How Do You Verify Your Verification Components?

Josh Rensch
Superion Technology Group, Inc.
Josh.Rensch@superiontech.com

Neil Johnson
ExtremeEDA Corporation
njohnson@xtereme-eda.com

Abstract - **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.**

I. Introduction

There are a number of opportunities for improvement in today's verification systems created by the steep learning curve to ramp up to build an environment. Everyone realizing that there is simply too much to be done in the amount of time that an average project has. How does one solve these issues? Management and executive's solutions revolve around incentivizes for employees to work harder, faster and longer. Industry committees are trying to create standards that remove the need for engineers to re-invent the wheel, be it protocols or languages. The EDA industry answer is to continuously creating new tools to expedite the day to day work of an engineer. These tools could be speeding up creation of stimulus, building platforms to run simulations faster or creating verification for standard protocols. The big thing that is missing is the lack of a systematic way to develop code in the hardware design space.

Many would argue that there is a systematic approach to creating RTL and corresponding verification. These people would point to the Waterfall Method as the main methodology to developing System on a Chip (SoC). At a high level, there might be some validity to this claim. But, Waterfall is heavily front loaded in it's process, the SoC does is rarely nailed down in a project and tends to be more fluid. The Waterfall Method is not being adapted to the climate of perpetual change involved in modern development. It was developed for software around 1970 and is reflective of it's time.

Another issue with the Waterfall Method is the verification testing is later on in the methodology. This means a verification engineer is often not involved early in the development cycle. In turn, this can lead to SoC features being nigh impossible to verify due to the focus on the implementing the design and not necessarily testing it.

The software industry has recognized the limitations of this methodology and has tried a number of alternative methodologies. In the early 2000's software engineering moved toward an Agile development process [1]. Not all Agile development processes and techniques apply to hardware design verification. It may not have all the answers but it solves a number of problems that inherit in the Waterfall Methodology.

Many people like to argue that software is different than hardware. But hardware and verification engineers have been borrowing from the software world for some time. Universal Verification Methodology (UVM) [2] and SystemVerilog are examples of taking concepts from the software world to make tasks in the hardware world easier. Both are based on object-oriented programming, a fundamental programming principle in this day and age. Both have concepts built into them borrowing from software engineering's design patterns, like Factory Method, Builder, and Singleton to name a few [3].

If the industry is going to borrow from the software engineering world, then it should only be natural to look at how software engineers have created a system to solve the ever changing requirements and the issue of releasing buggy code. That system is more and more Agile like.

To explain the differences between Agile and a traditional Waterfall SoC development cycle; let us look at the stages for Waterfall methodology. *Requirements*, *Architecture*, *Implementation*, *Verification* and *Maintenance*. Requirements is where the design team hashes out exactly what it is going to build. The requirements may come from a customer, from the internal marketing team, or be derived from requirements of a higher level product. After that point, the verification and hardware design break off to go through independent Architecture and Implementation stages. During the Verification stage, both the design and the verification environment need to be somewhat up and running with the same feature to test. Testing two things at once introduces more variables into testing. Is the design correct? Is it the verification? Is it the requirements? Is it the documentation? One or all of these could be incorrect.

What is needed is a systematic approach to testing as well as limiting the variables that are tested. There are two Agile techniques that can help teams develop a more systematic approach to the creation of better verification components: Test Driven Development (TDD) and Unit Testing. These can be used stand alone without any other Agile techniques or processes. The next section will explain what those two pieces are.

II. What is Test Driven Development and Unit Testing?

Test Driven Development

As mentioned in the introduction, 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 and is shown in Figure 1: TDD Flow.

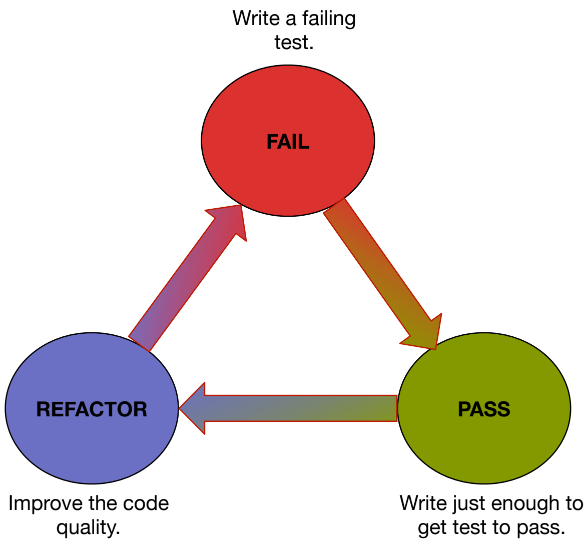

Write a failing test.
FAIL
PASS
Write just enough to get test to pass.
REFACTOR
Improve the code quality.

Figure 1: TDD Flow

This sounds like a daunting task. If a verification team was to write a test for, say, an agent, and then write the agent, it would be a lot of work. Why go through all that trouble? That's not quite how it's done. The trick for TDD to work is to create a test for a subset of the agent instead of the whole thing. A tiny piece, even a single line of code, could be enough for a test. The TDD created tests should be as small as possible. An example would be a UVM driver; in the driver, a test around the get_port() functionality would be built. The TDD test would then implement a put_port() and a check to make sure the output was equal to the input.

The whole process should take a relatively short amount of time. At times it might feel like no progress was made. The test writer might think "This too small to test". That is normal. That is the level of detail needed for this technique to be effective. Many tests are created, giving the component a series of tests that verify its functionality. The point here is to make sure each component is tested for what it needs to do. If the environment is made up of a number of these tested components, the confidence in the whole environment is higher.

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.

Unit Testing is used in a number of different languages, and even within a language there can be different flavors. Languages like Java have been using it since 2002 [4]. Most of the time it's built into an integrated development environment (IDE) for the language, but some languages have it built in to the language itself, like Cobra and Go [5]. Unit Test frameworks have ways to isolate the tested code. These frameworks make unit test development easier by using method stubs, mock objects, test harnesses and fakes. The next section details using SVUnit, a Unit Test for SystemVerilog and UVM [6].

III. Using SVUnit

Overview

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. In the case of this paper, it will be used to verify a simple verification component. By first verifying in isolation the building blocks that make up your design or testbench, the quality of those blocks is higher when integrated into a whole.

The feature characteristic of SVUnit is its ability to get developers up-and-running in an hour or less. SVUnit is open-source under Apache 2.0 licensing, making it accessible and productive for any engineer with a SystemVerilog simulator. This section is largely based on the SVUnit documentation and user guide [6] .

SVUnit Structure and Workflow

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). The next level would be a test suite, followed at the top by a test runner.

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.

Where does test runner come into the picture? Well, say this agent is made up of a bunch of sequences and a scoreboard like in most verification IP packages. A test suite would be created for the sequences and another would be created for the scoreboard. All these test suites would then be grouped together under one test runner. Figure 2 shows a generic test test structure.

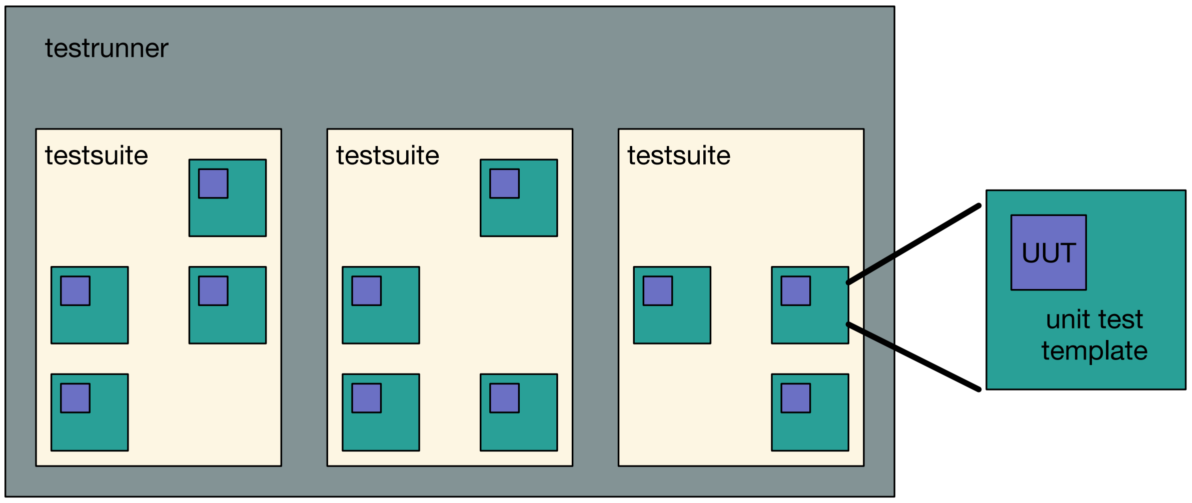


FIGURE 2: SVUNIT TEST STRUCTURE

Hierarchy is derived from the file/directory structure where all unit test templates located within the same directory are grouped into a test suite.

Installation and Setup [6]

Download to setup is about a 3-minute procedure:

1 Download the latest version of SVUnit from SourceForge at http://sourceforge.net/projects/svunit/
2 Extract the SVUnit archive
3 Source the SVUnit setup script (bash users source Setup.bsh. csh users source Setup.csh)
   A > *`cd svunit-code`*
   B > *`source Setup.bsh`*

Creating a Unit Test Template

New unit test templates are created using create_unit_test.pl. Usage of create_unit_test.pl is as follows…

```
Usage:  create_unit_test.pl [ -help | -out <file> | -i | -overwrite | uut.sv ]
Where   -help                 : prints this help screen
        -out <file>           : specifies a new default output file
        -overwrite            : overwrites the output file if it already exists
        uut.sv                : the file with the unit under test
```
FIGURE 3: CREATE_UNIT_TEST USAGE

For example, you can generate a new unit test template for a class called 'simple_model' with:

```
> create_unit_test.pl simple_model.sv
```

The default output for create_unit_test.pl is written to ./<name>_unit_test.sv. The default can be overridden, however, using the -out <file> switch. A different file name and/or directory can be specified as required. The file name, however, must follow the <name>_unit_test.sv format. Existing files will not be overwritten unless the -overwrite switch is used.

Integrating the UUT

The template generated by create_unit_test.pl includes an instance of the UUT as well as other important parts of the infrastructure. Inside the download of SVUnit install under the example directory, is the example used in this paper. It's a simple uvm_component along with a uvm_sequence_item. It has a put and get port and it moves the transaction to and from the component. This example is meant to be simple for purposes of this paper. The example is shown abridged in Figure 4 and Figure 5.

```
class simple_model extends uvm_component;
  uvm_blocking_get_port #(simple_xaction) get_port;
  uvm_blocking_put_port #(simple_xaction) put_port;
  .
  .
  .
endclass
```

FIGURE 4. SIMPLE_MODEL CODE

```
class simple_xaction extends uvm_sequence_item;
  rand int field;
  .
  .
  .
endclass
```

FIGURE 5. SIMPLE_XACTION CODE

Writing Unit Tests

The unit test template includes embedded instructions on structure and position of unit tests:

```
  //===================================
  // All tests are defined between the
  // SVUNIT_TESTS_BEGIN/END macros
  //
  // Each individual test must be
  // defined between `SVTEST(_NAME_)
  // `SVTEST_END
  //
  // i.e.
  //   `SVTEST(mytest)
  //     <test code>
  //   `SVTEST_END
  //===================================
  `SVUNIT_TESTS_BEGIN

  `SVUNIT_TESTS_END
```

FIGURE 6. MACRO DEFINITION OF SVUNIT TEST GROUPING

Multiple unit tests can be defined in a unit test template. All must be defined between the `SVUNIT_TESTS_BEGIN and `SVUNIT_TESTS_END macros. Individual unit tests are defined using the `SVTEST(<name>) and `SVTEST_END macros. For example, let’s test the get_port of our simple model.

```
  `SVTEST(get_port_not_null_test)
     …
     …
  `SVTEST_END
```

FIGURE 7. EXAMPLE MACRO DEFINITION FOR SIMPLE_MODEL.

The macros expand to a Verilog code block so any code that is legal within a code block can be used within a unit test. Other required variables, declarations, functions, tasks, etc. must be defined outside the BEGIN/END macros.

SVUnit Reporting Macros

SVUnit includes an integrated reporting mechanism such that the exit PASS/FAIL status of every unit test is collected, reported and used to report a cumulative result. To set PASS/FAIL status, SVUnit defines several logging macros that are integrated with the reporting structure.

```
  `define FAIL_IF(exp)
  `define FAIL_IF_LOG(exp,msg)
  `define FAIL_IF_EQUAL(a,b)
  `define FAIL_UNLESS(exp)
  `define FAIL_UNLESS_LOG(exp,msg)
  `define FAIL_UNLESS_EQUAL(a,b)
  `define FAIL_IF_STR_EQUAL(a,b)
  `define FAIL_UNLESS_STR_EQUAL(a,b)
```

FIGURE 8. REPORTING MACROS FOR SVUNIT

The most commonly used macros are `FAIL_IF and `FAIL_UNLESS that take a single boolean expression as input. The `FAIL_IF_EQUAL and `FAIL_UNLESS_EQUAL macros exit based on an '===' comparison of boolean inputs a and b. Likewise, `FAIL_IF_STR_EQUAL and `FAIL_UNLESS_STR_EQUAL do a string comparison between inputs a and b.

Setting Test Exit Status

The reporting macros can be used to verify outputs and response of a UUT and set test exit status accordingly. For example, we can use the `FAIL_IF from Figure 7 to make sure that the get_port isn't null.

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

FIGURE 9. FULL FUNCTION OF EXAMPLE UNIT TEST

If the conditions described by the macros in get_port_not_null_test are not satisfied, the test fails with an assert error and is reported as having failed. Tests are killed immediately at first failure so any code appearing after the failing assert statement does not execute.

Interacting with the UUT

Tests can interact with the UUT using simple procedural assignments to inputs or through helper functions and tasks for more complex interactions. These helper functions and tasks do not need to be defined inside the macro of `SVTEST.

Test Setup and Teardown

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 – the setup() task is run before every test and the teardown() task is run after every test. In the case of using a UVM component, SVUnit needs to have an understanding of UVM phases, so the line svunit_activate_uvm_component() is used with the parameter of my_simple_model.

```
    //===================================
    // 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
```

FIGURE 10. DEFINITION OF SETUP() FUNCTION

It is recommended that common initialization code be contained in the setup() task. Reset sequence or register initialization, for example, is common logic that should be included in the setup() task. As well, it is recommended that any general cleanup of the UUT or unit test harness be grouped in the teardown() task to avoid polluting the state space for subsequent tests (i.e. teardown() is for "cleaning the slate").

Running Unit Tests

```
  Usage:  runSVUnit -s|--sim <simulator> [-l|--log <log> -d|--define <macro> -f|--filelist <file>
-U|-uvm -r|--r_arg <option> -c|--c_arg <option> -o|--out <dir> -t|--test <test>]
    -s|--sim <simulator>     : simulator is either of questa, modelsim, riviera, ius or vcs
    -l|--log <log>           : simulation log file (default: run.log)
    -d|--define <macro>      : appended to the command line as +define+<macro>
    -f|--filelist <file>     : some verilog file list
    -r|--r_arg <option>      : specify additional runtime options
    -c|--c_arg <option>      : specify additional compile options
    -U|--uvm                 : run SVUnit with UVM
    -o|--out                 : output directory for tmp and simulation files
    -t|--test                : specifies a unit test to run (multiple can be given)
    -h|--help                : prints this help screen
```

FIGURE 11. USAGE OF RUNSVUNIT

Choosing a Simulator

SVUnit can be run using most commonly used EDA simulators using the '-s' switch. Supported simulators currently include Mentor Graphics Questa, Cadence Incisive, Synopsys VCS and Aldec Riviera PRO.

Adding Files for Simulation

Through the use of `include directives, both the unit test template and corresponding UUT file are included in compilation making it possible to build and verify on simple designs without any need to specify or maintain file lists. As designs grow, however, more files can be added using standard simulator file lists and the '-f' switch.

NOTE: The file svunit.f is automatically included for compilation provided it exists. Thus, files can be added to svunit.f without having to specify '-f svunit.f' on the command line.

Adding Run Time and/or Compile Time Options

It is possible to specify compile and run time options using the '-c_arg' and '-r_arg' switches respectively. All compile and run time arguments are passed directly to the simulator command line.

Enable UVM Component Unit Testing

For unit testing UVM-based components, the '-U' switch must be specified to include relevant run-flow handling.

Specifying a Simulation Directory

By default, SVUnit is run in the current working directory. However, to avoid mixing source files with simulation output, it is possible to change the location where SVUnit is built and simulated using the '-o' switch. It is an error to use the '-o' switch to runSVunit in a directory that doesn't exist.

Specifying Unit Tests to be Run

By default, runSVUnit finds and simulates all unit test templates within a given parent directory. For short runs, this is recommended practice. However, if simulation times grow to the point where they are long and cumbersome, it is possible to specify specific unit test templates to be run using the '-t' switch. For example, if a parent directory has 12 unit test templates but you only want to run mine_unit_test.sv, you can use the '-t' switch as:

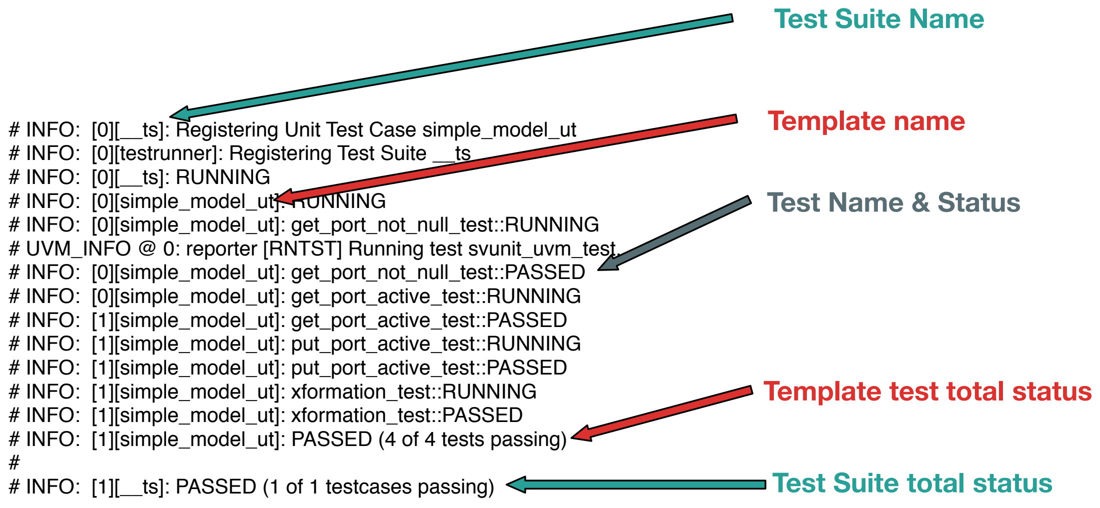
```
> runSVunit -t mine_unit_test.sv -s <your simulator>
```

The '-t' switch can be used to specify multiple unit test templates as:

```
> runSVunit -t mine_unit_test.sv -t yours_unit_test.sv -s <your simulator>
```

Simulator Output

Using built-in logging macros, the logged SVUnit output shows step-by-step run status for each test, unit test template and test suite as well as a cumulative result for the testrunner.



```
# INFO:  [0][__ts]: Registering Unit Test Case simple_model_ut
# INFO:  [0][testrunner]: Registering Test Suite __ts
# INFO:  [0][__ts]: RUNNING
# INFO:  [0][simple_model_ut]: RUNNING
# INFO:  [0][simple_model_ut]: get_port_not_null_test::RUNNING
# UVM_INFO @ 0: reporter [RNTST] Running test svunit_uvm_test
# INFO:  [0][simple_model_ut]: get_port_not_null_test::PASSED
# INFO:  [0][simple_model_ut]: get_port_active_test::RUNNING
# INFO:  [1][simple_model_ut]: get_port_active_test::PASSED
# INFO:  [1][simple_model_ut]: put_port_active_test::RUNNING
# INFO:  [1][simple_model_ut]: put_port_active_test::PASSED
# INFO:  [1][simple_model_ut]: xformation_test::RUNNING
# INFO:  [1][simple_model_ut]: xformation_test::PASSED
# INFO:  [1][simple_model_ut]: PASSED (4 of 4 tests passing)
#
# INFO:  [1][__ts]: PASSED (1 of 1 testcases passing)
```

FIGURE 12. OUTPUT OF A RUN OF SVUNIT

IV. BENEFITS OF TDD AND UNIT TESTING

The are a number of benefits to using TDD and SVUnit. One of the main benefits is finding bugs sooner in the design cycle. The cost of finding a bug earlier in the cycle can be up to 5 times less than later in the cycle. [7] This cost is comprised of financial, schedule and/or effort.

Applying TDD also helps to limit scope creep. People tend to want to put a lot of stuff into a design for “just in case.” 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, and then only writing enough code to make each test pass, it limits this risk [8].

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. You have a test for everything and when you add a new feature you create another test. This gives both you and the team confidence in the code.

Due to the bottom up approach to testing, you know your building blocks are in great shape so integration testing becomes easier. It’s difficult to build a house, if the material is either shoddy or has unknown properties. The same should hold true when it comes to the creation of a verification environments.

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. Tools like Doxygen [9] or Natural Docs [10] could be used to extract these tests along with comments and create documentation.

These unit tests are small and therefore run faster than the traditional full random simulation. This means that the tests won’t tie up the simulator licenses for long periods of time. It also acts as a great sanity test before check ins to make sure that whatever change was made does not break any of the tests.

The other benefit is that when building agents in UVM, they tend to become a part of a Verification IP (VIP) library that is used long after the SoC is completed. 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. Most companies want a re-use library and this gives them one that can be adaptable to future projects.

TDD is very systematic and structured. At first, the rigidity of it feels constraining and time consuming. But as you work with it, this process makes not only cleaner code but makes a more productive coder. The reason behind this is that it limits decisions a coder can make in a pass through the TDD cycle, one small change. By limiting the decisions, the coder reduces their decision fatigue. This in turn results in more energy. Many successful people, including President Obama, choose their wardrobes to limit decision fatigue [11].

V. Real World Result

In order to test SVUnit, an experiment was needed. What better way to test SVUnit then 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 the UVM. The intent was to rigorously verify the functionality of each component in isolation, an approach uncommon in hardware verification.

The result of UVM-UTest unit testing was 10 obvious defects found by 2 verification engineers over the course of 6 weeks. These were very basic defects found in core UVM components - namely the uvm_object, uvm_misc, uvm_printer and uvm_component - that have slipped through years of UVM usage and existing verification techniques entirely undetected.

In terms of test suite size, >500 unit tests are defined in UVM-UTest. For performance, all test suites run in a single executable in approximately 14 seconds; runtimes are comparable between leading industry simulators. This was described in full in the paper, "How UVM-1.1d Makes the Case for Unit Testing" [12].

VI. 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.

References

[1] Agile Alliance, "The Agile Manifesto," 2001. [Online]. Available: http://agilemanifesto.org/history.html. [Accessed 2016].

[2] Accellera, "Accellera.org," December 2015. [Online]. Available: http://accellera.org. [Accessed December 2015].

[3] E. Gamme, R. Helm, R. Johnson, J. Vlissides and G. Booch, Design Patterns: Elements of Reusable Object-Oriented Software, 1st Edition ed., Addison-Wesley Professional, 1994.

[4] JUnit, "JUnit-About," 2002. [Online]. Available: http://junit.org. [Accessed January 2016].

[5] GitHub, "testing - The Go Programming Language," November 2015. [Online]. Available: https://golang.org/pkg/testing/. [Accessed 5 January 2016].

[6] N. Johnson, "SVUnit User Guide," August 2015. [Online]. Available: http://www.agilesoc.com/open-source-projects/svunit/svunit-user-guide/. [Accessed December 2015].

[7] B. W. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," *Computer,* vol. 34, no. 1, pp. 135-137, 1 January 2001.

[8] K. Beck and C. Andres, Extreme Programming Explained Second Edition, Addison-Wesley Pearson Education, 2005.

[9] Doxygen, "Doxygen," 30 December 2015. [Online]. Available: http://www.stack.nl/~dimitri/doxygen/. [Accessed 6 January 2016].

[10] G. Valure, "Natural Docs," 2011. [Online]. Available: http://www.naturaldocs.org. [Accessed 6 January 2016].

[11] J. Haltiwanger, "The Science of Simplicity: Why Successful People Wear the Same Thing Every Day," Elite Daily, 14 November 2014. [Online]. Available: http://elitedaily.com/money/science-simplicity-successful-people-wear-thing-every-day/849141/. [Accessed January 2016].

[12] N. Johnson and J.-M. Tremblay, "How UVM-1.1d Makes the Case for Unit Testing," 7 July 2013. [Online]. Available: http://www.agilesoc.com/2013/07/07/how-uvm-1-1d-makes-the-case-for-unit-testing/. [Accessed 2 January 2016].