An Efficient Scenario Based Testing Methodology Using UVM

Conference Paper · December 2016

DOI: 10.1109/MTV.2016.14

2 authors, including:

Khaled Salah
Siemens
163 PUBLICATIONS **605** CITATIONS

An Efficient Scenario Based Testing Methodology Using UVM

Khaled Fathy, Mentor Graphics, Cairo, Egypt (*khaled_kabil@mentor.com)*
Khaled Salah, Mentor Graphics, Cairo, Egypt (*khaled_mohamed@mentor.com)*

Abstract—**Most of modern verification architects use randomness supported by system verilog (SV) to enable defining a generic path for a test to follow. This generic path stresses on a subset of features, and allows randomization to explore corners in depth. Setting up such test case requires a well-defined stimulus generation methodology that consumes less time to cover all the corner-cases. Moreover, Off-the-shelf scenario libraries, synchronization and scheduling process methodology for the parallel stimuli need to be reused across several test cases. In this paper, a novel methodology for creating test scenarios is proposed. Moreover, making use of object oriented principles to build composite layered scenario sequences with a generic parallel stimuli synchronization process is introduced. The proposed methodology is built as a generic library code to be reused in many designs. The results of applying this methodology on test cases show enhancements on reusability, test cases count, coverage closure and performance.**

Index Terms — Stimulus Generation; UVM Sequences; Test Case Generation; Constrained Random Testing (CRT); Coverage Driven Verification (CDV)

I. INTRODUCTION

System Verilog (SV) language released in 2002 by Accellera committee, as a unified hardware design, specification and verification language, merging different language constructs from Vera, SuperLog, C, Verilog and VHDL languages [1]. IEEE then standardized SV in 2005 (1800-2005). SV supports behavioral, register transfer level (RTL), and gate level hardware descriptions. It is also suited for test bench creation by the inclusion of object oriented constructs, functional cover groups, assertions and constraint random constructs. SV also has application programming interface (API) to foreign programming languages. SV adoption as a verification language increased from 25% in 2007 to 75% in 2014 [2].

Universal Verification Methodology (UVM) is a common standardized verification methodology for test bench creation. UVM composed of reusable verification components (VC), in which VC is an encapsulated, ready to use, configurable environment for an interface protocol, a design submodule, or a full system [3]. UVM also standardized by Accellera in 2010, to unify the discrete verification methodologies created by EDA companies. UVM showed adoption increasing in test bench creation from 40% in 2012 to 70% in 2015 [2].

Recent survey, [2], reports several measured aspects for the functional verification process. Verification activities consume 57% of the project time in the survey, and for a time distribution, for a typical verification engineer tasks, is distributed as follow, 11% in planning, 17% in test bench development, 19% in test creation, 29% in debugging and 24% in other tasks (ex. scripting). This numbers prove that an attempt to automate some of these tasks will enhance the overall performance [4].

In this work, starting from the need to accelerate the test case creation process, as it consumes as much time as test bench development itself, a scenario based test case creation methodology based on UVM is proposed. The proposed methodology needs to be efficient and facilitates tests creation with the aim of accurate and enhanced functional verification results.

Our main contributions are:

1- Two new types of UVM sequences are introduced.
2- Automatic parallel stimuli synchronization.
3- Methodology is SV native, and doesn't require 3rd party tool.

The rest of this paper is organized as follows: Section II discloses stimulus generation related work in the literature. Section III defines the proposed test case creation methodology and the library implanted for that purpose. Section IV discusses the usage of the proposed methodology in two benchmark designs and the results. Section V is for the conclusion and the future work.

II. RELATED WORK

Approaches followed in the literature for stimuli generation needed by the test cases using UVM, is categorized under two main groups. First group is a manual based approach, using UVM sequence component [3]. Second group is a tool based approach, which requires 3rd party tool to handle traffic generation to the UVM environment.

In UVM the "*uvm_sequence*" is the main component responsible for the transaction generation, the implementation model types for UVM sequences needed for transaction generation, is defined in [5], [6]. Nine main types are defined in these papers, and they are summarized as follows. The basic sequence is the first of all which is extended from "*uvm_sequence*" and parameterized with interface specific transaction, where user randomize the transaction and send it to sequencer. The next three types are shaped, configurable and resourced sequences, they are as the basic sequence but uses variables to create

2332-5674/16 $31.00 © 2016 IEEE
DOI 10.1109/MTV.2016.14

meaningful transactions, variables can be defined in the sequence or grouped in a separate configuration class, or called from the database as a shared resource class respectively. The fifth type is the library sequence, where different sequences grouped in a library package each sequence generates specific stimulus, then user selects from the test cases the needed sequence. Interface with C is the sixth sequence type, which uses SV PLI to call algorithms defined in C-language to create more advanced transactions.

Seventh type is the virtual sequence starts one or several sequences each generates its own transaction. Hierarchical sequence is the eighth type, and it uses several sequence classes stacked in hierarchical order, where the bottom generates basic transaction, and going up in hierarchy a more abstracted transaction is generated and lower sequence is called to start, as an example a lower letter sequence then word sequence then sentence sequence to generate meaningful sentences.

Lastly the ninth type is layering sequence which is applicable when a translation from one transaction sequence item form to another item form is required. An example of this would be converting an "*uvm_tlm_generic_payload*" item to a bus specific "*sequence_item*". The proposed scenario based testing library, combines the usage of shaped and configurable sequence, together with hierarchy sequence to manage the generation of parallel stimuli.

Tool based scenario creation methods promise to handle test cases development by eliminating the need of fully understand of the environment components, by only being familiar with the scenario library configurable options to create high traffic test cases. Examples are Treck-UVM [7] and test-IP [8]. Treck-UVM tool Implements the handling of scenario creation for several protocol interface type, and with optional transaction synchronization and scheduling, while Test-IP tool allows users to create test scenario graphs and the tool will guarantee coverage closure in return.

Test-IP verification technique [8] investigates the need of eliminating the fully understand of the test environment, where user can add test requirements to a configuration file and a traffic is generated using intelligent test bench automation (iTBA) graph-based method [9].

Test creation simplification and superior functional coverage closure are the main goals of the test-IP approach. the configuration files required by Test-IP specifies the scenario generation targeted by a test, with the support of random variable ranges and special variable sequences, this configuration files are read by iTBA specific UVM sequence for the transaction generation. The gains from using these tools are handling transaction traffic creation generation graphically to facilitate test creation and manage parallel stimuli generation (for Treck-UVM). While the main drawbacks are using 3rd party tools which increases the overhead knowledge required to build environment, these tools lack reusability proving as well.

The contributions of the proposed scenario based test creation methodology are:

1- Two new type of UVM sequence is introduced ,
2- Automatic parallel stimuli synchronization.
3- Methodology is SV native, and doesn't require 3rd party tool.

III. THE PROPOSED SCENARIO BASED TESTING METHODOLOGY

A. The proposed methodology

The proposed scenario based testing methodology consists of three main pillars, that constitute test parallel stimuli generation capability for a multi-bus protocol interfaces.

The first pillar is the single-stream scenario, where the scenario is defined as a randomly constrained sequence of transactions which is pre-defined in the test bench. Test cases would use several pre-defined scenarios from the scenario library, and furtherly constrain for a particular functionality.

The second pillar is the ability to harmonize parallel running scenarios. Typical environments require generation of several stimuli in parallel for the design protocol interfaces. Parallel running stimuli can be of the same type of transaction or different transaction format. This pillar defines synchronization phase to be added by the test case and controls the synchronization between the parallel transaction sequences automatically. Figure 1 is an example for a design with three protocols ports, the environment is injecting parallel stimuli for each interface, and synchronization barrier phase A and phase B are defined. For phase A, $Seq2_0$ is a release transaction thread, while $Seq1_0$ and $Seq0_0$ are catcher transaction threads waiting for the barrier to get released by $Seq2_0$. Similarly for phase B, this is considered as a second synchronization phase node in the parallel scenarios run by the environment.

The third pillar is a composite sequence structure proposed as a framework for the parallel hierarchical sequence scenarios connection. Figure 2 defines the proposed composite structure. The leaf sequence is the lower sequence that generates the final sequence items (transactions) which is passed to the agent's sequencer. Node sequences are an abstracted version of a lower node or leaf sequence that configure and start the lower sequence. Root sequence is a virtual sequence that starts node and/or leaf sequences and schedules them to run in parallel. Going up in hierarchy, more abstracted attributes are defined, and tunable variables are added to allow for the generation of scenarios.

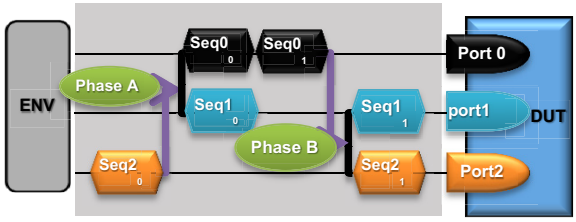

Figure 1, Multi-Stream scenario stimuli with synchronization phases

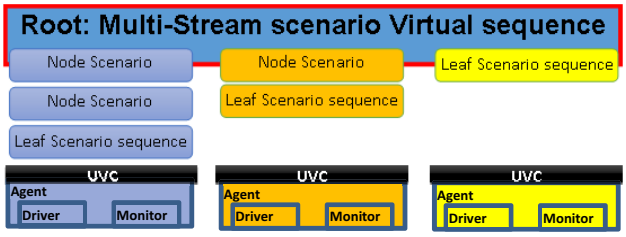

Figure 2, Composite sequence structure

B. The implemented library

The implemented library consists of three new classes that extend UVM base classes,

1) The single stream scenario class

uvm_ss_scenario_sequence clones the behavior of the *vmm_ss_scenario* base class [10] with some other added scenario synchronization functionalities. *uvm_ss_scenario_sequence* is extended from the *uvm_sequence* class, it is a basic harness where user extends and define its transaction specific scenarios using SV constrains to shape the transaction stream to be generated. The scenario is saved by a variable holds scenario name, where the user from the test can randomize the class and constrain the scenario selected.

2) The multi stream scenario sequence

uvm_ms_scenario_virtual_sequence is extended from the *uvm_sequence* class; it is expected to be a virtual sequence that holds the single stream scenarios and the related synchronization.

3) The multi-stream scenario event barrier class

ms_scenario_barrier class is a customized *uvm_barrier* class to work with single stream scenario sequences, and user use to schedule hold (catcher) or un-hold (releaser) threads at specific scenario transaction sequences.

C. Examples

In the below *my_alu_scenarios* sequence class, a scenario of ALU transaction example is defined using the single-stream scenario class from the proposed library. Scenario creation followed three steps, 1. Declare scenario integer, 2. Add scenario constraint, 3. Define scenario API.

```
class my_alu_scenarios extends uvm_ss_scenario_sequence
#(alu_txn) ;
int simple;                  // Declare scenario integer
constraint simple_scenario_c{
   if(scenario_kind == simple) {      // Add scenario constraint
     length  inside {4};
     foreach (items[n]) {
       items[n].opcode inside {ADD, SUB, MUL, DIV};
       items[n].opcode != items[n+1].opcode;
     }
function new (string name = "my_scenario_sequence");
   // Define scenario API
   simple = define_scenario("First scenario", 10);
endfunction
endclass
```

It is clear that a sequence transaction scenario pre-defined as in the example above is much more reusable, extendable and readable than being defined as procedural code in the body task of the default *uvm_sequence*.

Next example will be on the usage of Multi-stream scenario in the creation of a complete parallel traffic generation and phase synchronization for the example shown in example figure.

```
class my_ms_scenario extends
uvm_ms_scenario_virtual_sequence;
…
 // Selecting the scenarios.
 if(!(Seq0.randomize with { scenario_kind == simple ;}))
 if(!(Seq1.randomize with { scenario_kind == simple1;}))
 if(!(Seq2.randomize with { scenario_kind == simple2;}))
// Define Phase A
 PhaseA.add_barrier(Seq0.get_name, 0, 0); // catcher
 PhaseA.add_barrier(Seq1.get_name, 0, 0); // catcher
 PhaseA.add_barrier(Seq2.get_name, 0, 1); // releaser
 Seq0.add_seq_barrier(PhaseA);
 Seq1.add_seq_barrier(PhaseA);
 Seq2.add_seq_barrier(PhaseA); …
// Run in parallel
  fork
    Seq0.apply(null);
    Seq1.apply(null);
     Seq2.apply(null);
  Join
endclass
```

Three steps followed, 1. Select the scenario for each port, 2. Define the synchronization phases needed and assign phase class to the scenario classes, 3. Run the scenarios in parallel.

Trying to implement such traffic using a conventional manual method would require several procedural code to be defined for each test to generate a transaction sequence on each port, and consider one transaction from port to get finished before a second transactions on other ports to start. Test reusability, extendibility and readability degrade severely.

IV. CASE STUDIES AND RESULTS

Our previous work [4] reported the usage of the scenario based testing library in the verification of hybrid memory cube IP [10], where the library used to generate parallel scenarios to different bus protocols interfaces (I2C, power pins and I/O packet pins), the gain from applying the composite sequence structure in that IP verification was found to be efficient in terms of code coverage closure, where coverage closure was 3x faster than basic random sequence generation.

This section will demonstrate the usage of the scenario based testing library with a serial flash IP. MX25UM51245G [12] is 512 Mb bits serial nor flash memory with a serial peripheral interface (SPI) and software protocol allowing operation on a simple 3-wrire bus while it is in single I/O mode or 8 wire bus while in octal I/O mode. Bus consists mainly of a clock (SCLK), serial data input (SI) and a serial data output (SO). Serial access to the device is enabled by chip select (CS#) input. The device supports an octal peripheral interface (OPI) mode, with 8 bit I/O, clock (SCLK) and chip selects (CS#).

The single stream scenario library used in the creation of test cases for this IP, where several scenarios are defined

and reused in test cases to cover the stimuli generation needed, Figure 3 shows the count of scenarios used per test case, a peak of 34 scenario used in the registers test, and minimum 8 scenarios used at the basic read write test case, the average count is 18 scenarios per test case. While Figure 4 represents for each scenario how many it is used in the test cases, total is 29 scenario count, and used scenarios varies across test cases, a peak repeated usage of 24 for the write configuration register scenario, and minimum of 1 time for 3 scenarios. The previous results exhibit the reusability level of the scenarios through the test bench, where a repetition average of 11times for each of the 29 scenario across the 12 test case proves the efficient reusability.

Figure 5 is a graphical representation of the command sequence constrained by the basic write stream scenario. The scenario is randomized and constrained to go through any path of the commands sequence shown in the graph. This scenario can be un-wrapped to a 15 different command sequence.

Table 1 summarizes the contribution of the proposed methodology in adding two new sequence single stream scenario sequence and multi-stream scenario sequence, automatic parallel stimuli synchronization, SV native library and methodology that is compiled and simulated along with the test bench and design. Methodology proved also efficient testing capability in term of reusability, test cases count and coverage closure performance.

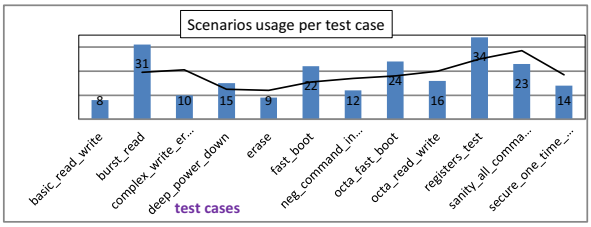

Figure 3, Number of used scenarios per test cases.

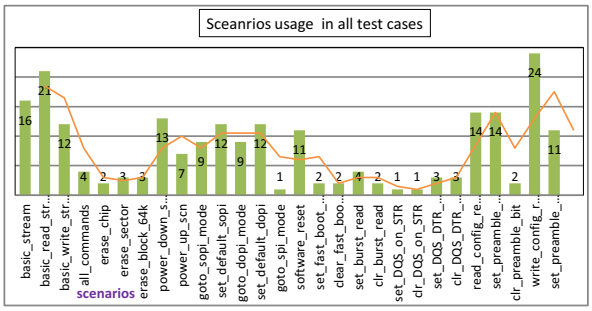

Figure 4, Sceanrios usage across all test cases.

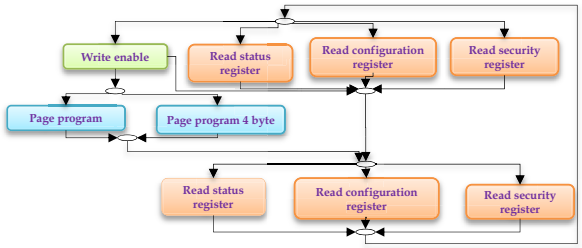

Figure 5, Basic write scenario graph.

Table 1, conventional vs. propsed test case stimuli generation

	Conventional	Proposed
Sequence types	9	Enhanced two types (11)
stimuli synchronization	Manual	Automatic synchronization between Stimuli.
Extra EDA tools	Some	No 3rd party tool is needed
Test case count	-	15x less
Coverage performance		- 70% less
Scenario reusability		- 18 scenarios per test case. 11 usages for each scenario across tests.

V. CONCLUSION AND FUTURE WORK

The proposed scenario based testing methodology achieves reusability with different protocol interface. Moreover, adopting the library in a specific test bench enhances the scenario sequence reusability across test cases. Library eliminates the user need to fully understand the details of the verification environment, as the library abstracts the stimulus generation and archives the needed scenarios to be used in several test cases.

As a future work, an interactive scenario sequence needs to be defined and applied in a real IP verification so that test scenario can be changed as per responses received from design. Moreover, the suggested scenario based testing methodology library will be reused for other protocol interface types, to confirm its reusability. The single stream scenario relies on constrains to define the scenarios, this constrains solving requirement can cause tool inefficiency, further analysis on writing solver friendly scenario constrains is needed, to avoid losing CPU time.

REFERENCES

1. IEEE 1800 System Verilog, HDL LRM.
2. H.Foster "Trends in ASIC/IC Functional Verification", Industry report from Wilson Research Group and Mentor Graphics, 2014.
3. Universal Verification Methodology Class Refrence Mannual, Release 1.2, www.accellera.org (as on Aug, 2014).
4. K. Fathy, K. Salah and R. Guindi, "A proposed methodology to improve UVM-based test generation and coverage closure," *2015 10th International Design & Test Symposium (IDT)*, Amman, 2015, pp. 147-148.
5. M. Peryer, "*Seven Separate Sequence Styles Speed Stimulus Scenarios*", DVCON, 2013.
6. R. Edelman and R. Ardeishar, "*Sequence, Sequence on the WALL, who is the Fairest of them All?*", DVCON, 2013.
7. T. Anderson, 2014, "Enhancing UVM Testbenches with Graph-Based Sceanrio Models" industrial Article, http://www.brekersystems.com/products/trekuvm/.
8. V. Bellippady , S. Haran and J. O'Donnell, "*Using Test-IP Based Verification Techniques in a UVM Environment*", DVCON, 2014.
9. Intelligent Testbench Automation, https://www.mentor.com/products/fv/infact/
10. Synopsys. (2009) VMM 1.2 Documentation
11. Hybrid Memory Cube Specification Version 1.0, Hybrid Memory Cube Consortium, January 2013.
12. Serial NOR Flash, MX25UM51245G, rev 0.04, Macronix, October