Determining Test Quality through Dynamic Runtime Monitoring of SystemVerilog Assertions

Kelly D. Larson
Nvidia Corp.
Austin, TX
klarson@nvidia.com

***Abstract*—At times it's hard to know that a test is doing what it's supposed to do. As long as nothing 'bad' happens which causes a design failure, a test which no longer achieves its intended purpose can easily slip under the radar, consuming valuable resources and providing a false, and sometimes dangerous, sense of security. This paper describes how to use the SystemVerilog assertion API in conjunction with a UVM testbench to dynamically 'instruct' the testbench what the intended behavior of a particular test is at runtime. By using this technique, one is able to tell immediately if the test is 'broken,' as it will now report a failure in much the same manner as when the design is 'broken.' Using these techniques will help minimize wasted simulation cycles caused by running broken tests, and help mitigate risks by eliminating unwanted coverage holes.**

Keywords—verification; assertions; SVA; UVM; ABV; SystemVerilog; coverage

I. INTRODUCTION

The SystemVerilog Assertion language (SVA) has proven to be an effective tool to verify correct design behavior. The SVA syntax features a very concise way to describe expected behavior at a low level. SystemVerilog assertions will cause an immediate failure if the design violates the specified behavior, and they can be efficient to debug because the reported failure is typically very close to the error.

This paper describes an entirely different way to use these same SVA assertions during simulation. While the more typical use of SystemVerilog assertions is often targeted towards DESIGN QUALITY, this paper describes how to effectively use assertions to target individual TEST QUALITY.

Test quality is an important concept in the verification of modern SoC (System-On-a-Chip) designs, as it helps insure that limited verification resources are being directed towards high-value activities. Besides wasting simulation cycles, tests which fail to completely test what they are supposed to might also introduce unwanted coverage holes, and increase the likelihood of bugs making it to silicon. While many tests are written to be self-checking in terms of detecting when a design failure occurs, many times these tests are not written to be self-checking in terms of whether or not they successfully create the condition that they are trying to test. Tests which work well initially may, during the course of the project, no longer reach the condition that they were trying to test by the end of the project, or when ported to the next project. Because of the complexity of many designs, the sheer number of tests involved, and because each individual test might target completely different features, efficiently determining whether or not each test is performing correctly can be a difficult task to accomplish.

Luckily, the SystemVerilog language itself provides facilities which allow us to dynamically monitor test behavior by providing an assertion API [1]. When used properly, this API can allow us to employ the powerful SVA language to help track the behavior of individual tests. In many cases the same SystemVerilog assertions which were written for measuring design quality can also be used to measure test quality, but it's important to realize that the fundamental goal is quite different.

This paper will describe how to write such an assertion monitor, and tie it into a UVM verification environment. An example implementation will be detailed, along with selected code examples.

II. TARGETTING TEST QUALITY

The key to using assertions to target test quality is to enhance the runtime testbench environment by providing the ability to tie the pass/fail condition of an individual test to whether or not one or more specific assertions were actually checked, or perhaps whether one or more specific coverpoints were hit during the course of the test. This in itself actually has nothing to do with the correct behavior of the design itself. If a design assertion fails, we are still expecting that the simulation will also fail. What we are interested in here, however, is whether or not the specific assertion itself was ever actively checked during the test. This is similar to analyzing assertion coverage reports, except the focus is on an individual test, and done while the simulation is running. This helps answer the important question of 'Is this test actually doing what I want it to do?'

This can be helpful for situations such as:

- My test was supposed to hit a specific coverage point or fail. Did it?
- I know my test was supposed to make condition 'X' happen exactly five times or fail. Did it?

- Because of the way that I wrote my test, I should never see 'Y' happen, and if it does I want the test to fail even though 'Y' itself is not illegal. Will it?

By doing this dynamically during runtime, we are able to immediately flag an error during a simulation, and fail the simulation through ordinary means, such as the standard UVM (Universal Verification Methodology) testbench failure mechanism [2]. UVM also provides facilities for parsing command line options for dynamic configuration of the assertion monitoring routines, as demonstrated later in this paper.

In addition to being useful for specifying the runtime characteristics of directed-style tests, this technique can also be used to provide dynamic feedback to constrained random sequences during the simulation to indicate completion of a goal, or to guide the generation of new stimulus.

III. Example Use Case

To help illustrate how one would use runtime assertion monitoring, let's look at a simple example. For this example, let's assume we are testing an arbiter block as shown in Figure 1. This arbiter has three request inputs for high, medium and low priorities, and three corresponding grant outputs.

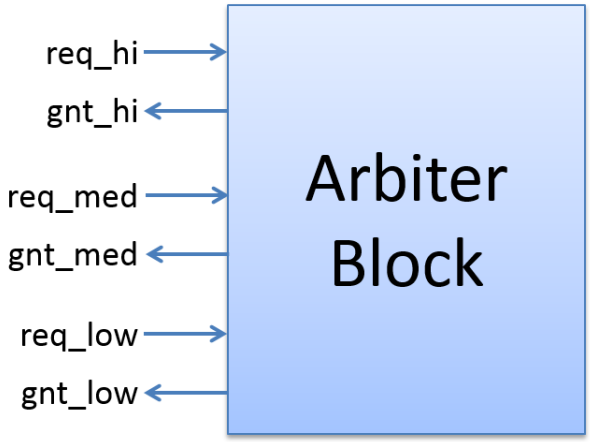

Figure 1 – Example Arbiter Block Diagram

To help verify this block, the verification engineer might also write some assertions to describe the desired behavior, as shown in Figure 2. (Please note that even for this very simple example, this set of assertions alone is not sufficient to completely describe the desired behavior.)

```
check_hi: assert property(@(posedge clk) disable iff (reset)
                    req_hi |=> gnt_hi);

check_med: assert property(@(posedge clk) disable iff (reset)
                    (req_med & !req_hi) |=> gnt_med);

check_low: assert property(@(posedge clk) disable iff (reset)
                    (req_low & !req_med & !req_hi) |=> gnt_low);
```

Figure 2 – Example SVA syntax

These assertions are helpful in that they will fail if illegal activity is observed, but they are not very helpful in determining whether or not the arbiter was tested correctly. If you had written a test that was supposed to generate simultaneous requests on all of the request lines, these assertions will not help you determine if that case was actually tested. To insure proper testing in this example, you might need to write a cover point, such as shown in Figure 3.

```
check_arb: cover property(@(posedge clk) disable iff (reset)
                    (req_hi & req_med & req_low));
```

Figure 3 – Example cover point syntax

With this cover property, you will now be able to see in a coverage report if this case was ever hit. This is helpful, but many times coverage is generated over an entire regression suite. While this will at least tell you that you did indeed hit the desired test condition, it may not be immediately apparent if the individual test you wrote to hit the condition was successful in hitting it or not. If the test has been broken for some reason, you might be wasting simulation cycles running a useless test without realizing it. This test itself might also be combining this coverage point with some other unique conditions, and in reality you now have a coverage hole that could go unnoticed.

What would be nice would be if every time we ran the test, we could add something like a simple plusarg to the simulation command line which would tell the simulation environment to fail the test if the required cover point was not hit. This capability would then immediately flag a test as 'broken' if for some reason it quit functioning correctly. This plusarg might look something like Figure 4.

```
+RequireAssert=check_arb
```

Figure 4 – Proposed syntax for requiring an assert/cover

This runtime monitoring capability could also be extended well beyond the capabilities of a coverage report. What if you wrote a test which, because of the way you wrote it, you know you should never get a high priority request. This same runtime monitoring capability could then be used to indicate which cover points should not be hit during simulation, as seen in Figure 5.

```
+ProhibitAssert=check_hi
```

Figure 5 – Proposed syntax for prohibiting an assert/cover

In this case the test would fail if the assertion 'check_hi' passed. Of course 'check_hi' passing is not illegal from the RTL design point of view, we have simply made it illegal from the point of view of this particular test.

IV. How is this done?

Now that we've explored some of the rationale behind monitoring assertions, the rest of this paper will describe how a runtime assertion monitor can be written. In order to accomplish all of this, the technique described in the paper makes use of both the SystemVerilog Assertion API, along with a UVM testbench framework.

The SystemVerilog language specification includes an Assertion API which provides a rich set of routines and access functions that allow us to dynamically interact with the assertions and coverpoints within the RTL at runtime. We will make use of two key features of this API:

1. The ability to iterate through a design to find specific assertions.
2. The ability to attach our own callback (subroutine) to an assertion which will get called whenever the assertion (or cover point) passes successfully.

V. Assertion Monitor

To create the assertion monitor we'll need two basic pieces. One piece is coded in SystemVerilog as part of the UVM testbench, and the other piece is a collection of C routines which will track the assertion activity during the simulation through the SystemVerilog assertion API.

A. UVM Portion of Assertion Monitor

The UVM portion of the assertion monitor consists of a class object, which is written in SystemVerilog and extended from a UVM component object. In UVM, all components are aware of the 'phase' of the test being run, or in other words they are aware of when the test is initializing, when the test is progressing, and when the test has finished. This is important to us, as our assertion monitor has various activities which it needs to perform before, during, and after the test. The primary responsibilities of this UVM class are:

- Before the main test phase begins, parse any command line directives which specify specific behavior for the simulation run, and call the appropriate assertion monitor DPI routines to instrument the required behavior.
- At the end of the test, do any final checks which are required to see that observed assertion and cover point behavior was within the specified parameters.
- Provide several utility functions that will allow the assertion monitor DPI routines to register warnings and errors through the UVM logging facilities.

B. C DPI Portion of Assertion Monitor

The DPI portion of the assertion monitor is a set of routines written in C. The reason these routines are written in C is that it allows access to key information about the simulation at runtime through programming API's, which are part of the SystemVerilog standard. The primary responsibilities of this set of C routines are:

- Provide a data structure to store runtime information about the assertions and cover points that we have chosen to monitor.
- Provide the mechanism to attach a callback routine to any assertion or cover point that we have chosen to monitor.
- Provide the callback routine which will be run every time a monitored assertion or cover point successfully passes. This routine will track assertion behavior, and generate a UVM error if the assertion behavior is outside of the limits that we have specified for the test.
- Provide an 'end of test' routine which will do a final check of the behavior of all monitored assertions and cover points, and generate a UVM error if any assertion behavior is outside of specified limits.

C. Plusarg Interface of Assertion Monitor

To utilize the assertion monitor, we need to be able to associate specific runtime behavior to a specific test. One way to do this is through the use of 'plusargs' on the simulation commandline. The advantage of this approach is that the set of assertions which are being monitored, and their parameters, are completely dynamic, and do not require recompilation of the simulation when the desired behavior changes. This can be extremely useful in testbenches such as those which contain an embedded processor, and whose tests consist of compiled code which is loaded into memory at the beginning of the simulation. In this type of environment, many different tests can be developed and run without recompilation of the model. For each test which is run, a different set of plusargs can be used to describe the required behavior of assertions and cover points in order for the test to pass.

This paper will describe one possible approach to defining a set of 'plusargs' to accomplish our goals, though the exact syntax used here is arbitrary. For the assertion monitor described in this paper, we will use two different plusargs, +RequireAssert and +ProhibitAssert. These plusargs are used to indicate which assertions to monitor, along with additional arguments to further specify desired boundaries and ranges.

Type	# Args	Example & Description	When Checked
Require	0	+RequireAssert=myassert. Assertion must fire at least once during the test.	End of test
	1	+RequireAssert=myassert:x, Assertion must fire at least 'x' times during the test.	End of test
	2	+RequireAssert=myassert:x:y, Assertion must fire in the range greater than or equal to 'x', and less than or equal to 'y' times.	During (too many), End (too few)
Prohibit	0	+ProhibitAssert=myassert, Assertion must never fire during the simulation.	During Test
	1	+ProhibitAssert=myassert:x, Assertion must not fire 'x' or more times (less is OK).	During Test
	2	+ProhibitAssert=myassert:x:y, Assertions cannot fire in the range of [x:y] inclusive (less or more is OK).	End of test

Figure 6 – Assertion monitor plusarg interface

Figure 6 summarizes the plusarg usage with the assertion monitor, along with the meanings of the various arguments. Where possible, the specified assertion behavior will be checked during the simulation with failing conditions reported immediately. Some checks, however, must be held until the end of the simulation to determine whether or not they are within the established bounds, as indicated in the table.

Other points to note:

- The assertion specification (shown in the table as *myassert*) does not need to be a full hierarchical path to the assertion, and for robustness should only contain enough of the path to insure its uniqueness. The assertion monitor is able to detect and warn if the assertion specification is not unique.
- Multiple plusargs can be used, and multiple comma-delimited assertions can be specified with a single plusarg.

D. *Procedural Interface of Assertion Monitor*

In addition to the plusarg interface, the assertion monitor should provide the same functionality through a procedural interface as well. This type of interface is not as flexible, as it is determined at compile time rather than runtime, but it allows access to assertion monitoring by things such as constrained random tests and sequences. (See Figure 16 for an example.)

VI. UVM/Verification Monitor Test flow

Figure 7 shows a high-level overview of the interaction of the both the SystemVerilog and C DPI portions of the assertion monitor with the RTL design during the different phases of the simulation.

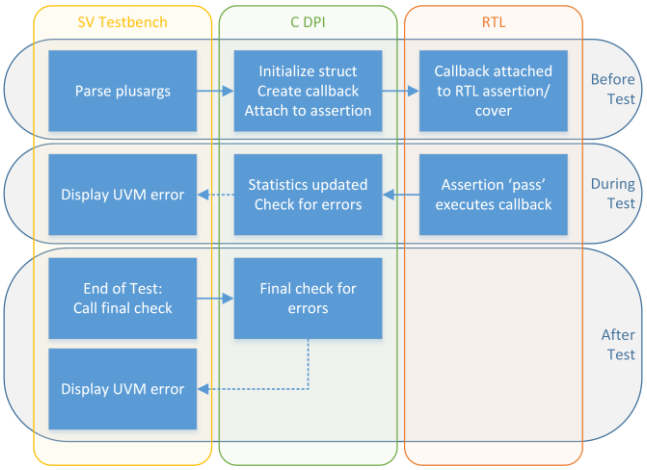

Figure 7 – High-level verification monitor test flow

At the beginning of the simulation, before time starts to advance and the actual test begins, the SV testbench parses the commandline plusargs to see what the expected behavior is for any assertions or coverpoints which the verification engineer would like to monitor during the test. If there are any requests to monitor, the appropriate C DPI routine is called to attach a callback to the assertion or coverpoint.

Once the test begins, every time the assertion passes non-vacuously, or every time the coverpoint condition is reached, this callback is executed, and the C DPI portion of the assertion monitor will update its tracking information. If the behavior is outside of the required range, the SV testbench is called to produce a UVM simulation error at the time of failure.

At the end of the test, the SV testbench will call one final end-of-test routine in the C DPI portion of the assertion monitor to check for any remaining errors.

VII. Detailed Code Description

This section will examine the assertion monitor source code in more detail. A complete code listing is not included here for the sake of brevity, however the most crucial sections will be discussed. First the SystemVerilog portion will be examined, followed by the C DPI portion.

A. *UVM Testbench Object Code*

The UVM portion of the assertion monitor is a UVM component with two primary functions, handling plusargs, and cleaning up at the end. First, at the beginning of the test it will parse any applicable plusargs. Figure 8 shows the assertion monitor calling the plusarg parsing routines during the end_of_elaboration phase, though any phase occurring before the run phase, where the test actually starts, would work fine.

```
// Called automatically during the end_of_elaboration
// phase. Parses any runtime requests for assertion monitoring
// via the plusargs.
function void assert_mon::end_of_elaboration_phase(uvm_phase phase);

   // Parse plusargs for runtime assertion monitoring requests
   // and parameters
   parse_assertion_arg("+ProhibitAssert=", PROHIBIT);
   parse_assertion_arg("+RequireAssert=", REQUIRED);

endfunction: end_of_elaboration_phase
```

Figure 8 – UVM end of elaboration phase routine

Figure 9 shows the routine to parse the plusargs and extract the assertion name and optional arguments. Here we make use of the UVM commandline processor to help us out. Once it has extracted the information for each assertion, it makes a call to 'register_assert'. This is of our DPI functions, described later, which will do the actual work of attaching a callback to the specified assertion.

```
// Routine to parse plusargs and extract assertion name and
// arguments.
// * Multiple plusargs of the same type can be used.
// * Multiple assertions can be used with the same plusarg by
//    using a ',' delimiter.
// * It is legal to have zero, one or two numerical parameters
//    with each assertion.
// * Numerical parameters are delimited by a ':' character.
function void assert_mon::parse_assertion_arg(string arg_string,
                                              ast_type_e ast_type);
   int    num_values;
   int    assert_param1;
   int    assert_param2;
   string arg_values[$];
   string multi_values[$];
   string param_values[$];

   num_values =
      uvm_cmdline_proc.get_arg_values(arg_string,arg_values);
   foreach(arg_values[i]) begin
      // Separate multiple comma delimited assertions
      uvm_split_string(arg_values[i],",",multi_values);
      foreach(multi_values[j]) begin
         // Now separate out any colon delimited assertion parameters
         uvm_split_string(multi_values[j],":",param_values);
         if (param_values.size() > 3) begin
            // Can't have more than two numerical parameters
            `uvm_warning("ASSERT/BADPARM",
                  $psprintf("Bad assertion plusarg: '%s'",
                  multi_values[j]));
         end
         else begin
            // Set parameters (Default -1), then register assertion
            assert_param2 = (param_values.size() == 3)?
                        param_values[2].atoi():-1;
            assert_param1 = (param_values.size() >= 2)?
                        param_values[1].atoi():-1;
            register_assert(param_values[0],assert_param1,
                        assert_param2,ast_type);
         end
      end
   end

endfunction: parse_assertion_arg
```

Figure 9 – UVM plusarg parsing

The same register_assert() function which is being called here can also be called directly from a UVM test or sequence as a procedural way to monitor assertions, without going through the plusarg interface. An example of this is shown later in this paper.

```
// Runs automatically during the report phase. Does the end of
// test checking for specified trigger limits, generating errors
// if any check fails.
function void assert_mon::report_phase(uvm_phase phase);
   super.report_phase(phase);

   assert_end_of_test();

endfunction: report_phase
```

Figure 10 – UVM Report phase routine

Finally, at the end of the test we call another DPI routine, assert_end_of_test() (Figure 10), which does a final check of the collected assertion statistics, reporting any errors as appropriate.

B. C DPI Code

The DPI portion of the assertion monitor is where the bulk of the real work is done. Here we have the responsibility to generate and maintain a data structure that contains information about the activity of all assertions that we are actively tracking.

While the specific data structure used could be implemented in a variety of ways, for the assertion monitor described here we will use a simple linked list of struct variables, as shown in Figure 11.

```
// Structure to store assertion monitoring information
struct mon_assert_s {
  char* name;              // Full pathname of assertion
  int handle;              // Unique assertion handle
  int required;            // Monitor Type: Req=1, Prohib=0
  int monarg1;             // Monitor limit argument 1
  int monarg2;             // Monitor limit argument 2
  int trigger_cnt;         // How many times assertion fired

  struct mon_assert_s* next;   // Pointer to next assertion
};
typedef struct mon_assert_s mon_assert;
```

Figure 11 – C DPI assertion tracking struct

One of these structures will be created for each assertion that we are actively monitoring. Figure 12 shows the code for the register_assert() routine, which is the routine called by the testbench when an assertion is to be monitored.

```
// register_assert()
//   Will attempt to find an assertion at the given path, create an
// assertion monitor for it, and register it with a callback.
int register_assert(char* assert_path, int assert_parm1,
                    int assert_parm2, int required) {
  vpiHandle       matched;
  int             matches;
  int             succeeded = 0;
  mon_assert*     new_assert;

  // Scan through existing assertions to find a unique name match
  matches = scan_for_matching_assertion(assert_path, &matched);

  // Did we get a unique match?
  if (matches == 1) {
    // Found the requested assertion, register the structure with an
    // assertion 'success' type callback.  Every time this assertion
    // 'fires', the callback is called with the assertion monitor
    // structure pointer passed to it as 'user data'
    new_assert =
      init_new_mon_assert(vpi_get_str(vpiFullName,matched),
                  assert_parm1, assert_parm2, required);
    add_assert_cb(matched,cbAssertionSuccess,new_assert);
    succeeded = new_assert->handle;
  }

  return(succeeded);
}

// add_assert_cb()
//   Does VPI call to register a callback with an assertion
int add_assert_cb(vpiHandle cb_assert, int cb_type, mon_assert* assmon)
{
  if (vpi_register_assertion_cb(cb_assert, cb_type, assertCBRtn,
                      (PLI_BYTE8*) assmon) == NULL) {
    vpi_printf("Failed to add %d on %s\n",
            cb_type, vpi_get_str(vpiFullName, cb_assert));
  }
}
```

Figure 12 – C DPI assertion registration

In this routine the design is first scanned for the assertion which we want to monitor. If the assertion is found, init_new_mon_assert() is called. This routine, not shown here, simply creates and initializes a new mon_assert_s struct object.

The most important part of our assertion monitor then follows, adding a callback routine through the vpi_register_assertion_cb() function. This not only allows us to add a callback routine to the assertion which gets called every single time the assertion successfully passes, but it also allows us to attach a pointer to the struct object that we've just created. This is extremely useful, since this pointer is passed back to the callback itself, so when the callback gets called we already have the pointer to the corresponding data structure where we are keeping our tracking information. This saves us from having to search the data structure ourselves every time the callback gets called.

```
// scan_for_matching_assertion()
//   Will attempt to find an assertion at the given path. Will cause
// an error if nothing matches, or if more than one assertion matches.
int scan_for_matching_assertion(char* assert_path,
                                vpiHandle* vhandle) {
  vpiHandle a,b;
  int matches = 0;

  // Iterate through all design assertions & cover points
  a = vpi_iterate(vpiAssertion, NULL);
  while (b = vpi_scan(a)) {
    if (name_match(vpi_get_str(vpiFullName,b),assert_path)) {
      matches++;
      *vhandle = b;
    }
  }

  if (matches > 1) {
    // Found more than one match
    sprintf(nae_buffer,"More than one assertion matches '%s', you must
specify more of the pathname.", assert_path);
    display_error(nae_buffer);
  } else if (matches == 0) {
    // Didn't find any matches
    sprintf(nae_buffer,"No match for specified assertion: %s",
            assert_path);
    display_error(nae_buffer);
  }
  return(matches);
}
```

Figure 13 – C DPI assertion search routine

Figure 13 shows the code for our scan_for_matching_assertion() routine that register_assert() uses to see if we can find an assertion with the specified name. This routine makes use of iterators which are part of the SystemVerilog API, and which make it relatively easy to search through all of the assertions and cover points which are contained within the design.

This routine also checks to see if there are multiple name matches for the specified assertions, which usually means that more of the hierarchical path needs to be specified to uniquely identify which assertion you are trying to monitor.

The display_error() function implementation is not shown, but it simply calls a DPI routine in our UVM testbench which uses the standard UVM reporting mechanism to register an error with the provided description string.

The callback routine that we register on monitored assertions, assertCBRtn(), is also interesting to look at (see Figure 14).

```
// assertCBRtn()
//   Callback routine which is registered with all monitored
// assertions. This routine will automatically be called whenever the
// requested event is observed (i.e.  cbAssertionSuccess). When this
// routine is called, it will have a pointer to the specific assertion
// monitor struct object in the user_data field.
//   Checks are made during this routine to insure that trigger
// behavior falls within the specified ranges for this test. Other
// checks are done at the end of the test within the
// assert_end_of_test() routine.
static PLI_INT32 assertCBRtn(PLI_INT32          reason,
                         p_vpi_time         ct,
                         vpiHandle          assert,
                         p_vpi_attempt_info info,
                         PLI_BYTE8*         user_data
                         ) {
  int i;
  mon_assert* assmon = (mon_assert*) user_data;

  if (reason != cbAssertionSuccess) display_error("Invalid CB Reason");

  assmon->trigger_cnt++;

  // RequireAssert Handling
  if (assmon->required) {
    // If both parms are set, we have a specific limit
    if ((assmon->monarg1 != -1) &&
        (assmon->monarg2 != -1) &&
        (assmon->trigger_cnt > assmon->monarg2)) {
      sprintf(nae_buffer,
              "Required Assert \'%s\' fired %0d times, exceeds [%0d:%0d]",
              assmon->name,assmon->trigger_cnt,
              assmon->monarg1,assmon->monarg2);
      display_error(nae_buffer);
    }
  // ProhibitAssert Handling
  } else {
    if (assmon->monarg1 == -1) {
      sprintf(nae_buffer,"Prohibited Assert \'%s\' fired %0d times",
              assmon->name,assmon->trigger_cnt);
      display_error(nae_buffer);
    } else if ((assmon->monarg2 == -1) &&
               (assmon->trigger_cnt >= assmon->monarg1)) {
      sprintf(nae_buffer,
              "Prohibited Assert \'%s\' fired %0d times, exceeds %0d",
              assmon->name,assmon->trigger_cnt,assmon->monarg1);
      display_error(nae_buffer);
    }
  }
  return 0;
}
```

Figure 14 – C DPI assertion callback routine

This routine will be called every time the assertion or cover point that it is attached to successfully passes. The pointer to the corresponding mon_assert object is passed in through the user_data parameter, so we are saved the trouble of having to search the linked list for the correct entry ourselves.

The callback routine now updates its information, and generates any runtime errors if any parameters are seen to be outside of the specified behavior.

The last piece of the assertion monitor is the final checking routine called at the end of the test (Figure 15).

```
// assert_end_of_test()
//   Should be called by the Assert SV object at the end of a test. At
// this time the list of all assertion monitors is scanned, and checks
// are made to insure that the assertion triggers fell within the
// specified limits for the test. Anything that that does not will
// cause an error at the end of the test.
//   Other checks are done while the test is progressing within the
// assertCBRtn() callback routine.
int assert_end_of_test() {
  mon_assert* cur_assert = assert_head;

  while (cur_assert) {
    // RequireAssert Handling
    if (cur_assert->required) {
      if ((cur_assert->monarg1 == -1) &&
          (!cur_assert->trigger_cnt)) {
        sprintf(nae_buffer,"Required Assert \'%s\' did not fire",
                cur_assert->name);
        display_error(nae_buffer);
      } else if ((cur_assert->monarg1 != -1) &&
                 (cur_assert->monarg2 == -1) &&
                 (cur_assert->trigger_cnt < cur_assert->monarg1)) {
        sprintf(nae_buffer,
                "Required Assert \'%s\' only fired %0d times, it did not meet
specified limit of %0d",
                cur_assert->name,cur_assert->trigger_cnt,
                cur_assert->monarg1);
        display_error(nae_buffer);
      } else if ((cur_assert->monarg1 != -1) &&
                 (cur_assert->monarg2 != -1) &&
                 (cur_assert->trigger_cnt < cur_assert->monarg1)) {
        sprintf(nae_buffer,
                "Required Assert \'%s\' only fired %0d times, it did not meet
specified range of [%0d:%0d]",
                cur_assert->name,cur_assert->trigger_cnt,
                cur_assert->monarg1,cur_assert->monarg2);
        display_error(nae_buffer);
      }

    // ProhibitAssert Handling
    } else {
      if ((cur_assert->monarg1 != -1) &&
          (cur_assert->monarg2 != -1) &&
          (cur_assert->trigger_cnt >= cur_assert->monarg1) &&
          (cur_assert->trigger_cnt <= cur_assert->monarg2)) {
        sprintf(nae_buffer,
                "Prohibited Assert \'%s\' only fired %0d times, within prohibited
range of [%0d:%0d]",
                cur_assert->name,cur_assert->trigger_cnt,
                cur_assert->monarg1,cur_assert->monarg2);
        display_error(nae_buffer);
      }
    }
    cur_assert = cur_assert->next;
  }
}
```

Figure 15 – C DPI end of test routine

In this routine we scan through all of the collected assertion behavior, and check that everything is within the designated boundaries.

VIII. Assertion Monitoring with Random Stimulus

In addition to providing value to more directed style testing, the type of active assertion monitoring described in this paper can also be used to provide dynamic feedback to help guide constrained random approaches as well. Figure 16 shows an simple example of this type of use.

```
// Sequence will attach a monitor to design overflow detection
// cover point, then randomize continuous transactions until the
// overflow condition is hit three times.
class bus_seq extends uvm_sequence #(bus_txn);
   bus_txn   tr;
   int       ahandle;
   int       successes;

   `uvm_object_utils(bus_seq);

   virtual task body();

      // Instrument cover point for tracking
      ahandle = register_assert("upper.overflow_detect",-1,-1,1);

      // Keep looping until cover point has been triggered 3 times
      while (successes < 3) begin
         `uvm_do(tr);
         successes = num_assert_successes(ahandle);
      end
   endtask: body

endclass: bus_seq
```

Figure 16 – Using assertion monitoring with a UVM sequence

Here from within a UVM sequence body, we are registering an assertion (or cover point) to be actively monitored by our assertion monitor. We then continuously generate a random transaction stream until the assertion is hit at least three times, at which time we exit the sequence. num_assert_successes() is a DPI routine (not shown) which simply searches our assertion monitor data structure for the matching handle and returns the number of times that assertion has fired.

While this is a fairly simple example, this approach could be extended to include much more complicated feedback that could guide the stimulus in interesting ways.

IX. Specific Tool Caveats

All of the code shown in this paper is based upon the IEEE SystemVerilog standard, and has been tested to work on three major simulators, with the following caveats:

A. *Synopsys*

An issue existed in the VCS tool from Synopsys prior to version 2012.09-SP1-1 which prevented accurate monitoring of assertion passes using the Assertion API. Newer versions have addressed this issue, however the "-assert cbSuccessOnlyNonVacuous" flag must be used at runtime. At the time of this writing, Synopsys' plan is to make this behavior default in the 2014.03 release, after which the flag would no longer be required.

B. *Cadence*

On the Cadence Incisive simulator you need to use the "-abvrecordcoverall" option to insure accurate monitoring results. If this switch is not used, callbacks are only triggered a single time for each assertion during the simulation for all passing conditions as a way of optimization.

C. *Mentor*

There are no known issues or additional options required for the Mentor simulator.

X. Summary

With today's increasingly complex designs and ever tightening design schedules, it's more important than ever to make sure that every simulation cycle is well spent. Being able to dynamically monitor test quality by insuring that each test continues to function properly throughout the course of the project is a good way to help prevent wasted cycles and unwanted coverage holes by running broken tests.

The SystemVerilog language has built-in facilities that allow us to dynamically track which assertions and coverpoints have been hit during the course of a specific test. Using the techniques outlined in this paper can go a long ways towards helping to insure the quality of each test, maximize simulation resources, and achieve the desired coverage goals throughout the course of a project.

References

[1] "1800-2012 IEEE Standard for SystemVerilog: Unified Hardware Design, Specification, and Verification Language," IEEE, New York, NY, 2013, Chapter 39.

[2] "Universal Verification Methodology (UVM) 1.1 User's Guide", Accellera, 2011