easily enabled)

pg. 95 - task reset - useful runtime message (off by default but easily enabled)

The following cited examples should replace **UVM_LOW** with **UVM_FULL** since these are just UVM testbench status messages that should be off by default and only enabled by using the command line option: **+UVM_VERBOSITY=UVM_FULL**

pg. 19 - In function **build_phase** - UVM testbench status
pg. 20 - In function **connect_phase** - UVM testbench status
pg. 21 - In function **build_phase** - UVM testbench status
pg. 21 - In function **start_of_simulation_phase** - UVM testbench status
pg. 34 - In function **build_phase** - UVM testbench status
pg. 36 - In function **build_phase** - UVM testbench status
pg. 47 - In function **build_phase** - **build_phase** interface usage status
pg. 48 - In function **build_phase** - UVM testbench status
pg. 50 - In function **build_phase** - **build_phase** interface usage status
pg. 50 - In function **build_phase** - UVM testbench status
pg. 52 - In function **build_phase** - **build_phase** interface usage status
pg. 56 - In function **build_phase** - UVM testbench status
pg. 59 - In function **build_phase** - UVM testbench status
pg. 60 - In function **build_phase** - UVM testbench status
pg. 64 - In function **build_phase** - UVM testbench status
pg. 93 - In function **build_phase** - UVM testbench status
pg. 94 - In function **build_phase** - UVM testbench status
pg. 98 - In function **build_phase** - **build_phase** interface usage status
pg. 98 - In function **build_phase** - UVM testbench status
pg. 99 - In function **build_phase** - UVM testbench status
pg. 99 - In function **connect_phase** - UVM testbench status
pg. 100 - In function **build_phase** - UVM testbench status

The following cited examples should replace **UVM_LOW** with **UVM_DEBUG** since these are just debug messages that should be off by default and only enabled by using the command line option: **+UVM_VERBOSITY=UVM_DEBUG**

pg. 40 - task data_phase - 2 places - useful during debug
pg. 95 - task get_and_drive - useful during debug
pg. 95 - task data_phase - useful during debug
pg. 96 - task data_phase - useful during debug

The following cited examples, in my opinion, use proper verbosity settings.

pg. 48 - In function **report_phase** - Simulation report - **UVM_LOW**
pg. 53 - In function **report_phase** - Simulation report - **UVM_LOW**
pg. 66 - 2 places in function **extract_phase** - Simulation report - **UVM_LOW**
pg. 101 - 2 places in function **extract_phase** - Simulation report - **UVM_LOW**

16.2. Meade/Rosenberg - improper usage examples

All citations are for the second edition published in 2013. The book has great content but most verbosity settings should be modified as shown below.

The following cited examples should replace **UVM_LOW** with **UVM_MEDIUM**. The examples do not merit the use of the high-priority **UVM_LOW** verbosity setting but instead should be displayed by default and easily suppressed using the command line option: **+UVM_VERBOSITY=UVM_LOW**

pg. 40 - Section 4.1.1 example - line 6
pg. 58 - Example 4-6 - line 31
pg. 82 - Example 4-20 - lines 5 and 6
pg. 92 - Example 4-23 - line 15 (this example might be useful to keep as **UVM_LOW**)
pg. 98 - Section 4.12 - lines 10, 12 and 14 (these infos might be useful to keep as **UVM_LOW**)
pg. 267 - Example 10-3 - line 35
pg. 272 - Example 10-6 - line 28
pg. 287 - Section 10.9.3 - 2 lines

The following cited examples should replace **UVM_LOW** with **UVM_HIGH** since these are passing messages or useful status messages that should be off by default and only enabled by using the command line option: **+UVM_VERBOSITY=UVM_HIGH**

pg. 53 - Example 4-5 - lines 38, 40 and 42
pg. 79 - Example 4-19 - lines 32 and 37
pg. 89 - Example 4-22 - two objection count examples on this page

The following cited examples should replace **UVM_LOW** with **UVM_FULL** since these are just UVM testbench status messages that should be off by default and only enabled by using the command line option: **+UVM_VERBOSITY=UVM_FULL**

pg. 53 - Example 4-5 - lines 10, 19 and 36
pg. 54 - Example 4-5 - lines 58 and 61
pg. 88 - Example 4-22 - two examples on this page
pg. 149 - Example 5-25 - line 8 (replace **UVM_HIGH** with **UVM_FULL**)

The following cited examples should replace **UVM_LOW** with **UVM_DEBUG** since these are just debug messages that should be off by default and only enabled by using the command line option: **+UVM_VERBOSITY=UVM_DEBUG**

pg. 76 - Example 4-17 - class my_project_driver
pg. 79 - Example 4-19 - lines 10, 19 and 21
pg. 84 - Section 4.9.1.2 - line 7
pg. 157 - line 7 (replace **UVM_MEDIUM** with **UVM_DEBUG**)
pg. 246 - lines 16 and 25 (replace **UVM_HIGH** with **UVM_DEBUG**)
pg. 248 - Example 9-10 - 2 lines (replace **UVM_HIGH** with **UVM_DEBUG**)
pg. 251 - Example 9-11 - line 8

pg. 296 - Example 11-1 - 2 places

The following cited examples, in my opinion, use proper verbosity settings.

pg. 130 - Example 5-13 - line 22 - **`UVM_MEDIUM`** for default display
pg. 179 - Example 7-6 - **`UVM_MEDIUM`** for default display

The following cited example, in my opinion, may or may not be using proper severity settings.

pg. 140 - Example 10-9 - line 10 - replace **`uvm_error`** with **`uvm_fatal`** (a missing virtual interface will cause catastrophic testing errors and should probably abort the simulation with a fatal message).

16.3. UVM 1.2 Class Reference - improper usage examples

Section 21.2:
```
uvm_report_info("MYINFO1", $sformatf("val: %0d", val), UVM_LOW);
`uvm_info("MYINFO1", $sformatf("val: %0d", val), UVM_LOW)
```

Section 21.2: Under Message Trace Macros
```
`uvm_info_begin("MY_ID", "This is my message...", UVM_LOW)
```

These examples are technically correct, but the Class Reference should not show typical usage examples that call the **`UVM_LOW`** verbosity setting as it only encourages the improper usage of this setting. These examples should use **`UVM_MEDIUM`** or higher.

16.4. UVM 1.1 User's Guide - improper usage examples

pg. 131 - Section 6.2.1: Replace the **`$display`** command with
```
`uvm_info("FACTORY", $sformatf("type of object is: %s", get_type_name()), UVM_DEBUG)
```

pg. 28 - Section 2.12 - BUG - **`uvm_info`** requires verbosity setting - use **`UVM_MEDIUM`**
pg. 114 - Section 5.7.2.2 - BUG - **`uvm_error`** requires <***id***> argument

The following cited example should replace **`UVM_LOW`** with **`UVM_MEDIUM`**. The examples do not merit the use of the high-priority **`UVM_LOW`** verbosity setting but instead should be displayed by default and easily suppressed using the command line option: **`+UVM_VERBOSITY=UVM_LOW`**

pg. 161 - Example: test_lib.sv - line 33

The following cited examples should replace **`UVM_LOW`** with **`UVM_DEBUG`** since these are just debug messages that should be off by default and only enabled by using the command line option: **`+UVM_VERBOSITY=UVM_DEBUG`**

pg. 137 - Section 6.3.2.1 - two example of **`UVM_LOW`** and one example of **`UVM_HIGH`** (replace with **`UVM_DEBUG`**)

pg. 137 - Section 6.3.2.2 - two example of **`UVM_LOW`** (replace with **`UVM_DEBUG`**) and both examples have a BUG, an extra **`driver`** argument at the end of the **`\`uvm_info`** argument list.

The following cited example, in my opinion, uses a proper verbosity setting.

pg. 162 - Example: test_lib.sv - line 48 - **`UVM_NONE`** for final test-PASS message.

17. UVM_VERBOSITY proposed extensions

One would think that the extensive verbosity settings along with the UVM message macros would be enough to satisfy any testing display commands. After extended usage, I have found a few places where minor enhancements would significantly simplify testbench development. Those experiences have caused me to propose the following message-control enhancements.

17.1. `uvm_info() macro with default verbosity setting of UVM_MEDIUM

All of the UVM message macros have a default verbosity setting except **`\`uvm_info()`**. It is rather annoying to be forced to add a default verbosity setting of **`UVM_MEDIUM`** each time this macro is coded.

Request to UVM Committee: add a default UVM_MEDIUM verbosity to **`\`uvm_info()`**. This does not break any backward compatibility and simplifies the use of the `uvm_info message macro to do standard display commands.

17.2. `uvm_warning() macro with default verbosity setting of UVM_NONE

The **`\`uvm_warning`** macro, in its current form, is almost useless. The unchangeable default for **`\`uvm_warning`** is a verbosity setting of **`UVM_NONE`**, which is almost impossible to turn off. There are times when warnings are useful and there are times when some warnings are just verbose, annoying, distracting and would be candidates to disable with a different verbosity setting.

I currently almost never use **`\`uvm_warning()`** because these message cannot be easily disabled. It was rather presumptuous of the UVM committee to assume that all warnings would be unconditionally printed.

Request to UVM Committee: add a default **`UVM_NONE`** verbosity to **`\`uvm_warning()`**. This does not break any backward compatibility and allows users to reduce the verbosity setting of specific warning messages as appropriate for the users UVM test environment.

17.3. New `uvm_info_pass() macro and UVM_PASS verbosity setting

By default, when running tests, the typical verification engineer does not want to see messages for all tests that are passing, but an engineer might want to turn those messages on without adding all of the other more verbose messages. I currently recommend using a verbosity setting of **`UVM_HIGH`** to display information that should only be shown occasionally. This is the same setting that I currently use to display messages for passing tests.

It would be most useful to have an additional verbosity setting between **`UVM_MEDIUM`** (200) and **`UVM_HIGH`** (300), reserved by methodology to display messages for all passing tests. New suggested verbosity names include **`UVM_PASS`** or **`UVM_SUCCESS`** (level 250). The actual name is not that important and alternate suggested names could be evaluated.

I further propose the addition of **`` `uvm_info_pass ``** or **`` `uvm_info_success ``** message macros with a default verbosity setting of **`UVM_PASS`** or **`UVM_SUCCESS`** (level 250) that could be changed by the user if desired. It would be very useful to setup scoreboard comparisons to report failing tests using **`` `uvm_error() ``** and passing tests using **`` `uvm_info_pass() ``** or **`` `uvm_info_success() ``**. The passing tests would not print by default but a verification engineer could enable all passing test messages, without adding any other messages, by adding an option such as **`+UVM_VERBOSTIY=UVM_PASS`** to the simulation command line.

Request to UVM Committee: add a verbosity setting of **`UVM_PASS`** or **`UVM_SUCCESS`** (or equivalent) with a verbosity level of 250.

Request to UVM Committee: add a message macro called **`` ` uvm_info_pass() ``** or **`` `uvm_info_success() ``** with a default verbosity of **`UVM_PASS`** (level of 250) or equivalent.

These are enhancement requests that do not break backward compatibility.

17.4. New `uvm_debug() macro

Adding debug messages is a typical and frequent verification activity. The current recommended practice is to add **`` `uvm_info(..., UVM_DEBUG) ``** for each debug message. This activity is so common that a little syntactic sugar would be most useful and appreciated.

Request to UVM Committee: add a message macro called **`` `uvm_debug() ``** that calls the **`` `uvm_info() ``** message macro with a default verbosity of **`UVM_DEBUG`**.

This is an enhancement request that does not break backward compatibility, but adds an extremely useful macro definition for a very common testbench coding activity.

18. Conclusions

UVM was released in 2011. UVM was the next generation of the combined efforts of OVM and VMM, and included advanced features from both predecessor methodologies. The Best Known Methods (BKMs) of UVM are still being developed and the industry is counting on smart verification engineers to suggest and promote those methods.

This paper has recommended BKMs for using messaging capabilities in UVM. The author is not deluded enough to believe that these techniques will ultimately be the very best techniques and hence should never be challenged.

Authors of existing UVM texts and reference materials showed initial usage models, which I believe were somewhat flawed, but they at least offered a starting point for message usage methodologies. My paper is built upon the work of existing authors and I believe the recommendations included in this paper form the next stage in message usage BKMs.

Readers are encouraged to further refine or build upon the recommendations in this paper and to share those recommendations with the industry in general, and by email to me individually. I welcome all feedback and suggested improvements to the recommendations included in this paper.

To explore expert techniques related to creating Advanced UVM Report Servers, readers are encouraged to review the paper by my colleague and friend Gordon McGregor[4].

19. Acknowledgements

I am grateful to my colleagues Jeff Montesano of Verilab, Jonathan Bromley or Verilab, and Kevin Geiger, Verification AC at Synopsys for their reviews, and for identifying errors and suggesting improvements to the content and flow of this paper.

20. References:

[1] Adam Erickson, "Are OVM & UVM Macros Evil? A Cost-Benefit Analysis," DVCon 2011. Copy can also be requested at: http://www.mentor.com/products/fv/verificationhorizons/horizons-jun-11

[2] Clifford E. Cummings, "UVM Transaction - Definitions, Methods and Usage," SNUG-SV 2014 - http://www.sunburst-design.com/papers/CummingsSNUG2014SV_UVM_Transactions.pdf

[3] Clifford E. Cummings, "The OVM/UVM Factory & Factory Overrides - How They Work - Why They Are Important," SNUG-SV 2012 - http://www.sunburst-design.com/papers/CummingsSNUG2012SV_UVM_Factories.pdf

[4] Gordon McGregor, "Applications of Custom UVM Report Servers," SNUG-Austin 2013- http://www.verilab.com/files/SNUG_Applications_of_custom_UVM_report_servers.pdf

[5] Kathleen A. Meade, Sharon Rosenberg, A Practical Guide to Adopting the Universal Verification Methodology (UVM), Second Edition, ISBN 978-1-300-53593-5. Published 2013

[6] OVM User Guide, March 2010, Available for download from: https://verificationacademy.com/topics/verification-methodology

[7] Universal Verification Methodology (UVM) 1.1 Class Reference, May 2011, Accellera, Napa, CA. www.accellera.org/home

[8] Universal Verification Methodology (UVM) 1.2 Class Reference, May 2014, Accellera, Napa, CA. www.accellera.org/home

[9] Vanessa R. Cooper, Getting Started with UVM: A Beginner's Guide, ISBN-10: 0615819974 | ISBN-13: 978-0615819976, Published 2013

[10] verificationacademy.com/cookbook/Reporting/Verbosity

21. AUTHOR & CONTACT INFORMATION

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 32 years of ASIC, FPGA and system design experience and 23 years of SystemVerilog, synthesis and methodology training experience.

Mr Cummings has presented more than 100 SystemVerilog seminars and training classes in the past nine years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.
Email address: cliffc@sunburst-design.com

Last Updated: September 2014

22. Appendix

This Appendix includes the files that I used to test the features described in the paper.

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)

       logic [15:0] dout;
  rand bit   [15:0] din;
  rand bit          rst_n;

  function new (string name="trans1");
    super.new(name);
  endfunction

  function string convert2string();
    return($sformatf("trans1: dout=%4h  din=%4h  rst_n=%b",
                             dout, din, rst_n));
  endfunction
endclass
```

Example 6 - File: trans1.sv

```
`ifndef CYCLE
  `define CYCLE 10
`endif
`timescale 1ns/1n
```

Example 7 - File: CYCLE.sv

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

  tb_agent agnt;

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

  function void build_phase(uvm_phase phase);
    agnt = tb_agent::type_id::create("agnt", this);
  endfunction

  function void start_of_simulation_phase(uvm_phase phase);
    `uvm_error("ENV", "env error")
  endfunction
endclass
```

Example 8 - File: env.sv

```
CYCLE.sv
tb_pkg.sv
+incdir+..
top.sv
```

Example 9 - File: run.f

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

  tb_driver     drv;
  tb_sequencer  sqr;

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

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    drv =    tb_driver::type_id::create("drv", this);
    sqr = tb_sequencer::type_id::create("sqr", this);
  endfunction

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

  function void start_of_simulation_phase(uvm_phase phase);
    `uvm_fatal("AGENT", "agnt fatal msg")
  endfunction
endclass
```

Example 10 - File: tb_agent.sv

```
class tb_driver extends uvm_driver #(trans1);
  `uvm_component_utils(tb_driver)

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

  task run_phase(uvm_phase phase);
    trans1 tr;
    `uvm_warning("DRIVER", "Starting tb_driver run_phase()")
    forever begin
      seq_item_port.get_next_item(tr);
      `uvm_info("DRIVER",  tr.convert2string(), UVM_MEDIUM)
      seq_item_port.item_done();
    end
  endtask
endclass
```

Example 11 - File: tb_driver.sv

```
`ifndef TB_PKG
`define TB_PKG

`include "CYCLE.sv"

`include "uvm_macros.svh"

package tb_pkg;
  import uvm_pkg::*;

  `include "test_error_demoter.sv"
  `include "test1_demoter.sv"

  `include "trans1.sv"
  `include "tb_driver.sv"
  `include "tb_sequencer.sv"
  `include "tb_agent.sv"
  `include "env.sv"

  `include "tr_sequence.sv"

  `include "test1.sv"
  `include "test1x.sv"
  `include "test2.sv"
  `include "test3.sv"
endpackage

`endif
```

Example 12 - File: tb_pkg.sv

```
class tb_sequencer extends uvm_sequencer #(trans1);
  `uvm_component_utils(tb_sequencer)

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

Example 13 - File: tb_sequencer.sv

```
class test1 extends uvm_test;
  `uvm_component_utils(test1)

  env e;

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

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

  function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    `uvm_warning("test1", "Test1 warning message")
    if (uvm_report_enabled(UVM_HIGH)) begin
      this.print;
      factory.print;
    end
  endfunction

  task run_phase(uvm_phase phase);
    tr_sequence seq;
    seq = tr_sequence::type_id::create("seq");
    //-------------------------------------
    phase.raise_objection(this);
    seq.start(e.agnt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Example 14 - File: test1.sv

```
class test1_demoter extends uvm_report_catcher;
  `uvm_object_utils(test1_demoter)

  function new(string name="test1_demoter");
    super.new(name);
  endfunction

  function action_e catch();
    if(get_severity() == UVM_FATAL) begin
      set_severity(UVM_ERROR);
      `uvm_info("demoter", "Caught FATAL / demoted to ERROR", UVM_MEDIUM)
    end
    //return CAUGHT;
    return THROW;
  endfunction
endclass
```

Example 15 - File: test1_demoter.sv

```
class test1x extends test1;
  `uvm_component_utils(test1x)

  test1_demoter demoter;

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

  function void build_phase(uvm_phase phase);
    demoter = test1_demoter::type_id::create("demoter");
    uvm_report_cb::add(e, demoter);
    super.build_phase(phase);
  endfunction
endclass
```

Example 16 - File: test1x.sv

```
`include "CYCLE.sv"
`include "uvm_macros.svh"
module top;
  import uvm_pkg::*;
  import  tb_pkg::*;

  initial begin
    run_test();
  end
endmodule
```

Example 17 - File: top.sv

```
class tr_sequence extends uvm_sequence #(trans1);
  `uvm_object_utils(tr_sequence)

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

  task body;
    repeat(2) do_item();
  endtask

  task do_item ();
    trans1 tr;
    //`uvm_info("SEQ","Running sequence do_item()",UVM_HIGH)
    `uvm_warning("SEQ","Running sequence do_item()")
    repeat(4) `uvm_do(tr)
  endtask
endclass
```

Example 18 - File: tr_sequence.sv

```
class tb_agent2 extends tb_agent;
  `uvm_component_utils(tb_agent2)

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

  function void start_of_simulation_phase(uvm_phase phase);
    `uvm_warning("AGENT", "agnt warning msg")
  endfunction
endclass
```

Example 19 - File: tb_agent2.sv

```
class test2x extends test2;
  `uvm_component_utils(test2x)

  test_report_catcher demoter;

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

  function void connect_phase(uvm_phase phase);
    demoter = test_report_catcher::type_id::create("demoter");
    uvm_report_cb::add(e.agnt, demoter);
    super.connect_phase(phase);
  endfunction
endclass
```

Example 20 - File: test2x.sv

```
class test_report_catcher extends uvm_report_catcher;
  `uvm_object_utils(test_report_catcher)

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

  // This example promotes "AGENT" warnings to error messages
  function action_e catch();
    if(get_severity() == UVM_WARNING && get_id() == "AGENT") begin
      set_severity(UVM_ERROR);
      set_message("Caught AGENT WARNING / promoted to ERROR");
    end
    return THROW;
  endfunction
endclass
```

Example 21 - File: test_report_catcher.sv