

Better Living Through Better Class-Based SystemVerilog Debug

Rich Edelman
Mentor Graphics
Fremont, CA

Raghu Ardeishar
Mentor Graphics
Columbia, MD

John Amouroux
Mentor Graphics
Fremont, CA

Abstract—SystemVerilog [1] UVM [2] class-based testbenches have become as complex as the hardware under test, and are evolving into large object oriented software designs. The usual RTL debugging techniques must be updated to match this new complexity. Debugging tools are addressing these complexities, but this paper will describe techniques and approaches that can be used to help debug these complex environments without advanced debug tools.

Keywords; *SystemVerilog class-based debug, Testbench debug, UVM debug*

I. INTRODUCTION

Debugging large testbenches has changed recently. The testbenches are larger than they used to be, and they are more like software than they used to be. In addition, the testbench language features use object-oriented constructs, and may use a new library of verification components. Each of these characteristics adds to the pain of debugging the testbench, which must be done before debugging of the actual device-under-test can begin.

This paper covers simulator independent debugging techniques that can be adopted as aids during the testbench debug period. Each technique is a small step for better debug, but when considered together can compensate for tool limitations, staff training issues, or other issues with adopting the SystemVerilog language and verification libraries. The code and techniques are standard SystemVerilog and should be available on all modern, compliant simulators.

We discuss simple organizational techniques for debug like naming conventions for files, directories, verification IP, class names and class member variables.

Breakpoints are important to track flow of control, but can be hard to use in dynamic, class based testbenches. We discuss breakpoint pools, and a small infrastructure that can be used to turn breakpoints on and off using the command line. This technique requires a little bit of planning, but can aid flow control debugging. These breakpoints can be class based or even instance based.

Verification libraries, like the UVM, contain many debug statements and aids that can be either compiled on, or turned on from the command line. The UVM itself contains dump() and print() routines that can be used to explore the UVM data

structures. We describe easy ways to build compiled-in debug and dynamically controlled debug for user testbenches.

Most large UVM based testbenches are transaction based. We discuss using transactions for debug, including simple convert2string() usage, and recommendations against using any of the field automation macros. Although not fatal, these macros can hide problems, and create an unnecessary distance between the testbench transaction and the verification engineer.

Transactions can be recorded, but usually printing transactions can be just as useful. Using the messaging system and convert2string(), transaction data and control flow can be debugged.

We recommend using one single simulation log file, which provides instant correlation of the entire system. With one log file, tools like sed, awk, grep and perl can be used to extract information, create reports and produce debug summaries. Using a single log file allows all the time correlated data produced by the simulator to be recorded as it was produced. If that information is split into multiple log files from the simulator, then it is hard to correlate events after the fact. Use post-processing tools to extract interesting data; don't use the simulator to do that.

The UVM reporting system (messaging) has many abilities, including verbosity control, message ID filtering and hierarchical property setting. This system is large and complex, and although it has been available for years, it is mostly opaque to all but the most energetic engineer. We'll show how to control this system with command line options, with a small message control block. Finally, we'll build a custom reporting environment with a user defined report server and report catcher.

II. STRATEGY

Any testbench must be planned, and a testbench which supports debug is no exception. Debug does not come for free, and must be designed in. Debug tools are available, and can minimize the need for planning and design-in, but even they can require command line switches, or require a different kind of compilation flow.

For SystemVerilog UVM class based debug, we are going to assume full SystemVerilog LRM support, and support for the latest UVM release.

This paper assumes there is no special debug tool being used. The only debugging available is debugging that comes from the simulator running.

III. NAMING CONVENTIONS

Naming conventions [3] [4] are often overlooked as either too simplistic or onerous, but they can often simplify debug by making it easier to understand a new test bench. The simplest conventions are usually the best, and should be followed where possible, but should not be used rigidly.

The best naming conventions are those that you agree to and that you use consistently. Some recommendations or suggestions are listed below – but they are just that – suggestions.

Naming conventions help eliminate mistakes by being consistent and simple to understand. Reading the code during debug is easy. Furthermore, finding things becomes easy, since they are where they should be. Finding all the axi_scoreboard factory creation calls in the testbench source code is easy with grep:

```
grep -r axi_scoreboard::type_id::create .
```

A. Names – Files, Classes, Member Variables

Use lowercase file names and directory names. Use descriptive suffixes like `_pkg`, `_scoreboard`, `_agent`, `_env`. Use the `.sv` file extension for files that get compiled. Use the `.svh` file extension for files that get included. Never compile a `.sv` file directly. Never ``include` a `.sv` file.

Generally speaking you should not use the ``include` construct, except to include a file in just one place. A ``include` should be used to build a package file (see below). If you need to reference a type or other definition, then use `'import'` to bring the definition into scope. Do not use ``include` for general solutions.

```
package abc_pkg;
`include "abc_macros.svh"
`include "abc_monitor.svh"
`include "abc_driver.svh"
...
endpackage
```

Files should be named after what they are - for example, `ahb_scoreboard.svh` will contain the AHB scoreboard, while the `axi_vip_pkg.sv` contains the AXI Verification IP.

Create a class to implement functionality with a name that describes the functionality. For example, a scoreboard for the ABC verification IP would be named - `abc_scoreboard`.

Use a prefix in the class name for the kind of object this class is associated with. In this case `'class abc_scoreboard'` is a scoreboard associated with an `'abc'` kind of verification component.

Any member that is meant to be private should be named with a `'m_'` prefix, and should be made local or protected. Any member that will be randomized should not be local or protected. The `'m_'` prefix is a simple reminder that this variable or function is not publically available.

B. Directories and Packages

Testbenches are constructed of SystemVerilog UVM code organized as packages, collections of verification IP organized as packages and a description of the hardware to be tested. Other files such as C models are also to be organized.

Each package should exist in its own directory. Each of these package directories should have one file that gets compiled - a file with the extension `.sv`

```
abc_pkg.sv
```

Each package should have at most one file that may be included in other code. This file may define macros.

```
abc_macros.svh
```

For a complex package that may contain tests, examples and documentation, create subdirectories:

```
abc_pkg/examples
abc_pkg/docs
abc_pkg/tests
abc_pkg/src/abc_pkg.sv
```

Sample file listing

```
abc_pkg/src
abc_pkg/src/abc_pkg.sv

abc_pkg/src/abc_macros.svh

abc_pkg/src/abc_env.svh
abc_pkg/src/abc_interface.sv

abc_pkg/src/abc_driver.svh
abc_pkg/src/abc_monitor.svh
abc_pkg/src/abc_scoreboard.svh

abc_pkg/src/abc_sequence_item.svh
abc_pkg/src/abc_sequencer.svh
abc_pkg/src/abc_sequences.svh

abc_pkg/docs/
abc_pkg/docs/abc_user_guide.docx

abc_pkg/tests/.....

abc_pkg/examples/....

testbench1/makefile
testbench1/tb_env.sv
testbench1/tb_top.sv
testbench1/test.sv
```

When you use a package, you import it, and include any macro definitions.

```
import abc_pkg::*;
`include "abc_macros.svh"
```

Using these suggestions will help you get organized, and keep you out of trouble.

IV. DESIGNING THE TESTBENCH

The testbench should be designed with debug in mind. The main elements that will be used in debugging will be transactions, sequences and phased components. Building each of these with debug in mind is important.

Transactions should be designed so that you can figure out what data is stored in the transaction. For example a routine named ‘convert2string’ should be provided that creates a string representing the current “value” of the transaction. A transaction may also want to register the fact that it has been constructed.

Sequences should be designed to print information when they start and when they end. They should also have a way to breakpoint. They may also want to register the fact that a sequence has been constructed.

Phased components should be designed to print information when each phase starts and ends. They should also have a way to breakpoint.

V. DO IT YOURSELF TOOLS

A. *Breadcrumbs*

In order to understand control flow, or function call order, the user might insert a print statement that printed the file and line number when executed.

```
$display("DBG: %s:%0d", `__FILE__, `__LINE__);
```

Each time this line is executed, it leaves a trail of where it has been. Instrumenting each line in a difficult section of code may help debug problems. The last line printed is where the problem is.

In addition to a simple file and line number, in class based debug, we can print the contents of the class. A SystemVerilog based solution is just to use ‘%p’ with a \$display or \$formatf:

```
$display("sequence_item=%0p",
        seq_item_handle);
```

A UVM based solution is just to use convert2string():

```
$display("sequence_item=%s",
        seq_item_handle.convert2string());
```

These basic print statements can provide a good amount of debug. They provide visibility into control flow, and visibility into data. If your code is stuck in a loop somewhere, and there are many conditions for exiting the loop, you could set a breakpoint at the top of the loop, and single step through the loop. Sometimes this could take a long time before the problem appeared, and it relies on the debug tool providing breakpoints and single stepping. A crude but effective alternative is to sprinkle the \$display that uses FILE and LINE on most lines in the loop or function. Then just run simulation, allowing the FILE and LINE messages to fill up your transcript. Eventually you may see a pattern. Certain lines may be repeating, and other lines may never be executed. With this information you can infer which if-then-else is failing or what expected exit condition is not being met.

This kind of breadcrumb trail can be extended to many situations. For example, at the beginning of every function or task call, you can add a \$display, and print a “start” message

```
$display("Function %s starting", "FunctionName");
```

With the UVM more information is available. The function or task in a UVM based class can include information from

itself, its containing class, or any ancestor class. For example a function in a uvm_component could print the name of the class (xyz_h), the type of the class (xyz), the full name of the class (top.env. xyz_h).

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

    task run_phase(uvm_phase phase);
        $display("%s %s %s", get_name(),
            get_type_name(), get_full_name());
    endfunction
endclass
```

Under normal circumstances, the debug messages are not needed or are undesirable. They can be deleted from the source code, but usually they get put back in to help on the next phase of debugging. Those kinds of messages can be conditionally turned off and on using a simple command line switch. Don’t leave breadcrumb messages turned on when you don’t need them. Either delete them, compile them out, or turn them off with a run time switch.

```
if ($test$plusargs("debug")) begin
    $display("%s %s %s", get_name(),
        get_type_name(), get_full_name());
end
```

Or

```
if (debug) begin
    $display("%s %s %s", get_name(),
        get_type_name(), get_full_name());
end
```

Or

```
`ifndef NO_DEBUG
if (debug) begin
    $display("%s %s %s", get_name(),
        get_type_name(), get_full_name());
end
`endif
```

The UVM also provides a powerful message logging or reporting system that can be used in place of the simple \$display. (^uvm_info).

B. *Threads*

Debugging threaded programs is difficult. Instrumenting each thread is important. Each branch should be instrumented – each decision point. The start and end of the thread should be instrumented. There will be many messages generated and they will be time correlated. It will be easy to see which thread is running, which thread is doing what.

A good example of a common problem is two threads communicating through an infinite size fifo. The producer thread runs, but the consumer thread never runs. By instrumenting the threads it will be apparent that the producer thread is always ready to run, so it continues to run, starving the consumer. By conditioning the producer on a clock edge, or to yield with a #0 after each 10 transactions are generated, the consumer will be allowed to run.

Threads exist in SystemVerilog and the UVM in initial and always blocks, in fork/join constructs, in UVM phases

(run_phase(), build_phase(), etc) and in UVM sequences (body() tasks). There are many opportunities to have threads running.

C. Plus args control

Plus args are command line switches supported by the simulator. They are “application specific” – they can be defined for each application. The SystemVerilog LRM defines that arguments beginning with the ‘+’ character will be available using the \$test\$plusargs and \$value\$plusargs PLI APIs. Using plusargs is a good way to set a mode – like the ‘debug’ mode. Or to set a value – like the ‘debug_level’.

```
begin
  bit debug;
  debug = 0;
  if ($test$plusargs("d"))
    debug = 1;
  $display("Debug = %0d", debug);
end

begin
  int debug_level;
  if ($value$plusargs("debug=%d", debug_level)==0)
    debug_level = 1000;
  $display("DebugLevel = %0d", debug_level);
end
```

Usage:

```
<simulator> +d +debug_level=42
```

Using plusargs allows one setting to be made. In the case of multiple settings (like setting many breakpoints using the +bp switch), a different solution is needed.

The SystemVerilog LRM specifies that the vpi_get_vlog_info function returns information about the tool invocation, including the command line arguments. In the UVM, an interface has been defined to this vpi_get_vlog_info function. It is called the command line processor (clp). Using the clp.get_arg_values(+bp=, list), a list of all the +bp= settings is returned in the array ‘list’. Below, ‘list’ is iterated, each iteration being a +bp= usage from the simulator command line.

```
uvm_cmdline_processor clp;
clp = uvm_cmdline_processor::get_inst();
// By Name => +bp:name=
clp.get_arg_values("+bp=", list);
$display("parsing breakpoints=%p", list);
foreach (list[i])
  BPP::set_by_name(list[i]);
```

Usage:

```
<simulator> +bp=.seqA.* +bp=.seqB.*
```

This is a useful technique to grab all the plusargs of a certain kind from the command line.

D. Quieting Debug Messages

Sprinkling code with debug messages makes an easy to debug, verbose testbench. But eventually the testbench will be debugged, and those messages should get turned off. They can be turned on and off with either compile switches and/or run time switches.

```
`ifndef NO_DEBUG
`define DEBUG(msg) \
  `uvm_info("DBG", msg, UVM_MEDIUM)
`else
`define DEBUG(msg)
`endif
```

In this case, `DEBUG(“abc”) is used in the testbench code to call `uvm_info and produce messages. When it is time to turn these messages off, a +define+NO_DEBUG can be used on the compile invocation.

Runtime control is also possible for example, by re-defining the macro to be

```
`ifndef NO_DEBUG
`define DEBUG(msg) \
  if (debug) `uvm_info("DBG", msg, UVM_MEDIUM)
`else
`define DEBUG(msg)
`endif
```

Elsewhere in the testbench, we declared ‘debug’ and set it from the simulation command line.

```
int debug = 0;
...
if ($test$plusargs("debug") == 1)
  debug = 1;
else
  debug = 0;
```

Note that this solution has both compile time and run time options.

E. Transaction Debug

A transaction is a piece of data that is being communicated. It can contain many different kinds of information – but for debug purposes a function like convert2string should be defined that returns the important state information in a formatted string. Creating convert2string in the base class means that any class based debug can use any class handle, and call

```
$display("%s", transaction_handle.convert2string());
```

In the code snippet below, convert2string is defined to format the id and data values. Additionally, a function call like ‘get_type_name()’ can provide useful information. For the code snippet, get_type_name is defined by the uvm_object_utils macro.

Another useful feature to embed in a transaction is a unique id. That way, if you ever see another transaction with the same id, then you know that it is really is the same transaction – the same object. The id below is such a unique id – it is incremented for each call to new(). The UVM also has a unique id built in for each uvm_object that is created. In the UVM, it is called ‘m_inst_id’, and it can be accessed by calling ‘get_inst_id()’.

```
class my_transaction extends uvm_transaction;
  `uvm_object_utils(my_transaction)

  static int g_id;
  int id;
  int data;

  function new(string name = "my_transaction");
```

```

    super.new(name);
    `ALLOCATED_TYPE
    id = g_id++;
endfunction

function string convert2string();
    return $sprintf("id=%0d, data=%0d",
        id, data);
endfunction
endclass

```

VI. THE UVM

A. UVM Breadcrumbs

Instead of \$display, use `uvm_info, `uvm_warning, `uvm_error or `uvm_fatal. Additional information is built-in, like filename, line number, time, and verbosity control. These breadcrumbs come with their own controls and capabilities.

B. UVM Macros

Don't use macros that generate complex code. It is very hard to debug complex code, and doubly hard when that code is hidden in a macro. There are useful macros, and macros to avoid [8].

C. Really Easy UVM Debug

Interested in what's in the factory?

```

uvm_factory factory= uvm_factory::get();
factory.print();

```

```

#### Factory Configuration (*)
#
# No instance overrides are registered with this
factory
#
# Type Overrides:
#
# Requested Type  Override Type
# -----
# driverA         driver2A
# <unknown>      <unknown>
#
# All types registered with the factory: 57 total
# (types without type names will not be printed)
#
# Type Name
# -----
# driver2A
# driverA
# envA
# questa_uvm_recorder
# sequence_A
# sequence_A_grandparent
# sequence_A_greatgrandparent
# sequence_A_parent
# sequence_item_A
# sequence_item_B
# test
# (*) Types with no associated type name will be
printed as <unknown>

```

Interested in which resource [9] (which configuration) setting was set, but never used? Check it with 'check_config_usage'.

```

uvm_top.check_config_usage(1);

#
# ::: The following resources have at least

```

```

#         one write and no reads :::
# config_object [/^test\.e2_a\..*$/] : ?
# -
# -----
# test.e2_a reads: 0 @ 0 writes: 1 @ 0
#
# config_object [/^test\.e1_a\..*$/] : ?
# -
# -----
# test.e1_a reads: 0 @ 0 writes: 1 @ 0
#
# my_int [/^test\.e2_a\.sequencer$/] : ?

```

Less useful, simply for the quantity and style of information produced, print the configurations available at each hierarchy level.

```

uvm_top.print_config(1, 1);

resources that are visible in
recording_detail [/^.*$/] : ?
-
-----
reads: 0 @ 0 writes: 2 @ 0
test reads: 1 @ 0 writes: 0 @ 0
test.e1_a reads: 1 @ 0 writes: 0 @ 0
test.e1_a.driver reads: 1 @ 0 writes: 0 @ 0
test.e1_a.driver.rsp_port reads: 1 @ 0 writes: 0 @ 0
test.e1_a.sequencer reads: 1 @ 0 writes: 0 @ 0

```

To see the UVM topology from the top use print_topology().

```

uvm_top.print_topology();

# UVM_INFO @ 0: reporter [UVMTOP] UVM testbench
# -----
# Name                                     Type
# -----
# test                                     test
#   e1_a                                   envA
#     driver                               driver2A
#       rsp_port                           uvm_analysis_port
#       sqr_pull_port                       uvm_seq_item_pull_port
#       sequencer                           uvm_sequencer
#       rsp_export                           uvm_analysis_export
#       seq_item_export                       uvm_seq_item_pull_imp
#   e1_b                                   uvm_env
#     driver                               driver2#(sequence_item_A)
#       rsp_port                           uvm_analysis_port
#       sqr_pull_port                       uvm_seq_item_pull_port
#       sequencer                           uvm_sequencer
#       rsp_export                           uvm_analysis_export
#       seq_item_export                       uvm_seq_item_pull_imp
#   e2_a                                   envA
#     driver                               driver2A
#       rsp_port                           uvm_analysis_port
#       sqr_pull_port                       uvm_seq_item_pull_port
#       sequencer                           uvm_sequencer
#       rsp_export                           uvm_analysis_export
#       seq_item_export                       uvm_seq_item_pull_imp
#   e2_b                                   uvm_env
#     driver                               driver2#(sequence_item_A)
#       rsp_port                           uvm_analysis_port
#       sqr_pull_port                       uvm_seq_item_pull_port
#       sequencer                           uvm_sequencer
#       rsp_export                           uvm_analysis_export
#       seq_item_export                       uvm_seq_item_pull_imp

```

Use one of the library calls mentioned in this paper to print a complete netlist.

```

show_connectivity(uvm_top, 0);

+ UVM Root: uvm_top
+ Test: test

```

```

+ Env: test.el_a
+ Component: test.el_a.driver
  Port: test.el_a.driver.sqr_pull_port
  Connected to Port:
    test.el_a.sequencer.seq_item_export

+ Sequencer: test.el_a.sequencer
  Port: test.el_a.sequencer.seq_item_export
  Provided to Port:
    test.el_a.driver.sqr_pull_port

```

D. Reporting System Control

The reporting system has many features, but the most useful and easiest to use is verbosity. Planning your testbench, you should liberally sprinkle calls to the reporting system in, so that you can be made aware of process flow and current state. Normally most of these debug and status statements are quiet – they are turned off. They can be turned on from the command line in a variety of ways.

If you want to turn them all on, then you can just run simulation with verbosity turned up very high – that way your testbench will be very verbose.

If you want to turn on just a few components, then you can control the instances which will be verbose.

Simulating with

```

+UVM_VERBOSITY=UVM_HIGH
or
+UVM_VERBOSITY=UVM_DEBUG

```

will produce many messages – likely some detailed messages from the UVM itself. Your output will be very detailed, and very verbose.

Instead of turning the verbosity up for the entire testbench, the verbosity for a small part can be increased using

```
+uvm_set_verbosity=e.consumer*,_ALL_,UVM_HIGH.
```

This causes the string ‘e.consumer*’ to get matched, and all IDs at that level get verbosity set to UVM_HIGH. In our producer/consumer example this makes the consumer verbose, while the producer stays quiet.

It is good debug practice to introduce a debug statement in interesting places, for example when a phase starts (begin_phase(), run_phase() especially) or when a sequence begins or ends (at the start and end of the body() task). These messages remain quiet if they are using a verbosity higher than the normal verbosity. Normal verbosity is UVM_MEDIUM. The verbosity choices are

```

UVM_NONE,
UVM_LOW,
UVM_MEDIUM,
UVM_HIGH,
UVM_FULL,
UVM_DEBUG.

```

Normally simulations run at UVM_MEDIUM, and debug messages are UVM_HIGH or UVM_FULL.

For example, a run_phase instrumentation:

```

task run_phase(uvm_phase phase);
...

```

```

`uvm_info(get_type_name(), "Starting... ",
UVM_HIGH)

```

Or a sequence body task instrumentation:

```

task body();
  T t;
  `uvm_info(get_type_name(), "Starting... ",
UVM_HIGH)

```

These messages won’t be printed unless the verbosity is at UVM_HIGH or higher.

The first argument in the `uvm_info macro is the ID. It’s a good practice to use the type_name – that way you can control all the messages from a certain type by causing the reporting system to change verbosity or other attributes based on the ID.

E. Data Structures

The UVM Class data structures can be explored to provide information about the testbench structure, its current state, and the state of any generated stimulus. There are many APIs available to choose from – the uvm-1.1a contains 316 class definitions in 134 files for a total of 67,298 lines of code. Test benches can contain hundreds or thousands of instances of class based components. They can generate thousands or millions of transactions or tests. They can run for hours or days. These are complex systems described by complex data structures.

F. Netlist

A simple netlist program can be built to traverse the UVM component hierarchy and print the connections. In Figure 1 a simple UVM producer is connected to a fifo. The fifo is connected to a UVM consumer. The netlist produced shows that the environment ‘e’ contains three components (consumer, fifo and producer). The consumer has a port named ‘port’, which is connected to a port on the fifo named ‘get_peek_export’.

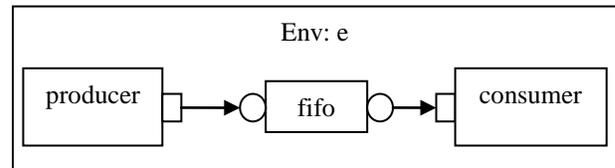


Figure 1 - Simple Producer / Consumer

The UVM code below constructs the hierarchy shown in Figure 1.

```

class env extends uvm env;
  `uvm_component_utils(env)

  producer p;
  consumer c;
  uvm_tlm_fifo #(my_transaction) fifo;

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

  function void build_phase(uvm_phase phase);
    p = new("producer", this);

```

```

    c = new("consumer", this);
    fifo = new("fifo", this);
endfunction

function void connect_phase(uvm_phase phase);
    p.port.connect(fifo.put_export);
    c.port.connect(fifo.get_export);
endfunction
endclass

```

Netlist output:

```

+ Env: e
+ Component: e.consumer
  Port: e.consumer.port
    Connected to Port: e.fifo.get_peek_export
+ Component: e.fifo
  Port: e.fifo.get_peek_export
    Provided to Port: e.consumer.port
  Port: e.fifo.put_export
    Provided to Port: e.producer.port
+ Component: e.producer
  Port: e.producer.port
    Connected to Port: e.fifo.put_export

```

A more interesting netlist from a UVM perspective is the netlist from Figure 2. This diagram shows a test containing an environment, containing a sequencer and driver. The netlist shows the driver `sqr_pull_port` connected to the sequencer `seq_item_export`. It also shows the reverse – the sequencer `seq_item_export` provided to the driver `sqr_pull_port`.

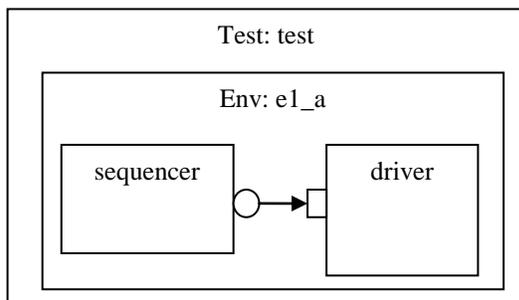


Figure 2 - Sequencer / Driver Connection

Netlist output:

```

+ UVM Root: uvm_top
+ Test: test
+ Env: test.e1_a
+ Component: test.e1_a.driver
  Port: test.e1_a.driver.sqr_pull_port
    Connected to Port:
      test.e1_a.sequencer.seq_item_export
+ Sequencer: test.e1_a.sequencer
  Port: test.e1_a.sequencer.seq_item_export
    Provided to Port:
      test.e1_a.driver.sqr_pull_port

```

The netlisting code is shown abbreviated below. The function `show_connectivity()` is called with a component 'c'. First all the direct children are collected into an array named 'children'. Each child is visited using the `foreach(children[i])` loop. A child is either a port or a component; the `$cast()` determines which. If this child is a port, then use 'get_connected_to()' and 'get_provided_to()' to fill in any connections. Next print the port name, and iterate the

connections lists, printing connected-to and provided-to information. If this child was not a port, then it is a component, and `show_connectivity()` is called recursively with this child and depth incremented by 2. In this way the entire component hierarchy from the original component 'c' down to the last leaf will be visited.

```

function automatic void show_connectivity(
    uvm_component c = null;
    int depth = 0);

// Array of the children to iterate through.
uvm_component children[$];

// Nothing to do.
if (c == null)
    return;

begin
    ...
    type_name = "Component";
    $display("%s+ %s: %s", {depth{" "}},
        type_name, name);
end

// Fill up the array with all the
// children of 'c'.
c.get_children(children);

// Iterate for each child.
foreach (children[i]) begin
    // Cast to port_component_base to see if
    //this is a port.
    uvm_port_component_base port_component_base;

    // Test to see if this handle is a port.
    if ($cast(port_component_base, children[i]))
        begin
            uvm_port_list connected_to_list;
            uvm_port_list provided_to_list;

            // See what is connected.
            port_component_base.get_connected_to(
                connected_to_list);

            // See what is connected.
            port_component_base.get_provided_to(
                provided_to_list);

            // If there is something connected,
            // print the port name and the connections.
            if ((connected_to_list.size() > 0) ||
                (provided_to_list.size() > 0)) begin
                // Print the name of the port.
                $display("%s Port: %s", {depth+2{" "}},
                    children[i].get_full_name());
            end

            // Iterate across all the connections,
            // printing the thing this port is
            // connected to.
            foreach (connected_to_list[i])
                $display("%s Connected to Port: %s",
                    {depth+2+2{" "}},
                    connected_to_list[i].get_full_name());

            // Iterate across all the connections,
            // printing the thing this port is
            // provided to.
            foreach (provided_to_list[i])
                $display("%s Provided to Port: %s",
                    {depth+2+2{" "}},
                    provided_to_list[i].get_full_name());
        end
    else begin
        // It's a component. Recurse.

```

```

        show_connectivity(children[i], depth+2);
    end
end
endfunction

```

This code is simple but powerful. It recursively traverses the component and its children finding the connections to any ports, printing what they are connected to. It uses four API calls ('get_children()', 'get_connected_to()', 'get_full_name()', 'get_provided_to()') to produce a useful list of all components, their children and connections.

The netlisting code visits a potentially large data structure, printing it in human readable form. Data structures can also be explored one at a time. The code below has two ports, and just asks what they are connected to using the debug_connected_to() and debug_provided_to() calls.

```

fifo.put_export.debug_connected_to();
fifo.put_export.debug_provided_to();

fifo.get_export.debug_connected_to();
fifo.get_export.debug_provided_to();

```

These two snippets of code; a full netlister, and a partial netlister have explored just the basic "children" data structure in the uvm_component, and have used ports to show connectivity. The UVM has many other useful data structures waiting to be explored as a debug exercise for the reader.

G. Active Sequences

Most UVM testbenches will use sequences. The sequences will be used as stimulus generators. Each agent is likely to have at least one active sequence, if not more. These sequences are started, and work together to generate interesting stimulus conditions for testing. It is hard to know what sequences are running at any given time. An easy solution is to register the sequence when it is started, and then de-register it when it ends. A sequence has just the hooks needed – the pre_body() and post_body() tasks. By instrumenting the pre_body and post_body, we can keep track of what sequences are currently active.

The class 'active_sequences' below is a simple wrapper around an associative array indexed by sequence handle. When a sequence is started the function active_sequences::add(seq) should be called to register the sequence. When a sequence is ending the function active_sequences::remove(seq) should be called to de-register the sequence. At any given time, calling the function active_sequences::show() will print the list of currently active (currently registered) sequences.

```

class active_sequences;
// A list of sequences.
static uvm_sequence_base
    list[uvm_sequence_base];

// Add 'seq' to the list. If it is already in
// the list, just override the existing one.
static function void add(
    uvm_sequence_base seq);
    list[seq] = seq;
endfunction

// Remove 'seq' from the list. If it is not
// in the list, just ignore it.
static function void remove(

```

```

    uvm_sequence_base seq);
    if (list.exists(seq))
        list.delete(seq);
endfunction

static function void show(
    string file = `__FILE__,
    int line = `__LINE__);

    uvm_report_info("Seq",
        "Active Sequences", UVM_MEDIUM,
        file, line);
    foreach (list[seq])
        uvm_report_info("Seq",
            $sformatf(" %s", seq.get_full_name()),
            UVM_MEDIUM, file, line);
endfunction

...
endclass

```

The active_sequences class and the functions used to register and de-register can be used without any further support. They can be used in any location where the functionality is useful. The tasks pre_body and post_body are two such places.

Instrumenting pre_body and post_body can be done manually, inserting the calls to active_sequences::add() and active_sequences::remove() in each sequence. This is tedious and error prone. Putting this kind of functionality in a base class is a better idea. Many times there is a common base class which all sequences in the testbench derive from.

```

task pre_body();
// Begin a sequence
active_sequences::add(this);
super.pre_body();
endtask

task post_body();
// End a sequence
super.post_body();
active_sequences::remove(this);
endtask

```

If no common sequence base class exists, one can be created – the uvm_debug_sequence. All sequences should derive from this common sequence, and will inherit the pre_body() and post_body() functionality that calls keeps track of active sequences.

```

class uvm_debug_sequence #(
    type REQ = uvm_sequence_item,
    type RSP = REQ) extends uvm_sequence#(REQ, RSP);

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

task pre_body();
// Begin a sequence
active_sequences::add(this);
super.pre_body();
endtask

task post_body();
// End a sequence
super.post_body();
active_sequences::remove(this);
endtask

```

```
endclass
```

Using the new base class takes a little planning, changing what class is extended from.

```
class sequence_B #(type T = int)
  extends uvm_debug_sequence #(T);

  `uvm_object_param_utils(sequence_B#(T))

  const static string type_name =
    {"sequence_B#(", T::type_name, ")"};
  virtual function string get_type_name();
    return type_name;
  endfunction
  ...
```

At any time in the simulation, a list of the active sequences can be printed by calling `active_sequences::show`.

```
active_sequences::show();
```

A report will be generated like the report below. The report below shows four sequencers with sequences running.

```
e2_b.sequencer@@ggp_seq test.e2_a.sequencer.ggp_seq
e2_b.sequencer@@ggp_seq test.e1_b.sequencer.ggp_seq
e2_b.sequencer@@ggp_seq test.e1_a.sequencer.ggp_seq
e2_b.sequencer@@ggp_seq test.e2_b.sequencer.ggp_seq
e2_b.sequencer@@ggp_seq test.e2_b.sequencer.seq#(T)
e2_b.sequencer@@ggp_seq test.e2_a.sequencer.seq
e2_b.sequencer@@ggp_seq test.e1_b.sequencer.seq#(T)
e2_b.sequencer@@ggp_seq test.e1_a.sequencer.seq
```

H. Objects Allocated

In many testbenches transactions – classes – are created dynamically and created based on randomized control. It is not always possible to predict either how many classes will be constructed or what types they will be. Many problems can occur in such a testbench, including allocating too many classes or too many classes of a specific type.

Adding simple instrumentation each time a class is constructed; we can keep track of each allocation, and print a report when needed. This simple technique will tell us the maximum allocated, but won't tell us the maximum that actually was allocated at any given time. We're not able to track the garbage collection – the destruction of objects, so we don't know when objects are “unallocated”. This kind of information may become available in the future, but for now even just the total ever allocated and type is useful for tracking runaway allocations, expensive construction of already existing classes, or the uneven distribution of allocating one type over another.

When a class is allocated (constructed) we'll instrument the code with a call to register that fact:

```
function new(string name = "sequence_item_B");
  super.new(name);
  objects::add(get_type_name());
  id = g_id++;
endfunction
```

Calling `objects::add(get_type_name())` in each constructor of our simple sequences example, increments the count for this type. Calling `objects::show()` at the end of simulation produces the following report:

```
# Objects Allocated (by type)
# 324: sequence_item_A
```

```
# 80: sequence_A
# 80: sequence_B#(sequence_item_A)
# 26: sequence_B_parent#(sequence_item_A)
# 26: sequence_A_parent
...
# 4: driver2#(sequence_item_A)
# 2: driverA
...
# 2: sequence_A_greatgrandparent
# 1: test
```

The underlying data structure is a simple associative array, indexed by a string representing the type name of the allocated object. Each call to `object::add()` increments the corresponding array element. The `show()` puts the associative array elements into a SystemVerilog queue, and sorts it. The code then iterates the sorted queue, printing the allocation report seen above.

```
class objects;
  static int obj[string];

  static function void add(
    string type_name = "object");

    if (obj.exists(type_name))
      obj[type_name] += 1;
    else
      obj[type_name] = 1;
  endfunction

  static function void show();
    typedef struct {
      int count;
      string name;} item;

    item q[$];
    item i;

    $display("Objects Allocated (by type)");

    // Put each item in a queue.
    foreach (obj[type_name]) begin
      i.count = obj[type_name];
      i.name = type_name;
      q.push_front(i);
    end

    // Sort the queue.
    q.rsort with (item.count);

    // Print the items in descending order.
    foreach (q[x])
      $display("%6d: %s", q[x].count,
        q[x].name);
  endfunction
endclass
```

```
`define ALLOCATED_TYPE()\
  objects::add(get_type_name());
```

I. Conditional Breakpoint

When running simulation it is hard to stop at exactly the right place. This is especially hard for dynamic classes.

Such a conditional breakpoint might be “stop in packet processing after the ten thousandth packet has been processed if the current packet is of type ETH, and there are at least 10 errors”. These kinds of conditional breakpoints are sometimes most easily hard-coded into the SystemVerilog test – for example:

```
if ((packet_count > 10000) &&
```

```

(packet_type == ETH)    &&
(error_count > 10))
$stop();

```

Once simulation stops, you can examine the state of the packet processing. If stopping and examining is not possible, a summary report could be written out to the logfile, and simulation continued.

J. Dynamic message control

Sometimes certain debugging should start after many clocks, or after some condition is detected. This is a case where we want to run quietly – with little or no messages until the condition is reached. Once reached, we want to change the debug level or the verbosity level, producing many log messages. For example, after 100000 clocks, turn on debug messages in the AXI agent:

```

repeat (100000) @(posedge clk);
axi_agent_h.set_report_verbosity_hier(...);

```

K. Breakpoints in sequences

Sequences can be created at any time during simulation. Sequences can have short or long lifespans. Many instances of the same sequence can be constructed. Many different debug scenarios could be imagined, but with a small breakpoint library, sequences can be stopped based on their name, their type or their instance name (full path name).

The breakpoint library is a collection of code which uses the built-in UVM regular expression interface to match desired breakpoints with current sequence execution. For example, if the sequence named “seq” hits the “breakpoint”, then the breakpoint library will search to see if any regular expression matches the sequence name “seq” (or the type or the full name).

Using the breakpoint library is easy. Just call the breakpoint from the place where the check should occur. In a sequence, this is most likely at the start of the body() routine. Alternatively, the pre_body() could be instrumented to get all sequences instrumented.

Breakpoints are specified as plusargs on the simulation command line:

```

+bp=gp_*
+bp:type=.*greatgrand.*

```

The ‘greatgrandparent’ sequence is instrumented with `BP:

```

class sequence_B_greatgrandparent #(type T = int)
  extends sequence_B_parent #(T);
  `uvm_object_param_utils(
    sequence_B_greatgrandparent#(T))
  ...
  task body();
  ...
  `BP // The Breakpoint
  for(int i = 0; i < 3; i++) begin
    ...
  end
  ...
endtask

```

```

...
endclass

```

When the body() task is called, the code from the BP macro will be executed. The definition of BP is a call to the check function in the breakpoint pool library. (BPP::check()). Three arguments are passed in – the current values that are going to be checked against the regular expressions that are currently set as breakpoints. The three current values are get_full_name, get_name and get_type_name. Additionally, the file and line number are passed in for better debug messages.

```

// Macro: BP
//
// Must only be used in the context where the
// functions get_name(), get_full_name() and
// get_type_name() exist.
// For example, from within a sequence or within
// a component.
//
// Other macros can be created for other contexts.
//
`define BP if (BPP::check( get_full_name(), \
                          get_name(), \
                          get_type_name(), \
                          `FILE_, \
                          `__LINE__)) $stop;

```

Evaluating breakpoints, writing regular expressions, and getting the UVM names right is hard. The breakpoint pool comes with tracing and debug messages.

```

% grep BPP::trace transcript
[BPP::trace matched] .*greatgrand.* matched

% grep BPP::debug transcript
[BPP::debug - full_name] test.e2_b.sqr.seq#(T)
[BPP::debug - name] seq#(T)
[BPP::debug - type_name] seq_B_greatgrand#(seq_A)

```

We instrumented our body() task with a macro - `BP. We supplied command line arguments +bp:type=.*greatgrand.*. When we ran simulation we stopped at the exact line where BP was placed in the body() task – line 220:

```

# Break in Task
# envB_pkg/sequence_B_greatgrandparent::body at
# envB.sv line 220
# Stopped at envB.sv line 220

196 class sequence_B_greatgrandparent
...
217 task body();
...
220 `BP // The Breakpoint
221 for(int i = 0; i < 3; i++) begin

```

L. Random stability in debug

Any debug solution must be randomly stable. A randomly stable solution for debug means that when debug is “turned on”, the simulation results will be the same as without.

The following code is unstable with UVM 1.1. It is simple code that calls `uvm_info when the command line contains a non-zero setting for ‘x’ (+x=1). In the code, when x takes the value zero, then the if-statement will be true, and the macro `uvm_info will be called. This call to `uvm_info is what makes this code unstable – it changes the random number order by creating a new class object.

```

import uvm_pkg::*;
`include "uvm_macros.svh"
module top();
  int x, r;
  initial begin
    if ($value$plusargs("x=%0d", x) == 0)
      x = 0;
    if (x == 0)
      `uvm_info("DBG", "x is zero", UVM_MEDIUM)
      r = $urandom();
      $display("x=%0d, r=%0d", x, r);
    end
  endmodule

```

In UVM 1.1a this issue is fixed.

For user code, sometimes you must ensure your code stays randomly stable, by capturing the random state before a “destabilizing” event. Then after the destabilizing event, restore the random state. The following code is stable using UVM 1.1, adding the lines commented ‘ADDED’:

```

import uvm_pkg::*;
`include "uvm_macros.svh"
module top();
  int x, r;
  initial begin
    automatic process p; // Process - ADDED
    string p_rand; // State - ADDED

    if ($value$plusargs("x=%0d", x) == 0)
      x = 0;

    p = process::self(); // Capture - ADDED
    p_rand = p.get_randstate(); // Capture - ADDED

    if (x == 0)
      `uvm_info("DBG", "x is zero", UVM_MEDIUM)

    p.set_randstate(p_rand); // Restore - ADDED

    r = $urandom();
    $display("x=%0d, r=%0d", x, r);
  end
endmodule

```

M. Using a report catcher

A report catcher is a UVM construct which allows a message for the messaging system to be “caught”. For example, a report_catcher could be implemented to catch messages for a specific ID.

Once the report catcher is executing, it can choose to ignore or catch the message. If THROW is used, then the next report_catcher will process this message. If CATCH is used, then this message will not be processed by any other report_catcher – the message has been caught.

A simple report catcher skeleton is listed below. It simply uses the API to retrieve message attributes (get_id(), get_fname(), etc), then it returns “THROW”.

```

class my_report_catcher
  extends uvm_report_catcher;

  string id;
  string filename;
  string name;
  string message;
  int line;
  int verbosity;
  uvm_severity severity;
  uvm_report_object client;

```

```

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

```

```

function action_e catch();
  uvm_severity_type usv;

  id = get_id();
  filename = get_fname();
  line = get_line();
  severity = get_severity();
  verbosity = get_verbosity();
  message = get_message();

```

```

  client = get_client();
  name = client.get_full_name();

```

```

  usv = uvm_severity_type'(severity);

```

```

  // Process this message.
  // Decide THROW or CATCH.
  ...

```

```

  return THROW;
endfunction

```

```

endclass

```

Using a report_catcher is easy. It must be constructed, and then it must be added to the list of report catchers managed by the reporting system (uvm_report_cb::add()). A report catcher can be prepended (UVM_PREPEND) or appended (UVM_APPEND) to the existing report catcher list.

```

class test extends uvm_test;
  ...
  my_report_catcher my_report_catcher_inst;
  my_report_catcher_inst =
    new("my_report_catcher_inst");
  uvm_report_cb::add(null,
    my_report_catcher_inst, UVM_APPEND);
  ...

```

N. Using a report server

A report server is a UVM construct which formats the reports in the messaging system. The report server below formats the name and id to fixed widths. The widths are programmable.

```

class my_report_server extends uvm_report_server;
  int name_width = 20;
  int id_width = 20;

```

```

function string pad(string s, int width);
  if ( s.len() == width )
    return s;

```

```

  // s is short. Pad at the end.
  if ( s.len() < width )
    return {s, {(width - s.len()){" "}}};
  else

```

```

    // s is too long. truncate.
    return s.substr(s.len()-width, s.len()-1);
endfunction

```

```

function string compose_message(
  uvm_severity severity,
  string name,
  string id,
  string message,
  string filename,
  int line
);
  // Make the width be exactly name_width
  // and id_width.

```

```

name = pad(name, name_width);
id   = pad(id,   id_width);

return super.compose_message(
    severity, name, id, message, filename, line);
endfunction
endclass

```

Using a report server is easy. It must be constructed. After it is constructed it can be configured. In this case, the configuration consists of setting the name and id widths. Name is set to 28 characters, and id is set to 20 in this example. Finally, the new report server is registered using 'set_server()'.

```

class test extends uvm_test;

// Make my report server.
begin
    my_report_server my_report_server_inst;
    my_report_server_inst = new();

// Configure.
my_report_server_inst.name_width = 28;
my_report_server_inst.id_width   = 20;

// Set.
uvm_report_server::set_server(
    my_report_server_inst);
end

```

VII. SUMMARY

Classes are dynamic objects in SystemVerilog. A traditional debugging approach using standard Verilog tools may not be sufficient for debugging classes. Classes are really software – they have dynamic life spans, they have user defined types, they have object-oriented properties, like inheritance and virtual functions. Debugging SystemVerilog

classes requires adopting new debugging techniques. Additionally, debugging UVM testbenches for best results requires understanding and traversing the UVM class library data structures.

Successful debug depends on getting organized.

Once the infrastructure is in place, messages can be created in log files and analyzed with standard programming tools like grep, awk and perl. For example, find all drivers, but not AXI drivers:

```
grep driver transcript | grep -v AXI
```

Reports can be generated gathering statistics about the distribution of packet type over certain periods of time. Transfer rates can be calculated. The possibilities are limitless, but first you have to get organized, and plan your debug.

VIII. REFERENCES

- [1] SystemVerilog LRM Std 1800-2009
- [2] UVM – www.accellera.org
- [3] Coding Guidelines en.wikipedia.org/wiki/Coding_conventions
- [4] Google Style Guide, <http://code.google.com/p/google-styleguide/>
- [5] Stu Sutherland, *SystemVerilog for Design*
- [6] Chris Spear, *SystemVerilog for Verification*
- [7] Doulos, *SV Golden Reference Guide*
- [8] Adam Erickson, "Are OVM & UVM Macros Evil? A Cost-Benefit Analysis", DVCON 2011
- [9] Mark Glasser, "Advanced Testbench Configuration with Resources", DVCON 2011

IX. UVM CLASS LIBRARY

The UVM Class library is large. There are many classes, with many controls and options (The uvm-1.1a contains 316 class definitions in 134 files for a total of 67,298 lines of code). Some of the most interesting options, switches, flags and functions are listed below by UVM class name. This is not an exhaustive list, nor necessarily a useful list; just an interesting list for future investigation in the workings of the UVM.

A. *uvm_root*

```
+UVM_TESTNAME
+uvm_set_verbosity
+UVM_TIMEOUT
+UVM_SET_INST_OVERRIDE
+UVM_SET_TYPE_OVERRIDE
+UVM_SET_CONFIG_INT
+UVM_SET_CONFIG_STRNG
+UVM_MAX_QUIT_COUNT
+UVM_DUMP_CMDLINE_ARGS
+UVM_VERBOSITY
uvm_root::find()
uvm_root::find_all()
uvm_root::print_topology()
```

B. *UVM_COMPONENT*

```
+uvm_set_action=<comp, id, severity, action>
+uvm_set_severity=<comp, id, orig severity, new severity>
uvm_component::get_parent()
uvm_component::get_children()
uvm_component::lookup()
uvm_component::get_depth()
uvm_component::phase_started()
uvm_component::phase_ready_to_end()
uvm_component::phase_ended()
uvm_component::check_config_usage()
uvm_component::print_config_settings()
uvm_component::print_config()
uvm_component::print_config_with_audit()
uvm_component::print_config_matches = 1
uvm_component::raised()
uvm_component::dropped()
uvm_component::all_dropped()
uvm_component::print_override_info()
uvm_component::set_report_id_verbosity_hier()
uvm_component::set_report_severity_id_verbosity_hier()
uvm_component::set_report_severity_action_hier()
uvm_component::set_report_id_action_hier()
uvm_component::set_report_severity_id_action_hier()
uvm_component::set_report_severity_file_hier()
uvm_component::set_report_default_file_hier()
uvm_component::set_report_id_file_hier()
uvm_component::set_report_severity_id_file_hier()
uvm_component::set_report_verbosity_level_hier()
uvm_component::pre_abort()
uvm_component::set_config_object()
uvm_component::check_config_usage()
uvm_component::print_config
```

C. *uvm_object*

```
uvm_object::get_inst_id() (incremented on new object)
uvm_object::get_inst_count() (total count until now)
uvm_object::get_type_name()
```

D. *uvm_CONFIG_DB*

```
+UVM_CONFIG_DB_TRACE
uvm_resource_db_options::is_tracing()
    (`uvm_info(..., UVM_LOW)
uvm_config_db_options::turn_on_tracing()
uvm_config_db_options::turn_off_tracing();
```

E. *Uvm_resource_db*

```
+UVM_RESOURCE_DB_TRACE
dump()
```

F. *uvm_resource*

```
uvm_resource_opts::trace_on_auditing
uvm_resource::print_resources()
uvm_resource::print_accessors()
```

G. *uvm_resource_pool*

```
uvm_resource_pool::dump_get_records()
uvm_resource_pool::find_unused_resources()
uvm_resource_pool::print_resources()
uvm_resource_pool::dump()
```

H. *uvm_callback*

```
uvm_callback::display()
`uvm_cb_trace_noobj()
```

I. *uvm_event*

```
uvm_event::add_callback()
uvm_event::do_print()
```

J. *uvm_factory*

```
uvm_factory::debug_create_by_type
uvm_factory::debug_create_by_name
uvm_factory::m_debug_pass
uvm_factory::print
```

K. *uvm_objections*

```
uvm_objections::trace_mode
Report ID: OBJTN_TRC
uvm_objections::raised()
uvm_objections::dropped()
uvm_objections::all_dropped()
uvm_objections::get_objectors()
uvm_objections::display_objections()
```

L. *uvm_phase*

```
`UVM_PH_TRACE
+UVM_PHASE_TRACE → m_phase_trace
`uvm_info(..., UVM_DEBUG)
uvm_phase::get_state()
    wait(get_state() ... ) → change phase state
uvm_phase::wait_for_state()
```

M. *uvm_sequence_base*

```
uvm_sequence_base::m_sequence_state
uvm_sequence_base::get_sequence_state()
uvm_sequence_base::wait_for_sequence_state()
uvm_sequence_base::pre_start()
uvm_sequence_base::pre_body()
uvm_sequence_base::pre_do()
uvm_sequence_base::mid_do()
uvm_sequence_base::body()
uvm_sequence_base::post_do()
uvm_sequence_base::post_body()
uvm_sequence_base::post_start()
```

N. *uvm_sequencer_base*

```
uvm_sequencer_base::print()
uvm_sequencer_base::convert2string()
`uvm_info("PHASESEQ", ..., UVM_FULL)
```

X. EXAMPLE SEQUENCE INSTRUMENTATION

The sequence 'sequenceB' below is an example sequence with instrumentations. Not all instrumentations shown here are required for good debug, but these are listed as potential sequence debug instrumentation points.

First, the sequence extends the `uvm_debug_sequence`, inheriting any instrumentation from it.

The sequence defines its `type_name`, which is hard for parameterized objects. The sequence uses the `\ALLOCATED_TYPE` macro to register the construction of this object. The sequence is managing an id for each different parameterized sequence that is created. The sequence defines `pre_body()` and `post_body()`, using the `\DEBUG` macro for output. The sequence uses the `\uvm_info` macros and `$display` in the task body(), providing breadcrumbs to trace execution flow. The sequence uses the `\BP` macro to allow breakpoint access to this sequence. Finally, the sequence defines `do_record` in a way that will record the actual types of the variables into the transaction database, instead of recording a "large enough" bit vector as the type.

```
class sequence_B #(type T = int)
  extends uvm_debug_sequence #(T);
  `uvm_object_param_utils(sequence_B#(T))

  const static string type_name =
    {"sequence_B#(", T::type_name, ")"};

  virtual function string get_type_name();
  return type_name;
endfunction

  int id;
  int var_sequence_B;
  int iteration;
  static int g_id = 0;

  function new(string name = "sequence_B");
    super.new(name);
    `ALLOCATED_TYPE
    id = g_id++;
  endfunction

  // `uvm_debug_sequence_body
  virtual task pre_body();
  `DEBUG("pre_body()")
  super.pre_body();
endtask

  virtual task post_body();
  `DEBUG("post_body()")
  super.post_body();
endtask

  task body();
  T t;
  `uvm_info(get_type_name(),
    "Starting...", UVM_MEDIUM)
  `DEBUG($psprintf("My type_name is %s",
    get_type_name()))

  `BP

  begin
    int simple_int = -1;
    if (!m_sequencer.get_config_int(
      "simple_int", simple_int))
      $display("GET CONFIG FAILED");
    else
```

```
      $display("GET CONFIG OK");

    `uvm_info("SEQ_B",
      $psprintf("simple_int=%0d", simple_int),
      UVM_MEDIUM)
  end

  for(int i = 0; i < 3; i++) begin
    t = new($psprintf("t%0d", i));
    var_sequence_B = t.id;
    start_item(t);
    finish_item(t);
  end
  `uvm_info(get_type_name(),
    "Finishing...", UVM_MEDIUM)
endtask

function void do_record(uvm_recorder recorder);
  super.do_record(recorder);
  `uvm_record_field("id", id);
  `uvm_record_field("iteration", iteration);
  `uvm_record_field("var_sequence_B",
    var_sequence_B);
endfunction
endclass
```

XI. BREAKPOINT POOL LIBRARY

The breakpoint pool library, the other classes and macros mentioned in this paper are available. Please contact the authors.

```
// Class: BP
//
// A breakpoint. Really a list of strings
// (regular expressions) that will be used to
// match against. When a match occurs that
// means a breakpoint has been "hit".
//
// This class is not normally used directly,
// but rather from the BPP class.
//
class BP;
  typedef struct { string name; } bp_t;
  bp_t breakpoints[$];

  // add_breakpoint()
  // Put the argument in our list.
  function void add_breakpoint(string str);
    bp_t bp;
    bp.name = str;
    breakpoints.push_front(bp);
  endfunction

  // check()
  //
  // Returns 1 if we matched; the 'match'
  // string has matched something in the
  // breakpoint list.
  // Returns 0 if nothing matched.
  // Return the regular expression that matched
  // in the argument 'thing_that_matched'.
  //
  // Uses regular expression matching.
  //
  function int check(string match = "",
    output string thing_that_matched);
    int nomatch;

    if (match == "")
      return 0; // We didn't match

    foreach (breakpoints[j]) begin
      nomatch = uvm_re_match(breakpoints[j].name,
        match);
      if (nomatch == 0) begin
        thing_that_matched = breakpoints[j].name;
```

```

        return 1; // Meaning we matched...
    end
end
return 0; // We didn't match.
endfunction

// print()
// Print all the breakpoints.
function void print();
    foreach (breakpoints[j])
        $display(" Breakpoint '%s'",
            breakpoints[j].name);
    endfunction
endclass

// Class: BPP
//
// A breakpoint-pool.
//
// Collects together three sets of breakpoints.
// Name, Type and Full name.
//
// This is a static class, and is normally
// used from macros.
//
class BPP;

    // Displays trace messages for check().
    static bit trace = 1;
    // Displays debug messages for check().
    static bit debug = 1;

    static BP by_type      = new();
    static BP by_name      = new();
    static BP by_full_name = new();

    // Fill in the by type
    static function void set_by_type(
        string breakpoint);
        by_type.add_breakpoint(breakpoint);
    endfunction

    // Fill in the by name
    static function void set_by_name(
        string breakpoint);
        by_name.add_breakpoint(breakpoint);
    endfunction

    // Fill in the by full name.
    static function void set_by_full_name(
        string breakpoint);
        by_full_name.add_breakpoint(breakpoint);
    endfunction

    // Calling trace_on causes trace messages to
    // come out.
    static function void trace_on();
        trace = 1;
    endfunction

    // Calling trace_off turns trace messages off.
    static function void trace_off();
        trace = 0;
    endfunction

    // Calling debug_on causes debug messages to
    // come out.
    static function void debug_on();
        debug = 1;
    endfunction

    // Calling debug_off turns debug messages off.
    static function void debug_off();
        debug = 0;
    endfunction

    // Trick to get the command line parsed

```

```

// without any intervention from the user.
static bit args_parsed = parse_args();

// parse_args()
//
// Legal syntax:
//
// +bp:type=
// +bp:full=
// +bp=
//
static function bit parse_args();
    string list[$];
    uvm_cmdline_processor clp;
    clp = uvm_cmdline_processor::get_inst();

    // By Type => +bp:type=
    clp.get_arg_values("+bp:type=", list);
    foreach (list[i])
        BPP::set_by_type(list[i]);

    // By Full Name => +bp:full=
    clp.get_arg_values("+bp:full=", list);
    foreach (list[i])
        BPP::set_by_full_name(list[i]);

    // By Name => +bp=
    clp.get_arg_values("+bp=", list);
    foreach (list[i])
        BPP::set_by_name(list[i]);

    return 1;
endfunction

// check()
//
// returns 0 for no match.
// returns 1 for full name match.
// returns 2 for name match.
// returns 3 for type name match.
//
// Usually called from a macro:
//
// check(get_full_name(),
//       get_name(),
//       get_type_name(),
//       `__FILE__,
//       `__LINE__)
//
static function int check(
    string full_name = "",
    string name      = "",
    string type_name = "",
    string file      = "",
    int line         = 0);

    string thing_that_matched;
    int ret;

    if (debug) begin
        `uvm_info("BPP::debug - full_name",
            full_name, UVM_MEDIUM)
        `uvm_info("BPP::debug - name",
            name, UVM_MEDIUM)
        `uvm_info("BPP::debug - type_name",
            type_name, UVM_MEDIUM)
    end

    if (by_full_name.check(
        full_name, thing_that_matched))
        ret = 1;
    else if (by_name.check(
        name, thing_that_matched))
        ret = 2;
    else if (by_type.check(
        type_name, thing_that_matched))
        ret = 3;

```

```
else
    ret = 0;

// Tell what matched
if ((trace) && (ret != 0))
    `uvm_info("BPP::trace matched",
        $sformatf("%s matched",
            thing_that_matched), UVM_MEDIUM)
return ret;
endfunction

// print()
//
```

```
// Debug/information.
//
static function void print();
    $display("By Type");
    by_type.print();
    $display("By Name");
    by_name.print();
    $display("By Full Name");
    by_full_name.print();
endfunction
endclass
```