Applying Test-Driven Development Methods to Design Verification Software in UVM-*e*

Doug Gibson (doug.gibson@hp.com) - Hewlett Packard Company

Mike Kontz (michael.kontz@hp.com) - Hewlett Packard Company

ABSTRACT

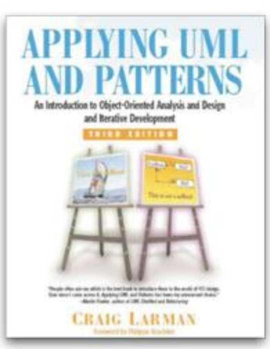
Hewlett Packard's Systems VLSI Lab has been using Test Driven Development (TDD) best–practices to develop software in their custom SystemC-based verification environment for several years. Software component quality and development times have both improved with the addition of this framework. During our recent adoption of the UVM-e methodology, we wanted to maintain this TDD capability. The new UVM-e unit-test framework enables us to continue this valuable development process.

Keywords—TDD; unit-test; UVM-e; eUnit

INTRODUCTION

Origins

HP's Enterprise Server division has been developing large scale ASICs to support the Integrity Superdome and ProLiant DL980 servers for many years. The verification methodology for these programs was originally developed in 2004 and was based on SystemC and the SystemC Verification extension libraries. This testbench became a large C++-based programming effort, being developed mainly by electrical and computer engineers. Software design methodologies were noticeably lacking.



During a transition between programs in the summer of 2010, many members of the team enrolled in "Applying UML and Patterns: Hands-on Mastery of OOAD, Agile Modeling, and Patterns with TDD [1]", a class designed to teach newer Agile programming disciplines. The team recognized that some of the methodologies presented in the class could be applied in ways unique to the problem of verification software development.

Goals and Solution

Our organization invests a lot of effort developing scoreboards and checkers to verify the correctness of the chips we develop. On most designs, these components are staffed early so that they can meet the demands of the design team as the RTL is turned on. Unfortunately, this often means that the DV engineer starts developing code before they can properly stimulate it to ascertain that the component is really operating properly. Furthermore, as the RTL and checker are both turned on together, the feature testing is usually geared towards the RTL without as much emphasis on validation of the DV component. Any focus on the DV component quality is derived from fixing bugs exposed during **RTL** turn-on. The pseudo-random testing methodology relies heavily on the checker/scoreboard catching errors, but often the software code behind the flagging of the errors has never been explicitly tested!

Test-driven development can help solve this problem of DV component quality and schedule. It allows the DV software developer to build a test jig for the software before the RTL is available. Various stimulus scenarios can be applied to the component to validate correctness, both for the RTL "correct" path, as well as RTL "incorrect" paths. This gives the component developer the ability to get ahead of the RTL development, providing a much higher quality component at initial RTL turn-on. It also validates the efficacy of the checks being applied to the RTL. Finally, the DV software developer will have a set of qualification tests available to ensure that changes made later in the project don't inadvertently break parts of the checking strategy.

CPPUnit Solution

Our team's original foray into TDD was on a C++-based testbench. We developed a unit-test capability using CPPUnit, a popular open-source Test-Driven Development (TDD) framework. Classical TDD specifies that the developer identify a feature to be developed and first write a unit-test that will verify that the feature meets particular specifications. Initially the tests will fail, as the feature does not meet the test specification. Then the developer codes the feature until all of the tests pass, at which point the feature is complete and development stops. Our testbench software is not the primary product, so our team has not been quite so dogmatic about following the TDD methodology. Instead, we've chosen to apply the capabilities inherent in a unit-testing methodology to enhance the quality and productivity of our software development process.

Unit tests are organized as a series of individual tests, each housed within a testsuite (see Figure 1). The testsuite is a specialization of the test framework's base class and enables instantiation of the object to be tested. Each testcase within the testsuite will cause a new, unique object under test to be created. The testcase will be run against that object, producing a pass-fail result, after which the object under test is destroyed. In this way, each testcase is isolated from all others and execution order of the tests is irrelevant. The unit test framework provides assertion capabilities so that the test writer can code result expectations and trap program exceptions without disturbing any other tests during the run. Typically, this type of framework can run many simple tests in a few seconds, although long tests are not unacceptable. The goal is to provide quick feedback to the software developer confirming the quality of their code.

UNIT TEST FRAMEWORK IN UVM-E

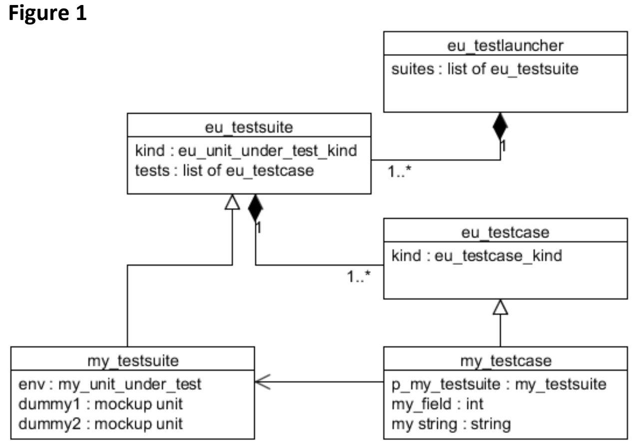
The e language is object-oriented and thus should be able to support a unit-test framework similar to CPPUnit. We initially looked at developing a framework, but ran into a fundamental Specman design constraint. In Specman-e, all units are constructed at time zero when the testbench is generated. This is at odds with the unit-test concept where the object to be tested is constructed at the beginning of a test and destroyed at the end. Tests cannot be truly independent if the unit they are testing is preserved across testcases. Further, a testsuite trying to iterate through numerous configurations of the same unit would run into name collisions and wasteful memory usage.

We approached Cadence looking for a solution, "Please give us the ability to construct and destroy units at will when we're running unit tests." They responded that they were interested in developing an entire unit-test framework, and the eUnit Testing Framework was born.

Testsuite and Testcases

The eUnit framework mimics the structure found in many industry unit-test frameworks, with a set of

Figure 1



eu_testsuites, each of which homes one or more testcases, as shown in Figure 1. A key attribute of the framework is that it can generate and run all tests without the need to reload the code. The test throughput is much more acceptable in this mode, while the isolation of each test from all others is preserved. In addition, the test runner implements a watchdog timer so that tests that hang will be aborted, allowing following tests to run.

Mock Objects and Subtypes

Many verification components are built to operate in the context of a complete testbench and not in isolation. As will be shown in later examples, the ease of unit testing an object has a lot to do with how its ability to be instantiated and run in isolation. To enable unit testing of objects with external dependencies, some modifications to their implementation may be needed. The natural way to do this in e is with ***when*** subtypes. The objects to be created in the unit test are given a new MOCK subtype with which they can add to or modify the existing object's implementation to facilitate unit test. This could be something as simple as a checker not probing an internal RTL path if in unit test/MOCK mode. The objects instantiated in the testsuite and/or testcase are often referred to as "mock" objects. One or more of the mock objects will be the unit under test. Other mock objects may be needed to fulfill requirements of the unit under test. A checker may need a register model object for instance.

Unit Test Assertions

Different types of assertions are provided by the unit test framework. In the case of UVM-e, the assertions target verification operations like error signalling with **dut_error** and emitting events. As will be seen in examples, these assertion statements are strategically placed in the unit test testcase to implement the desired checking.

Assertions in eUnit:

- **expect <expr> to be <expr>** - Verify that the two expressions are equal at this point in the test. The <expr> fields can be any value returning expression such as a value returning method call, an internal state variable or a constant value.
- **expect_dut_error <str>** - Verify a dut_error containing the error message <str> has been thrown by this point in the test.
- **expect_no_more_dut_errors** – Verify that no unmatched dut_errors have occurred up to this point in the test. A matched dut_error is one that has been captured by expect_dut_error.
- **wait_for_expected_event <event> <within cycles>** - Verify the <event> has been emitted <within_cycles> from this point in the test.

Terminology

Here is a summary of unit-test and eUnit terminology before covering use cases and example code.

Unit Under Test – Similar to a Device Under Test (DUT) in an RTL testbench, the unit test framework has a Unit Under Test which is the verification component/feature being tested.

Mock-Up Units – Since the Unit Under Test may have dependencies on other verification components and even RTL components to function properly, the unit test testbench must create "mock" versions of these missing components to enable the desired testing. The Unit Under Test itself is also considered a mock object. Different testcases may target different mock objects within the same testsuite.

TestSuite – The testbench the Unit Under Test and any required Mock-Up Units reside in.

TestCase – A single test run on a TestSuite.

Assertions – Methods which aide in checking the desired Unit Under Test behaviour. Error expectation is one important type provided.

VIP COMPONENT TESTING

In the following examples we've developed tests based upon the UVM Reference Flow [2]. This testbench applies UVM-e to a sample subsystem which implements an APB to UART controller design. This relatively simple testbench still has most of the structures that appear in standard UVM-e environments, but gives us a readily available sample to which we can apply the unit test concepts.

Transaction Structs

Some of the first verification code to be written in a project is often the structs which represent the transactions for the DUT. These range from a single cycle, low level payload object to a multi-cycle, layered protocol packet. Besides physical storage for the transaction payload, these objects typically contain methods for use by other verification components in the testbench. These methods can often be easily unit tested since they use only information contained in the transaction struct itself. Some examples are:

- Packing/Unpacking the transaction to/from a raw bit stream or lower level transaction type.
- Validating the correctness and legality of the transaction field values.
- Calculating and checking protection codes like parity, CRC, and ECC.
- Printing the transaction in specific formats.

Figure 2 demonstrates an UVM-e unit test on the UART frame struct. The first frame created is completely legal and all the respective check functions pass along with no dut_errors being thrown. The second frame created is full of problems and the unit test verifies that all methods and error messages recognize the errors. This unit test is very much a directed test and would probably be used at the very start of development. It's also not a bad way to get familiar with the UART frame struct object if the user were new to the code.

A pure TDD approach to using this test would be to code the dut_error expectations while the UART frame struct methods were just empty shells. The test would fail when run and the developer could then iterate until the desired error messages were all emitted.

One could picture making this unit test a bit more rigorous by randomizing or enumerating all the different control settings relevant to this frame class.

Figure 2 - UART frame methods testing

```
add testcase test_uart_frame_methods to uart_frame with
scenario {

   // frame object to test
   var uframe : uart_frame_s = new;
   gen uframe keeping {
      .databit_type == EIGHT;
      .parity_type  == ODD;
      .stopbit_type == TWO;
   };

   // test a good frame
   var bs_good : list of bit = %{0b111101010100};
   unpack(packing.low,bs_good,uframe);
   print "UFRAME_GOOD: ",uframe;
   eu_expect uframe.get_payload() to be %{0xaa};
   eu_expect uframe.calc_parity(bs_good,uframe.parity_type)
             to be 0;
   eu_expect uframe.parity_ok()   to be TRUE;
   eu_expect uframe.stopbit_ok()  to be TRUE;
   uframe.unpack_check_frame(bs_good,TRUE);
   eu_expect_no_more_dut_errors;

   // test a bad frame
   var bs_bad : list of bit = %{0b000101010101};
   unpack(packing.low,bs_bad,uframe);
   print "UFRAME_BAD: ",uframe;
   eu_expect uframe.parity_ok()   to be FALSE;
   eu_expect uframe.stopbit_ok()  to be FALSE;
   uframe.unpack_check_frame(bs_bad,TRUE);
   eu_expect_dut_error "Frame start bit is not 0";
   eu_expect_dut_error "Frame has bad parity";
   eu_expect_dut_error "Frame stop bit is not correct";
   eu_expect_no_more_dut_errors;
};
```

Benefits to Transaction Structs

If the checking in this example code looks too mundane, picture more complicated stacks like a layered packet protocol requiring correct assemble and disassemble methods at each layer transition. Making such methods robust before monitors and drivers rely on these methods would certainly benefit the development process. Debugging a pack or unpack problem or errant error message in a full-fledged interface UVC with RTL will definitely take more time than the quick unit test testbench.

Module UVC Development

Scoreboard Checking

In our legacy testbench environment, most of our unit testing work was focused on developing and verifying scoreboards and block checkers. These components are at the heart of our design verification strategy and must be of very high quality. They also require significant development time and effort, so engineer productivity is crucial.

Figure 3 - APB to UART scoreboard checking

```
import testsuite_uart_ctrl_scbd_monitor.e;

// Test uart_frame->apb path with correct payload
add testcase uart_input to uart_ctrl_scbd with scenario {
  var input_apb_trans : apb_trans_s;
  var uart_frame : uart_frame_s;
  gen uart_frame;
  gen input_apb_trans keeping {
    .addr == UART_RX_FIFO;
    .direction == READ;
    .data == pack(NULL,uart_frame.payload);
  };
  p_testsuite_uart_ctrl_scbd.ports_bundle.
                 mock_uart_frame_add$.write(uart_frame);
  p_testsuite_uart_ctrl_scbd.ports_bundle.
                 mock_apb_trans_match$.write(input_apb_tran
                 s);
  eu_expect_no_more_dut_errors;
};
// Test uart_frame->apb path with incorrect payload.  Expect
                 error
add testcase uart_input_bad_payload to uart_ctrl_scbd with
                 scenario {
  var input_apb_trans : apb_trans_s;
  var uart_frame : uart_frame_s;
  gen uart_frame;
  gen input_apb_trans keeping {
    .addr == UART_RX_FIFO;
    .direction == READ;
    .data == (1 ^ pack(NULL,uart_frame.payload));
  };
  p_testsuite_uart_ctrl_scbd.ports_bundle.
                 mock_uart_frame_add$. write(uart_frame);
  p_testsuite_uart_ctrl_scbd.ports_bundle.
                 mock_apb_trans_match$.
                 write(input_apb_trans);
  eu_expect_dut_error "Mismatch";
  eu_expect_no_more_dut_errors;
};
```

The reference environment implements a uart_ctrl_scoreboard to verify that the dataflow across the DUT from the APB interface to the UART interface is correct. Most of the functionality of the scoreboard is implemented using the UVM scoreboard package. We've implemented unit tests (Figure 3) to demonstrate how a simple scoreboard unit test would work.

The first test generates a uart_frame_s and a corresponding apb_trans_s. The uart frame is driven into the scoreboard by writing to the TLM analysis port. This mimics an incoming frame being driven into the DUT. Then the APB transaction is driven to the TLM match port. This imitates the testbench driving a read to the uart's RX FIFO register. This particular sequence of stimulus will cause the scoreboard to attempt to match the contents of the two transactions and compare the payloads. The eu_expect_no_more_dut_errors call asks the unit test framework to verify that no DUT errors have been detected during the test.

In the second test, the same sequence of scoreboard stimulus is applied, but the payload of the apb_trans_s transaction is corrupted. This mimics a design flaw in the DUT that should be detected by the checker. After the apb_trans_s is written to the scoreboard, we see **eu_expect_dut_error "Mismatch"** indicating that a dut_error call with the string "Mismatch" in it should've been observed. Failure to see this error will cause the test to fail, indicating that the scoreboard has not detected an error that it should have.

Benefits to Module UVC components

Checkers are a primary candidate to benefit from unit test from our point of view. Their main function is to detect and report errant functionality of the RTL DUT. Explicitly testing this error detect and report functionality makes a lot of sense given this viewpoint. Further, executing this verification coincident with or immediately following development has the engineer in a better mind-set

Figure 4 - Wire level checking test

```
add testcase drive_frame to uart_env with scenario {
  var frame : uart_frame_s;
  var out_data : list of bit;
  watchdog_timer = 10000;  // Keep the test running for 10K
              cycles
  var bfm := p_testsuite_uart_env.uart_env_u.tx_agent.
              as_a(ACTIVE TX uart_agent_u).bfm.as_a(TX
              uart_bfm_u);

              p_testsuite_uart_env.uart_env_u.tx_agent.as_
              a(has_monitor TX
              uart_agent_u).monitor.do_check = TRUE;
  var config := p_testsuite_uart_env.uart_env_u.config;
  for each (parity) in all_values(uart_frame_parity_t) {
    for each (stopbit) in all_values(uart_frame_stopbit_t) {
      for each (databit) in all_values(uart_frame_databit_t) {
        config.parity_type = parity;
        config.stopbit_type = stopbit;
        config.databit_type = databit;
        gen frame keeping {
          it.parity_type  == parity;
          it.stopbit_type == stopbit;
          it.databit_type == databit;
          it.legal_frame  == TRUE;
          it.delay_to_next_frame == 0;
        };
        var parity : list of bit = frame.with_parity ?
              {frame.as_a(with_parity uart_frame_s).parity} :
              {};
        var expected : list of bit = %{frame.start_bit,
              frame.payload, parity, frame.stop_bit};
        p_testsuite_uart_env.sig_sec_cdma_ctsb$ = 0;
        p_testsuite_uart_env.clk();
        out_data.clear();
        all of {
          {  // Packet stimulus thread
            bfm.drive_frame(frame);
            p_testsuite_uart_env.sig_sec_cdma_tx_data$ = 1;
          };
          {  // Clock & wire thread
            for i from 1 to expected.size() {
              p_testsuite_uart_env.clk();
              out_data.add0(p_testsuite_uart_env.
                sig_sec_cdma_tx_data$);
            };
            p_testsuite_uart_env.clk();
          };
        };
        eu_expect_no_more_dut_errors;
        eu_expect out_data to be expected;
      };
    };
  };
};
```

for finding software errors while the code development is fresh in their minds.

Besides verifying the main purpose of the checkers, unit tests allow the entire checker to be tested before any RTL is available. This is a common situation in our experience since design specs are usually available to both the RTL and DV engineers at the same time. How nice would it be if the RTL turn-on had a proven checker ready to go day one? The benefit we've observed is less time wasted back-tracking in the verification software during the middle and later phases of the project and more time spent identifying and debugging the implementation flaws in the RTL.

Interface UVC Development

Wire-level Checking

Developing a unit test for wire-level functionality of an interface UVC requires some judgement. These tests can be tricky to get right and may be less useful than the standard example topology and demo.sh script. Where unit tests for BFMs or monitors may be highly effective is when the env is configurable. Turning on the BFM may end up being simpler when all of the possible configuration options can be quickly iterated through while testing for correct operation. This can be a lot less tedious than other methods.

Note Figure 4, a unit test in which the uart_env mockup is setup for each possible set of configuration parameters. A uart_frame_s is then generated and driven by the BFM by calling the drive_frame() method. While drive_frame() is running, another thread is running to collect the output of the uart's TX port. At the end of the frame, the output collected is compared to an expected bitstream looking for discrepancies. Simultaneously, the TX monitor has been enabled to perform protocol checking of the frame. This test quickly checks all permutations of the parameter set for correctness for little extra cost relative to testing one particular configuration.

A similar unit test could be developed for the uart monitor code where the state machine is somewhat more interesting. We will discuss a unit test of the protocol checking of the received uart frame in **Packet Checking in the UVC Monitor**. This testsuite would be more targeted at validating the correctness of frame detection and creation by varying input signal timing. If written prior to development of the monitor, this testsuite could be incorporated as part of the BFM testsuite to enable holistic testing of all of the wire-level components of the UVC. In more complex protocols, this methodology would be very useful to ensure that the UVC can detect and recover from errors, both intentional and unexpected, on the interface. Often testing of these error conditions is started later in the hardware development cycle, leading to unexpected and hasty development work on the part of the UVC developer.

Packet Checking in the UVC Monitor

In our legacy environment, we did not expend any effort validating packet-level checkers as most of these were either complete, or incorporated in other, higher-level checkers. In UVM, packet checking is considered to be an integral part of the UVC monitor and needs to be tested for high-quality testing of packets. As a sample, we've developed a set of unit tests to verify the quality of the check_receive_data method of the uart_monitor_u unit. The testsuite constructs a uart_env and our test generates a random uart_frame packet which is packed and sent to the check_receive_data method. The test first calls the method with an unmodified frame and expects no errors to occur. We then corrupt the start bit, data, parity, and stop bits in turn, testing to ensure that the appropriate DUT error is reported by the check method. This basic flow is iterated over all permutations of payload size, parity setting, and number of stop bits to ensure that the checker will work under all conditions. The code for the test can be seen in Figure 5.

Development of this testcase took approximately two hours and involved debugging two latent defects in the checker code. Both of these defects were "silent" in that they caused the checker to not report errors, for instance when bad parity was sent from the DUT. One of the defects appears as though it may have been introduced when a developer modified the code to enhance the performance of the checker. These kinds of changes can be dangerous in that the checker's correctness might have been visually confirmed during initial development, but subsequently broken. Unit tests serve a valuable function in this instance, allowing the developer to re-check the correctness of the code after every change. Well written tests should expose any unintended side-effects of any changes made. This is where unit tests shine – verifying that all of the checks

Figure 5 - Packet protocol checker test

```
add testcase check_receive_data to uart_env with scenario {

  var frame : uart_frame_s;
  var monitor := p_testsuite_uart_env.uart_env_u.rx_agent.
                 as_a(has_monitor RX uart_agent_u). monitor;
  var config = p_testsuite_uart_env.uart_env_u.config;

  // Iterate all combinations of parity, stop bit and data width
  for each (parity) in all_values(uart_frame_parity_t) {
    for each (stopbit) in all_values(uart_frame_stopbit_t) {
      for each (databit) in all_values(uart_frame_databit_t) {
        config.parity_type = parity;
        config.stopbit_type = stopbit;
        config.databit_type = databit;
        gen frame keeping {
          it.parity_type  == parity;
          it.stopbit_type == stopbit;
          it.databit_type == databit;
          it.legal_frame  == TRUE;
        };
        var data : list of bit = pack(packing.low,frame);
        var parity_loc : uint = (1 + databit.as_a(int));
        var stop_loc : uint = parity_loc + ((parity != NONE) ?  1
                 : 0);
        var orig_data := data.copy();

        // Check with no error
        monitor.check_receive_data(data);
        eu_expect_no_more_dut_errors;

        // Corrupt the start bit
        data = orig_data.copy();
        data[0] = ~data[0];
        monitor.check_receive_data(data);
        eu_expect_dut_error "Frame start bit is not 0";
        eu_expect_no_more_dut_errors;
…
        // Corrupt the parity bit
        if(parity!=NONE) {
          data = orig_data.copy();
          data[parity_loc] = ~data[parity_loc];
          monitor.check_receive_data(data);
          eu_expect_dut_error "Frame has bad parity";
          eu_expect_no_more_dut_errors;
        };
      };
    };
  };
};
```

work at original development time and that none of the checks are subsequently broken.

Other Usage Models

Test Replay

In our C++-based environment, we developed unit testsuites for some of our existing software components. The goal was not to do classic test-driven development, but to apply some of the productivity enhancements to maintenance of those components. A good example of this was test replay. We instrumented the monitor ports of the component to be unit tested so that we could capture all of the input stimulus to the checker during a normal testcase run. The unit test was designed to read that input stimulus and apply it to the checker in the unit test environment. By doing this, we could rerun a long simulation in seconds, allowing the checker owner to easily develop a fix for the code while not having to wait for long tests to be rerun each time they made a change. Although there was overhead to develop the framework, it had a clear ROI on components which were actively being changed during the program.

We've not explored this concept within the eUnit Testing Framework to date. Specman provides more effective replay mechanisms than were available in the C++ environment, and this technique seems to be more appropriate to apply with existing components that don't have a unit test capability.

Generating Unit Tests from Simulation

Extending the replay concept, we added the ability to annotate and modify the trace taken from the simulation in our C++ environment. By changing a transaction in the trace and adding an annotation to "expect the following error at this time", you can develop checker validation tests that are much more sophisticated than those written completely by hand. This kind of checker validation can be very powerful in the right context, although we did find issues with it that would need to be addressed. Specifically, the tests developed this way may or may not remain valid as the RTL becomes more mature. Also, it can be somewhat time consuming to find appropriate traces and modify them. One area where this could be of considerable use is in capturing particular bugs to ensure that the checker will still detect them once the RTL is corrected.

Stimulus/Constraints

In our previous environment, another successful use of unit testing was for traffic generation. We had the equivalent of an interface UVC with many knobs to control various aspects of the traffic generation. Unit tests were coded to generate traffic under various knob settings and validate the resulting traffic did indeed obey the knobs. We believe a similar approach may be applicable to UVM-e interface UVC's at the get_next_txn() interface between the sequence_driver and the BFM. The knobs in this case would be constraints.

One addition to this idea is to incorporate cover groups into these unit tests. The hope is coverage holes may be identified early by looking at the resulting coverage after running the core set of sequences for this interface UVC.

BENEFITS

Verification IP development starts earlier

No RTL is needed to begin full development and thorough testing of Verification IP.

Shorter development time

Testing of your VIP coincident with or immediately following development has the engineer in a better mind-set for finding software errors while the software design is fresh in the head.

The VIP development and test cycle is continuous as unit test results are available quickly. There is less need to multitask while waiting for simulation results.

Less debugging of VIP in larger, complex environments. Most issues will be found in small unit tests.

Fewer verification holes

Error detection and messaging has been unit tested. All configurations covered. Critical verification components can be adequately tested for quality before exposure to RTL.

RTL turnon really is just RTL turnon

Verification components are already tested and turned on before RTL. The validation portion of the project stays focused on RTL issues and testing.

Faster and better quality fixes for verification IP components

Reproduce a VIP bug with a unit test and use it to validate the software fix. Then rerun the unit test library to make sure your fix doesn't introduce any new issues.

Excellent learning tool for new component owner

When the component is turned over to a new owner, these tests provide excellent training tools and help keep the quality of the UVC high during the learning phase.

CONCLUSIONS

In our experience, using an architected framework to apply TDD practices to verification software improves its quality and accelerates the overall design schedule. The new unit test framework in UVM-e allows us to continue this important design practice on our next generation verification environment.

REFERENCES

[1] C. Larman, *Applying UML and Patterns: Hands-on Mastery of OOAD, Agile Modeling, and Patterns with TDD,* 2010.

[2] Cadence, "UVMWorld Contribution," May 2013. [Online].

Available: http://forums.accellera.org/files/file/99-uvm-ref-flow-soc-kit-originally-from-cadence-modified-to-work-with-vcs-questa-ius/.