Are You Smarter Than Your Testbench?
With a little work you can be.

Rich Edelman
Mentor Graphics
Fremont, CA

Raghu Ardeishar
Mentor Graphics
McLean, VA

***Abstract*- As design size increases complexity, testbenches complexity grows at least as fast. Today's testbench is as complicated as the design itself, and care must be taken to understand it from both a performance and functionality point of view. This paper will discuss ways to keep check on the testbench performance and to understand the functionality being implemented and the effectiveness of the tests.**

Keywords: testbench debug, SystemVerilog UVM, SystemVerilog class debug

I. INTRODUCTION

Modern SystemVerilog [1] UVM [2] testbenches have grown to be large, complex systems in themselves. They can be equally as hard to debug as the underlying hardware being verified. By instrumenting your testbench in a few places with a few "hooks", your testbench will be more debuggable, more transparent and easier to manage. This paper will illustrate a few such hooks – you can imagine many more. The basic philosophy is to keep the hooks as simple as possible – as few lines as possible.

II. ARCHITECTURAL DECISIONS

The solutions presented in this paper require change – instrumentation. Either changing the testbench, the UVM source code or some combination. From this instrumentation an analysis database will be created which can answer questions like – "How many sequences are running on agentX?"

Instrumenting the testbench

The testbench instrumentation will naturally require that the testbench be changed. Lines of code will change – for example from start_item(t) to `S_START_ITEM(t). New lines may be added to track interesting places. Extra `uvm_info() lines may be added.

Instrumenting the UVM

The UVM can be changed so that base classes get all the new functionality, and the testbench doesn't know anything has changed. The base classes can call into the analysis database. Changing the UVM is a convenient solution for the testbench creator – the UVM is changed one time, and all testbenches get the benefit automatically. The problem with this solution is that the UVM does change over time, and worse, some vendors provide custom UVM implementations. Changing the UVM also removes some of the debug transparency. When the testbench is changed to help improve debug, it is obvious what is being instrumented, and what it information will be available.

Analysis Database

An analysis database is created which can answer questions. This analysis database will keep track of what's going on in the testbench. An implementation could create a database of all the handles – sequences and sequence_items that are generated in the testbench. As simulation proceeds, analysis can be performed. The problem with this solution is that the database would be retaining handles to objects that could possibly have been garbage collected. The function of the garbage collector is beyond the scope of this paper, but it is not a good idea to change the way objects are garbage collected. It may cause debug situations to change from non-debug situations.

Another implementation could build a database which does not use handles, but rather uses information about the objects.

This solution – change the testbench, create a stand-alone analysis database, do not retain object handles

In the solution presented in this paper, there is no requirement to change the UVM, and there is no storing of handles. The testbench will be changed, and the analysis database will be built without holding on to any object handles.

The stand-alone database is created which represents the sequences and sequence_items executions. For example, when a sequence is started, the sequence type_name and instance name are put into a table for later type and name matching.

There are three kinds of instrumentation considered here. The first is simple `uvm_info() instrumentation with recommendations on how to make sure it does not slow your simulation. The second is instrumentation to understand the "allocation pattern" of sequences and sequence items. The third is instrumentation to build a trail that specifies the various source code locations that a class handle - usually a sequence_item - has occupied. This occupancy list is very handy to understand where a transaction has been and when it was there.

The solution is a small amount of SystemVerilog code with a few simple access routines spanning 5 files and 350 lines of code.

III. VERBOSE MODE

One easy solution to testbench visibility is print statements. Using $display() is a common debug technique. `uvm_info() is advertised as a smarter replacement. It is smarter in that it can be turned off – so it does less work when not needed. It helpfully prints the time and scope, along with a user defined message. But the catch is that even a turned off `uvm_info() does too much work.

Many testbenches use UVM_LOW `uvm_info() messages for very important messages. These are messages that should almost never be turned off. UVM_MEDIUM `uvm_info() messages are for “normal” messages. They are informative, but not too verbose, and can easily be turned off. UVM_HIGH `uvm_info() messages are for very chatty, very verbose messages used in detailed debugging. These messages are off by default. They get turned on to do detailed debug. The example below shows a code snippet which uses the `uvm_info() calls.

```
// `uvm_info("INIT", "Hello World!  (LOW)", UVM_LOW)
if (uvm_report_enabled(UVM_LOW,UVM_INFO,"INIT"))
  uvm_report_info ("INIT", "Hello World!  (LOW)", UVM_LOW, `__FILE__, `__LINE__);

// `uvm_info("INIT", "Hello World! (MEDIUM)", UVM_MEDIUM)
if (uvm_report_enabled(UVM_MEDIUM,UVM_INFO,"INIT"))
  uvm_report_info ("INIT", "Hello World! (MEDIUM)", UVM_MEDIUM, `__FILE__, `__LINE__);

// `uvm_info("INIT", "Hello World!  (HIGH)", UVM_HIGH)
if (uvm_report_enabled(UVM_HIGH,UVM_INFO,"INIT"))
  uvm_report_info ("INIT", "Hello World!  (HIGH)", UVM_HIGH, `__FILE__, `__LINE__);
```

This code has had the macros expanded for readability for this paper. Each call to `uvm_info() is a call to check for “applicability” [uvm_report_enabled()] and then a call to format and print the message [uvm_report_info()]. If a message is not applicable, then the call to format and print the message is never executed – that’s the advertised savings. Many testbenches run in regression mode turn off UVM_MEDIUM messages and above. The idea is that a regression will run faster if it is not printing so much information. That is absolutely true. But a turned off `uvm_info() is still doing work. For example, a message like the message on lines 19 and 20 – a call the a UVM_HIGH `uvm_inf() still does a lot of work, despite being turned off.

There are two problems. The first problem is that the UVM_HIGH applicability check takes roughly 17 ‘single-step’ executions.

```
t.sv 19  Module top
 src/base/uvm_globals.svh:118        uvm_report_enabled
 src/base/uvm_root.svh:285           uvm_pkg/uvm_root::get
 src/base/uvm_root.svh:290           uvm_pkg/uvm_root::get
 src/base/uvm_root.svh:291           uvm_pkg/uvm_root::get
 src/base/uvm_globals.svh:119        uvm_report_enabled
 src/base/uvm_report_object.svh:464  uvm_pkg/uvm_report_object::uvm_report_enabled
 src/base/uvm_report_object.svh:429  uvm_pkg/uvm_report_object::get_report_verbosity_level
 src/base/uvm_report_handler.svh:285 uvm_pkg/uvm_report_handler::get_verbosity_level
 src/base/uvm_report_handler.svh:292 uvm_pkg/uvm_report_handler::get_verbosity_level
 src/base/uvm_pool.svh:134           uvm_pkg/uvm_pool::exists
 src/base/uvm_pool.svh:135           uvm_pkg/uvm_pool::exists
 src/base/uvm_report_handler.svh:296 uvm_pkg/uvm_report_handler::get_verbosity_level
 src/base/uvm_report_handler.svh:298 uvm_pkg/uvm_report_handler::get_verbosity_level
 src/base/uvm_report_object.svh:430  uvm_pkg/uvm_report_object::get_report_verbosity_level
 src/base/uvm_report_object.svh:466  uvm_pkg/uvm_report_object::uvm_report_enabled
 src/base/uvm_report_object.svh:469  uvm_pkg/uvm_report_object::uvm_report_enabled
 src/base/uvm_globals.svh:120        uvm_report_enabled
```

This is code that executes just to not print anything. A UVM_HIGH message is a noisy, chatty message – perhaps a message on every clock edge in a monitor or driver. This high frequency message is not printed normally, but the checking code is executed. A faster, smarter testbench could redefine `uvm_info() for regression runs where UVM_LOW is the normal verbosity. In the code below, if the VERBOSITY was UVM_MEDIUM or UVM_HIGH, just the simple “if (VERBOSITY == UVM_LOW)” will be executed, instead of 17 single-step equivalents.

```
`define uvm_info(ID,MSG,VERBOSITY) \
   begin \
     if ( VERBOSITY == UVM_LOW ) \
       if (uvm_report_enabled(VERBOSITY,UVM_INFO,ID)) \
         uvm_report_info (ID, MSG, VERBOSITY, `uvm_file, `uvm_line); \
   end
```

The second problem is that the call to `uvm_info() actually calls uvm_report_enabled() one more time. It is checking applicability twice. Once in the if() and once deep inside uvm_report_info() [uvm_report_server::report()].

So `uvm_info() is the right way to instrument a testbench, but just be careful, and for regressions have a redefined `uvm_info() that is very quiet.

Finally, if you are using UVM 1.2, at last count it was 23 ‘single-step executions’ instead of the 17 for uvm-1.1d.

IV. ALLOCATION DEBUG – WHO MADE THAT?

Sometimes in a runaway testbench there is a “fountain” which generates many, many transactions. The testbench may be running properly, but there are an (unnecessarily) large number of transactions being allocated. This is a particularly hard bug to find, since things are actually working fine. One symptom might be that memory usage is high, or that the system “seems slow”, but the problem is that it is hard to know when this is happening.

With the stats instrumentation these fountains can be quickly identified, either by file and line number, by sequence instance, by sequence_item type or by sequence type. A quick glance at the report below shows that the xyz_sequence is quite active, as is the xyz_item.

```
# Stats: Counted by file_line
# Stats:                  1 : ../abc_pkg/abc_seq_lib.svh:58
# Stats:                 48 : ../abc_pkg/abc_seq_lib.svh:79
# Stats:                  2 : ../xyz_pkg/xyz_seq_lib.svh:56
# Stats:                 97 : ../xyz_pkg/xyz_seq_lib.svh:74
# Stats: Counted by seq_full_name
# Stats:                 49 : uvm_test_top.env.agent1.sequencer.seq1
# Stats:                 50 : uvm_test_top.env.agent2.sequencer.seq2
# Stats:                 49 : uvm_test_top.env.agent3.sequencer.seq3
# Stats: Counted by seq_item_type_name
# Stats:                 48 : abc_item
# Stats:                  1 : abc_reset_item
# Stats:                 97 : xyz_item
# Stats:                  2 : xyz_reset_item
# Stats: Counted by seq_type_name
# Stats:                 49 : abc_even_sequence
# Stats:                 99 : xyz_sequence
```

At the end of simulation, the report has changed:

```
# Stats: Counted by file_line
# Stats:                  2 : ../abc_pkg/abc_seq_lib.svh:141
# Stats:                  2 : ../abc_pkg/abc_seq_lib.svh:58
# Stats:               1998 : ../abc_pkg/abc_seq_lib.svh:79
# Stats:                  4 : ../xyz_pkg/xyz_seq_lib.svh:56
# Stats:               3996 : ../xyz_pkg/xyz_seq_lib.svh:74
# Stats: Counted by seq_full_name
# Stats:                  2 : uvm_test_top.env.agent1.sequencer.seq
# Stats:               2000 : uvm_test_top.env.agent1.sequencer.seq1
# Stats:               2000 : uvm_test_top.env.agent2.sequencer.seq2
# Stats:               2000 : uvm_test_top.env.agent3.sequencer.seq3
# Stats: Counted by seq_item_type_name
# Stats:               2000 : abc_item
# Stats:                  2 : abc_reset_item
# Stats:               3996 : xyz_item
# Stats:                  4 : xyz_reset_item
# Stats: Counted by seq_type_name
# Stats:                  2 : abc_dump
# Stats:               2000 : abc_even_sequence
# Stats:               4000 : xyz_sequence
```

We can see that ‘seq1’, ‘seq2’ and ‘seq3’ have each created 2000 objects. That seems correct for the testbench plan. The source code for the sequence which starts these sequence_items is below. The normal calls to create(), start_item() and finish_item() have been replaced with calls to the macros.

```
//               item = abc_item::type_id::create($sformatf("item%0d", transaction_count++));
`S_CREATE_OBJECT(item,  abc_item,                 $sformatf("item%0d", transaction_count++))

// start_item(item);
`S_START_ITEM(item)

// Randomize ...

// finish_item(item);
`S_FINISH_ITEM(item)
```

The three macros, `S_CREATE_OBJECT(), `S_START_ITEM() and `S_FINISH_ITEM() are replacements for the original calls to create(), start_item() and finish_item(). The `S_CREATE_OBJECT() macro calls the create() routine and then registers the new object in the analysis database.

```
`define S_CREATE_OBJECT(lhs, type_t, typename) \
  lhs = type_t::type_id::create(typename); \
  stats_pkg::created_object(`__FILE__, `__LINE__, typename, lhs, this);
```

The `S_START_ITEM() and `S_FINISH_ITEM() macro calls implement the same start_item() and finish_item() as the original, but with a before and after call to the analysis database.

```
`define S_START_ITEM(seq_item)                          \
  begin                                               \
  stats_pkg::follow_sequence_item(`__FILE__, `__LINE__, "start_item  start", seq_item, this); \
  start_item(seq_item);                               \
  stats_pkg::follow_sequence_item(`__FILE__, `__LINE__, "start_item  done ", seq_item, this); \
  end

`define S_FINISH_ITEM(seq_item)                         \
  begin                                               \
  stats_pkg::follow_sequence_item(`__FILE__, `__LINE__, "finish_item start", seq_item, this); \
  finish_item(seq_item);                              \
  stats_pkg::follow_sequence_item(`__FILE__, `__LINE__, "finish_item done ", seq_item, this); \
  end
```

Using this family of three macros keeps track of each sequence_item, from creation, to start, and to finish. A sequence of "interest" would be instrumented with the macros above replacing the normal create(), start_item(), finish_item(). Not every sequence needs to be instrumented – only the suspect ones.

V. WHAT'S THE LOAD AVERAGE?

Load average [3] is a concept familiar to UNIX users. It is the number of active processes running on the CPUs. It averages the CPU usage across time, and can give an indication not only how busy the machine is now, but how busy it has been in the past. The load average here is modeled after this UNIX load average.

A "load average" is useful in the UVM for limited resources like drivers. How many outstanding transactions are on the driver?

```
# loadav -------------------
# uvm_test_top.env.agent1: loadav [   5.4    1.7   0.53]
# uvm_test_top.env.agent2: loadav [     5    1.6   0.49]
# uvm_test_top.env.agent3: loadav [     5    1.6   0.49]
```

By examining the load average over time you can know if a driver gets more or less heavily loaded. Or if there is unusual loading – too many transactions coming into one driver, and not spread equally.

The load average calculation is done at equal intervals during simulation – like an interrupt timer. This timer can be adjusted as needed. Calculate the load every clock edge:

```
  always begin
    repeat ( 1) @(posedge clk);
    stats_pkg::stats.calc_load(1);
  end
```

The load average can be printed at any time, by calling the loadav() function, either from the simulation or a control script. For example the code below calls the load average every 100 clocks – like a heartbeat function.

```
  always begin
    repeat (100) @(posedge clk);
    stats_pkg::stats.loadav();
    end
```

VI. WHAT SEQUENCES ARE USED?

Keeping track of which sequences are used can help figure out if sequences are being unnecessarily or incorrectly created. The UVM sequence creation and start procedure is quite slow – creating a sequence just to send one transaction for example is very expensive. In the code example a very few sequences are created which in turn create the thousands of sequence_items.

```
  # Sequence     4 : 'xyz_sequence' (last started at 0)
  # Sequence     2 : 'abc_even_sequence' (last started at 0)
  # Sequence     2 : 'abc_dump' (last started at 79790)
  # Sequence     1 : 'b' (last started at 179810)
  # Sequence     1 : 'a' (last started at 0)
```

In order to keep track of what sequence are used, change the ‘sequence.start()’ call to instead use a macro.

seq.start(sqr) changes to `S_START_SEQUENCE(seq, sqr)

For example:

```
    `S_START_SEQUENCE (seq1, env.agent1.sequencer) // seq1.start(env.agent1.sequencer);
    `S_START_SEQUENCE (seq2, env.agent2.sequencer) // seq2.start(env.agent2.sequencer);
    `S_START_SEQUENCE (seq3, env.agent3.sequencer) // seq3.start(env.agent3.sequencer);
```

The macro implementation accesses the database before and after an embedded call to seq.start().

```
    `define S_START_SEQUENCE(seq, sqr)                  \
       begin                                           \
       stats_pkg::follow_sequence(`__FILE__, `__LINE__, "Starting", seq, sqr); \
       seq.start(sqr);                                 \
       stats_pkg::follow_sequence(`__FILE__, `__LINE__, "  Ending", seq, sqr); \
       end
```

This instrumentation enable the analysis database to keep track of the beginning and ending of all instrumented sequences.

VII. WHERE HAVE MY TRANSACTIONS BEEN?

The SystemVerilog UVM class based testbench creates objects and “sends” those objects to other parts of the testbench. For example, a sequence might create a sequence_item (transaction) and send it to the driver. The driver will get the transaction and turn it into pin wiggles on the bus. A monitor might see the pin wiggles and turn them into a transaction. That monitored transaction might get sent out an analysis port to a waiting subscriber – maybe a scoreboard, checker or other analysis code. A large testbench can have many objects being created and those objects can go to many different places. It would be helpful to know where a transaction has been, where it came from and how long it stayed in any one place.

In the report below, the transaction has been tracked from construction at line 59 in abc_seq_lib.svh, to the driver lines 46 and 65, then on to the checker system.

```
  @   0: <reset_item> Construct= (abc_pkg/abc_seq_lib.svh:59)

  @   0: <reset_item> (abc_pkg/abc_seq_lib.svh:61)       [uvm_test_top.env.agent1.sequencer.seq1]
  @   0: <reset_item> (abc_pkg/abc_driver.svh:46)        [uvm_test_top.env.agent1.driver]
  @2570: <reset_item> (abc_pkg/abc_driver.svh:65)        [uvm_test_top.env.agent1.driver]
  @2570: <reset_item> (abc_pkg/abc_checkerboard.svh:24) [uvm_test_top.env.agent1.checkerboard2_0]
  @2570: <reset_item> (abc_pkg/abc_seq_lib.svh:63)       [uvm_test_top.env.agent1.sequencer.seq1]
  @2580: <reset_item> (abc_pkg/abc_checkerboard.svh:46) [uvm_test_top.env.agent1.checkerboard2_0]
  @2583: <reset_item> (abc_pkg/abc_checkerboard.svh:24) [uvm_test_top.env.agent1.checkerboard2_1]
  @2593: <reset_item> (abc_pkg/abc_checkerboard.svh:46) [uvm_test_top.env.agent1.checkerboard2_1]
  @2599: <reset_item> (abc_pkg/abc_checkerboard.svh:24) [uvm_test_top.env.agent1.checkerboard2_2]
  ...
```

In the report below, the transaction has been tracked from the construction at line 48 in the abc_monitor.svh to the checker system and the coverage collector.

```
@   0: <t0> Construct= (abc_pkg/abc_monitor.svh:48)

@2660: <t0> (abc_pkg/abc_monitor.svh:60)       [uvm_test_top.env.agent1.monitor]
@2660: <t0> (abc_pkg/abc_checkerboard.svh:24) [uvm_test_top.env.agent1.checkerboard0]
@2660: <t0> (abc_pkg/abc_coverage.svh:67)      [uvm_test_top.env.agent1.coverage]
@2660: <t0> (abc_pkg/abc_coverage.svh:55)      [uvm_test_top.env.agent1.coverage]
@2670: <t0> (abc_pkg/abc_checkerboard.svh:46) [uvm_test_top.env.agent1.checkerboard0]
@2671: <t0> (abc_pkg/abc_checkerboard.svh:24) [uvm_test_top.env.agent1.checkerboard1]
@2681: <t0> (abc_pkg/abc_checkerboard.svh:46) [uvm_test_top.env.agent1.checkerboard1]
@2686: <t0> (abc_pkg/abc_checkerboard.svh:24) [uvm_test_top.env.agent1.checkerboard2]
@2696: <t0> (abc_pkg/abc_checkerboard.svh:46) [uvm_test_top.env.agent1.checkerboard2]
@2706: <t0> (abc_pkg/abc_checkerboard.svh:24) [uvm_test_top.env.agent1.checkerboard3]
@2716: <t0> (abc_pkg/abc_checkerboard.svh:46) [uvm_test_top.env.agent1.checkerboard3]
...
```

The time, transaction name, file, line number and current scope are all tracked.

A cut-out of the sequence might get instrumented:

```
item = abc_item::type_id::create($sformatf("item%0d", transaction_count++));
`W_NEW(item)

`W_MARK(item)
  start_item(item);

// Randomize ...

`W_MARK(item)
finish_item(item);
`W_MARK(item)
```

The macro interface for tracking transactions is quite simple. When a transaction is created (constructed), then the `W_NEW(t) is used. When a transaction appears at an “interesting” place, then `W_MARK(t) is used. To print the trail of places a transaction has “been”, then use `W_PRINT(t). In the code snippet above, a transaction is created using the create() call. Then `W_NEW(item) is used to “register” the construction. Then the verification engineer has decided that instrumenting before the call to start_item() and before the call to finish_item() is a good idea. `W_MARK(item) is used at those locations. Finally, to signify that the transaction has “finished”, `W_MARK(item) is used after finish_item() returns.

Using `W_NEW(t) and `W_MARK(t) a testbench can be instrumented with a few easy text changed. The code that calls start_item(), finish_item() and create() will need to be updated with `W_MARK() and `W_NEW(). Following transactions is not limited to just these calls. Any place that a transaction is used can be followed. It is as simple as inserting a line `W_MARK(t). This call will register ‘t’ with the analysis database each time it executes.

VIII. WHERE HAVE MY TRANSACTIONS GONE? SPIKED DATA IS GOOD.

Knowing where transactions came from and where they went in the testbench is good, but it is also good to be able to follow them across RTL. For an automated system, some kind of id or tag is necessary that can connect the pin wiggle transactions to the class-based testbench transactions. Some protocols have such tags or ids. Others do not. A general solution to be able to follow cause and effect – and to make sure the data got to the right place is to use “spiked data”. Spiked data is just data you can recognize. It is not randomly generated data, since you won’t be able to look at it at the destination and know right away if it is correct.

Many testbenches are already written using data tagging or data spiking. The most common spiked data is the payload “DEADBEEF”. It is easy to see in memory – you know right away that the data made it across the switching fabric. But you still don’t know which transaction caused it – perhaps there are two writes in flight. Which one won? You can send “DEADBEE1” and “DEADBEE2”. What about 4 writes? Each different situation requires different spiked data.

For an AXI-like protocol transferring beats, the data payload was created as byte, with each byte containing its own byte count number. Then the first 4 bytes were replaced with the ID of the master who sent the burst. Now when the payload arrives at the slave we know if any bytes got rearranged, and we know which master originated the data. A really simple solution to tracking data across a fabric or collection of switches.

```
#  beat #0: 0x060606060405060708090a0b0c0d0e0f
#  beat #1: 0x101112131415161718191a1b1c1d1e1f
#  beat #2: 0x202122232425262728292a2b2c2d2e2f
#  beat #3: 0x303132333435363738393a3b3c3d3e3f
#  beat #4: 0x404142434445464748494a4b4c4d4e4f
#  beat #5: 0x505152535455565758595a5b5c5d5e5f
#  beat #6: 0x606162636465666768696a6b6c6d6e6f
#  beat #7: 0x707172737475767778797a7b7c7d7e7f
#  beat #8: 0x808182838485868788898a8b8c8d8e8f
#  beat #9: 0x90910000000000000000000000000000
```

What about other kinds of “well-known” words? Some of my favorite words using A, B, C, D, E, F, 0 and 1 are B00D1E, BA100, B00B00 and FA1AFE1.

IX. MISCELLANEOUS

There are many other parts of the testbench that you may want to instrument, either with verbose printing or with interfaces to analysis databases. These include:

- Debugging your constraints. Instrument your calls to randomize(). Use post_randomize() and pre_randomize() to provide before and after snapshots.
- Debugging your covergroups. Instrument your calls to sample(). Create a sample() implementation that allows for covergroup debug as needed.
- Debugging your assertions. Instrument assertions. Simple print statements on pass and fail.

X. CONCLUSION

With a little bit of overhead an existing testbench can be instrumented with debug information that will create a new view of the underlying test infrastructure. The load average on an agent – how many transactions are outstanding – has been shown. The total number of sequences and sequence items has been demonstrated. A faster `uvm_info has been demonstrated along with spiked data. But the most useful tool will be the transaction trail that the whence database produces. A lost or wayward transaction can be tracked down quickly.

The solution presented is a small amount of SystemVerilog code with a few simple access routines spanning 5 files and 350 lines of code.

Please contact the authors for the complete source code and example.

XI. References

[1] SystemVerilog LRM, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
[2] UVM 1.1d Reference implementation code, http://www.accellera.org/downloads/standards/uvm/uvm-1.1d.tar.gz
[3] Load average - http://en.wikipedia.org/wiki/Load_%28computing%29

XII. Appendix – The Allocation Database

```
`define S_START_SEQUENCE(seq, sqr)                  \
  begin                                           \
  stats_pkg::follow_sequence(`__FILE__, `__LINE__, "Starting", seq, sqr); \
  seq.start(sqr);                                 \
  stats_pkg::follow_sequence(`__FILE__, `__LINE__, "  Ending", seq, sqr); \
  end

// Should be used from within a sequence as
//   start_item(item).  → `S_START_ITEM(item)
//   finish_item(item). → `S_FINISH_ITEM(item)
`define S_START_ITEM(seq_item)                          \
  begin                                               \
  stats_pkg::follow_sequence_item(`__FILE__, `__LINE__, "start_item  start", seq_item, this); \
  start_item(seq_item);                               \
  stats_pkg::follow_sequence_item(`__FILE__, `__LINE__, "start_item  done ", seq_item, this); \
  end

`define S_FINISH_ITEM(seq_item)                         \
  begin                                               \
  stats_pkg::follow_sequence_item(`__FILE__, `__LINE__, "finish_item start", seq_item, this); \
  finish_item(seq_item);                              \
  stats_pkg::follow_sequence_item(`__FILE__, `__LINE__, "finish_item done ", seq_item, this); \
  end

// Used for Sequence Items, from within a Sequence.
`define S_CREATE_OBJECT(lhs, type_t, typename) \
  lhs = type_t::type_id::create(typename); \
  stats_pkg::created_object(`__FILE__, `__LINE__, typename, lhs, this);
```

XIII. Appendix – The Counted Object Database

```
typedef int count_keyed[string];
count_keyed count_keyed_thing[string] = '{default: '{default:0}};

function void count_keyed_thing_print();
  count_keyed c;
  foreach (count_keyed_thing[name]) begin
    $display("STATS: Counted by %s", name);
    c = count_keyed_thing[name];
    foreach (c[thing])
      $display("STATS:              %5d : %s", c[thing], thing);
  end
endfunction

function void created_object(string filename, int linenumber,
      string typename,
      uvm_sequence_item seq_item,
      uvm_sequence_base seq);
  string key;
  key = $sformatf("%s:%0d", filename, linenumber);
  count_keyed_thing["file_line"][key] += 1;
  key = $sformatf("%s", seq.get_full_name());
  count_keyed_thing["seq_full_name"][key] += 1;
  key = $sformatf("%s", seq.get_type_name());
  count_keyed_thing["seq_type_name"][key] += 1;
  key = $sformatf("%s", seq_item.get_type_name());
  count_keyed_thing["seq_item_type_name"][key] += 1;
endfunction
```

XIV. Appendix – The Sequence types Running Database

```
class counted_items;
  int count;
  string name;
  time t;
endclass

counted_items  sequence_types_running_q[$];
counted_items  sequence_types_running[string];
```

XV. Appendix – The Sequence Item Occupancy Database - loadav

For the sequence_item occupancy, we have a “load average” class created and stored in an associative array indexed by the “resource” being occupied, in this case the ‘agent name’.

```
class f_loadav;
  real count = 5;
  real avenrun[3];
  int active_jobs;

  function void start_job();
    active_jobs++;
  endfunction

  function void end_job();
    if (active_jobs > 0) active_jobs--;
  endfunction

  function string loadav();
    return $sformatf("[%6.2g %6.2g %6.2g]", avenrun[0], avenrun[1], avenrun[2]);
  endfunction

  real exp[3] = '{ 0.92004, 0.98347, 0.99446 };
  real small_number = 0.01;

  function void calc_load(int ticks);
    count -= ticks;
    if (count < 0) begin
      count = 5;
      for(int i = 0; i < 3; i++) begin
        avenrun[i] = active_jobs*(1.0-exp[i]) + avenrun[i] * exp[i];
        if (avenrun[i] < small_number) avenrun[i] = 0.0;
      end

      // $display("STATS: @%0t: calc_load = %s", $time, loadav());
    end
  endfunction
endclass

class STATS;
  f_loadav loads[string];    // Resource: agent or driver.

  function void end_job(string name);
    if (!loads.exists(name)) loads[name] = new();
    loads[name].end_job();
  endfunction

  function void start_job(string name);
    if (!loads.exists(name)) loads[name] = new();
    loads[name].start_job();
  endfunction

  function void loadav();
    $display("STATS: loadav -------------------" );
    foreach (loads[x])
      $display("STATS: %s: loadav %s", x, loads[x].loadav());
  endfunction
  function void calc_load(int ticks);
    foreach (loads[x])
      loads[x].calc_load(ticks);
  endfunction
endclass

STATS stats = new();
```

XVI. Appendix – The Whence Database

```
`define GENERIC

`ifdef GENERIC
`define handle_to_string(c) $sformatf("0x%08x", c)
`else
`define handle_to_string(c) $get_id_from_handle(c)
`endif

`define     W_NEW(class_handle) whence_pkg::whence_db::construct(
    `handle_to_string(class_handle),
    class_handle.get_full_name(), `__FILE__, `__LINE__, $sformatf("%m")                   );

`define    W_MARK(class_handle) whence_pkg::whence_db::mark(
    `handle_to_string(class_handle),
                                  `__FILE__, `__LINE__, $sformatf("%m"), this.get_full_name());

`define W_DESTROY(class_handle) whence_pkg::whence_db::destroy(
    `handle_to_string(class_handle),
                                  `__FILE__, `__LINE__, $sformatf("%m")                   ); \
     class_handle = null;

`define   W_PRINT(class_handle) whence_pkg::whence_db::print(
    `handle_to_string(class_handle)

  typedef struct {
    string file;
    int line;
    string scope;
    string get_full_name;
    time t;
  } place;

  class whence;
    string name;
    place construct_place;
    place destroy_place;
    place q[$];

    ...

    function void mark(string file, int line, string scope, string get_full_name);
      place p;
      p.file = file;
      p.line = line;
      p.scope = scope;
      p.get_full_name = get_full_name;
      p.t = $time;
      q.push_back(p);
    endfunction

    function void construct(string file, int line, string scope);
      construct_place.file  = file;
      construct_place.line  = line;
      construct_place.scope = scope;
      construct_place.t     = $time;
    endfunction

    function void print();
     ...
    endfunction
  endclass
```

```
class whence_db;
  static whence whence_table[string];

  static function whence newget(string c);
    whence_table[c] = new();
    return whence_table[c];
  endfunction

  static function whence get(string c);
    return whence_table[c];
  endfunction

  static function void construct(string c,
      string class_handle_name, string file, int line, string scope);
    whence w;
    newget(c).construct(file, line, scope);
    w = get(c);
    w.name = class_handle_name;
  endfunction

  static function void destroy(string c, string file, int line, string scope);
    get(c).destroy(file, line, scope);
    get(c).q.delete();
    whence_table.delete(c);
  endfunction

  static function void mark(string c,
      string file, int line, string scope, string get_full_name);
    get(c).mark(file, line, scope, get_full_name);
  endfunction

  static function void print(string c);
    get(c).print();
  endfunction

  static function void print_all();
    foreach (whence_table[s])
      print(s);
  endfunction
endclass
```

XVII. APPENDIX – SPIKED WORDS

```
  deadbeef
  ab aba abaca abba abbe abed abele able abled abode aboded accede acceded accolade ace aced ad
add addable added addle addled ado adobe adobo affable al alba albedo ale alec alee alf alfa
alfalfa all allele allocable aloe aloof baa baaed baba babble babbled babe babel baboo bad bade
baffle baffled balada balboa bald baldfaced bale baled ball ballad ballade balled baloo baobab be
bead beaded beadle bed beddable bedded bee beef beefalo beefed befall befell befool bell belle
belled bello blab blabbed blade bladed bleb bled bleed blob blobbed bloc blood blooded bo boa bob
bobbed bobble bobbled bocce bode boded bola bold bolded boldface boldfaced bole boll bolo boo
booboo boodle booed c ca cab cabal cabala caballed cabbala cabbed cable cabled cabob caboodle
cacao cad cade caeca cafe calf call calla callable called ceca cede ceded celeb cell celled cello
clad cladded clade cladode clef cloaca cloacae clod co coal coaled coalface cob cobb cobble
cobbled coble coca coco cocoa cocobolo cod coda codded coddle coddled code codec coded coed
coffee coffle col cola cold cold-blooded coldblooded coo cooed cool cooled da dab dabbed dabble
dabbled dace dad dada dado daedal dal dale de dead deadfall deaf deal deb debacle decadal decade
decaf decal decaled deco decodable decode decoded deed deeded deface defaceable defaced del dele
deled dell do doab doable dob doc doddle dodo doe doff doffed dolce dole doled doll dolled doo
doodad doodle doodled ebb ebbed ecce eco eddo eel effable efface effaceable effaced el elf ell
elodea fab fable fabled facade face faceable faced fad fade faded faff faffed falafel fall
falloff fecal fed fee feeble feed feel fell fella fellable felled feoff feoffed flab flabella
flea fled flee fleece fleeced floe flood floodable flooded foal foaled fobbed focal foe fold
foldable folded food fool fooled la lab label labeled labella labelled lablab lac lace laced lad
lade laded ladle ladled le lea lead leaded leadoff leaf leafed led lee lo load loaded loaf loafed
lob lobbed lobe lobed lobo local locale loco locoed lode loll lolled loo looed loofa oaf obo oboe
od odd oddball ode of off offal offed offload offloaded old ole oleo olla
```

XVIII. SIMPLE SCRIPT TO GENERATE SINGLE-STEP TRACES

```
proc go { n } {
  for {set i 0} { $i < $n } { incr i } {
        step -current;
        set msg [vsim_kernel tb 0]
        puts $msg
  }
}
```

To generate a listing of single step locations (as discussed in the `uvm_info section) using the script above, a simulator vendor must have:

1. A single step command that will step into the next statement and will stay in the same thread.
2. A way to print the current location (file, line number, thread/context)

To create the listing source the script. This creates a new command called ‘go’. Type ‘go 500’ to single step 500 times. Examine the listing generated in the transcript.

Normal usage:

- Set a breakpoint at the place to begin tracing.
- Run until that breakpoint.
- go 500