Do You Verify Your Verification Components?

Josh Rensch
Superion Technology
Josh.Rensch@superiontech.com

Neil Johnson
XtremeEDA
njohnson@xtreme-eda.com

Why Verify Components?

Today's verification requires significant ramp up to build an environment. There is the huge learning curve of the standards; both SystemVerilog and UVM require a large amount of time investment in order to master. Not only that, but all too often verification engineers take the approach of using the RTL design to validate the correct functionality of a verification component. Now testing is being done on the verification environment and the RTL at the same time. This leads to a couple of issues. One issue is a potential delay. Both the design and the verification environment need to have the same feature up and running to test them against each other. The second issue is doing this introduces two variables into testing, the RTL and its corresponding verification environment. This expands the state space of where bugs reside. Is the bug in the RTL, verification, documentation or specification? There is a better method to develop verification code in a systematic way. Creating a systematic approach is the reason behind the development of SVUnit. Using the concept of Unit Testing from Test Driven Development, SVUnit is a way to develop verification building blocks. First step is to build a failing testcase, and then write the code that makes that test pass. This allows the verification of your verification components before the RTL design is ready as well as isolating the bugs. This paper will discuss Test Driven Development, Unit Testing and how to setup and use SVUnit.

What is TDD?

The concept of TDD comes from the software world's Agile Development. Unlike in the Waterfall process, with TDD flow, the *Verification* stage is before the *Implementation* stage. The idea is to first determine a feature of the code (in this case a verification component) and then write a test to test that feature. This test must fail. Only after the testcase fails, is the verification component feature written that the test was checking. After the test is passing, the code is refactored. Refactoring is the process of improving the code's readability and quality. Then, a new failing testcase is created for the next feature. This process is sometimes referred to as RED-GREEN-REFACTOR.

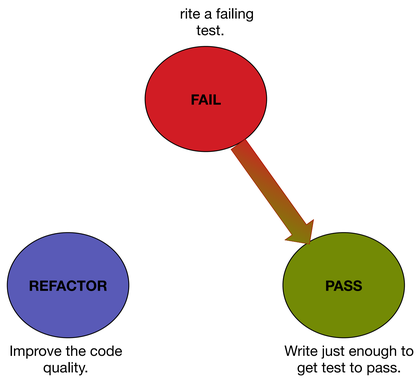

Figure 1: TDD Flow

Why Refactor Code?

The focus of the first two steps of the TDD flow is creating a passing test case. These steps may produce code that is not readable, straight forward or even commented. Common examples of issues fixed in the refactoring stage include breaking down large classes, removing duplicate code, or splitting a method that is simply too long.
Refactoring these sections will make the code easier to work with by making it not only easier to read, but also resolving hidden or dormant issues that may exist within the code. The beauty of the TDD flow is that testing the code after a refactoring exercise is a simple process, since the tests have already been created.

What is a Unit Test?

A Unit is the smallest piece of your code to be tested. In design, it would be the modules that make up the design. In UVM, it would be individual verification components. So unit testing is just testing individual pieces of a design or verification component. But to do that, TDD requires that each piece is tested in isolation. For example, in UVM, normally a driver works in concert with a UVM sequencer. In TDD, testing the driver should be independent of the sequencer. This means some sort of framework is needed to build these tests quickly. It would make it so much simpler to automate the whole process. A Unit Test is a testing framework used to implement TDD and it does just that. It relies on automation to make the testing quicker and simpler.

Using SVUnit - Overview

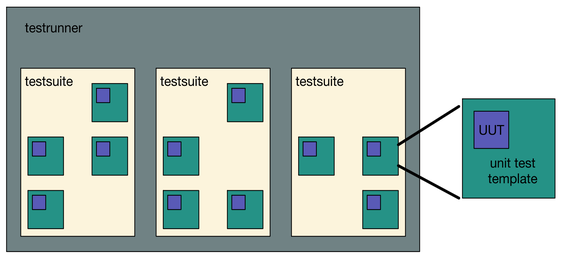

Figure 2: SVUnit Test Structure

SVUnit is a Unit Test framework for SystemVerilog and UVM. This simple verification framework is intended for design and verification engineers writing and running tests against Verilog modules, classes or interfaces, as well as against UVM components and objects. SVUnit uses a 3-level hierarchical structure. At the lowest level, SVUnit is built from a unit test template which contains a simple unit test along with the Unit Under Test (UUT). In UVM, individual pieces would go to Unit Test, grouping pieces to make a Testsuite would be like an agent and a whole protocol tester would be testrunner. Let's go back to our earlier idea of creating an agent. The components of an agent are the driver, the monitor, the sequence item and the interface. Each one of these would have their own unit test template. These would be grouped together in the next level of hierarchy where there is a test suite.

Using SV Unit

How to setup up and configure SVUnit can be found at http://www.agilesoc.com/open-source-projects/svunit/svunit-user-guide/ or in our full paper. The example demonstrates testing a unit called simple_model. The UUT is a simple component which has a put and a get port. The sequence item has only one random field. Each test inside of SVUnit has a macro at it's beginning and end. Inside the parentheses is the name of the testcase.

```
`SVTEST(get_port_not_null_test)
  `FAIL_IF(my_simple_model.get_port == null);
`SVTEST_END
```

These macros register the test with the SVUnit test runner. In this case the test is checking if the get port is not null. For behavior that is repeated before and after every test, the setup() and teardown() tasks in the unit test template are intended to group any logic that is repeated before and/or after every test, respectively.

```
//=====================================
  // Setup for running the Unit Tests
  //=====================================
  task setup();
    svunit_ut.setup();

    //-------------------------------------------------------
    // activate the component (i.e. add the component to
    // the default uvm_domain)
    //-------------------------------------------------------
    svunit_activate_uvm_component(my_simple_model);

    //-----------------------------
    // start the testing phase
    //-----------------------------
    svunit_uvm_test_start();
  endtask
```

In order to verify UVM components, svunit_uvm_test_start() must be called. This will kick off the running of the UVM phases. SVUnit has run and been tested on the big four simulators and comes with scripts and helper functions to help you get started.

Benefits of TDD and Unit Testing

- Applying TDD helps to limit scope creep. By creating an upfront list of exactly what the RTL and corresponding verification components are supposed to do and creating a test for each feature, unplanned or unnecessary tests can be avoided.
- Having these little tests that all pass allows changes in the code to be made with assurances that a change doesn't break any previous functionality.
- Due to the bottom up approach to testing, you know your building blocks are in great shape so integration testing becomes easier.
- As engineers, we traditionally balk at documentation. It's sort of in our DNA. Unit tests that are created correspond to the critical parts of the components and therefore are a sort of living document that can be extracted and used to create more formal documentation later.
- These unit tests are small and therefore run faster than the traditional full random simulation.
- With SVUnit test cases packaged with the VIP, you can make small changes in the agent and make sure they don't break the overall functionality of the agent.
- TDD is very systematic and structured. At first, the rigidity of it feels constraining and time consuming. This process makes not only cleaner code but a more productive coder.

Real World Results

In order to test SVUnit, an experiment was needed. What better way to test SVUnit than on the code most environments were created with? UVM-UTest is an open-source initiative that demonstrates the value of unit testing relative to an industry standard code library. In UVM-UTest, unit test suites were written for several core components of UVM. The intent was to rigorously verify the functionality of each component in isolation, an approach uncommon in hardware verification.

UVM-Utest core UVM components :
uvm_object, uvm_misc, uvm_printer, uvm_component

2 engineers + 6 weeks = 500 UVM unit tests

500 UVM unit tests + ~14 seconds (in simulation) =

10 UVM Defects!

Viewing Issues (1 - 10 / 10) [Print Reports]
[CSV Export]

P	ID	Type	Category	Severity	Status	Updated	Summary
	0004602	TBD	BCL	minor	new	2013-06-19	uvm_printer::print_object_header cannot support user specified scope_separator
	0004600	TBD	BCL	minor	new	2013-06-19	incomplete string/separator handling in uvm_leaf_scope
	0004640	TBD	BCL	minor	new	2013-06-17	uvm_has_wildcard is the only function that treats '+' as a wildcard
	0004638	TBD	BCL	minor	new	2013-06-17	uvm_is_array returns true for malformed string inputs
	0004637	TBD	BCL	minor	new	2013-06-17	incomplete is_wildcard handling in uvm_get_array_index_string leaves is_wildcard in erroneous state
	0004636	TBD	BCL	minor	new	2013-06-17	uvm_get_array_index_string accepts illegal index characters
	0004635	TBD	BCL	minor	new	2013-06-17	incomplete is_wildcard handling in uvm_get_array_index_int leaves is_wildcard in erroneous state
	0004634	TBD	BCL	minor	new	2013-06-17	uvm_get_array_index_int treats indices with radix specified as illegal
	0004609	TBD	BCL	minor	new	2013-05-28	uvm_printer::print_real argument to adjust_name is incorrect
	0004601	TBD	BCL	minor	new	2013-05-24	uvm_vector_to_string incorrectly displays negative numbers

Figure 3: UVM Defect List

Conclusion

Writing code is challenging and it's not getting easier; not only from a technical standpoint but also because it's more of an art than engineers would care to admit. Engineers are thrown into the fire, asked to complete a verification task with little to no planning or documentation - forced to peek at the design code in order to determine what should be tested. Everyone wants to create high quality code that is bug free and easy to read. If that is the case, the tests should not just test that the design behaves as coded; instead tests should target desired behavior. Agile software techniques such as TDD and Unit Testing offer a different approach for developing more robust verification environments.
To be fair, TDD and thereby SVUnit may not be the best solution for every project, but they both bring along a very systematic approach to developing verification environments that is sorely lacking in current approaches. It gives structure to the art of coding, making it possible to create cleaner code. TDD has been in use in the software world for some time now and it continues to be utilized to help create more complicated code with fewer bugs, and on tighter schedules.

Acknowledgements

We would like to thank both Superion Technology and XtremeEDA for giving us the opportunity to work on this. We'd like to thank all those of you using SVUnit.