Dynamic and Scalable OVM Stimulus for Accelerated Functional Coverage

Michael J Castle
Space and Airborne Systems, Raytheon,
2200 E. Imperial Hwy., El Segundo, CA 90245
Michael_Castle@Raytheon.com

Abstract - The IEEE 1800™ *SystemVerilog* [1][2] is a standard set language extended from Verilog. This language is used for both design and the development of an Open Verification Methodology (OVM) testing environment to allow developers the ability to verify a design and achieve full functional coverage. Importance of this paper is positioned on OVM's ability to dynamically allocate and modify class structures without re-coding the original design to reduce out of phase defects to obtain functional coverage faster through dynamic test vectors. The goal is to identify a set of concepts that take advantage of OVM's dynamic and reusable features, enabling designers to develop additional complex algorithms to achieve goals such as functional coverage and verification. We describe the various reusable OVM components that allow developers to produce code that dynamically creates real-time test vectors to reach full functional coverage faster in contrast to typical random testing. While the fact that random testing enables complete stimulus randomness, it does not enable the ability to reach full functional coverage in a timely manner.

Keywords – *SystemVerilog; OVM; functional verification; Dynamic Programming*

I. INTRODUCTION

This paper presents a method to easily scale and parameterize OVM components to enable complete dynamic stimulus. In addition, it provides basic code illustrating the distinct characteristics to achieve dynamic test stimulus. The example will show the OVM environment rather than the Universal Verification Methodology (UVM) due to the UVM library performing the same functions as OVM for this experiment.

Stimulus going into the design must trigger all events in the device under test (DUT) and the results are to verify the requirements were met. Implementing OVM components such as sequencers enable higher levels of complexity to allow faster achievement towards complete functional coverage through run-time vector manipulation. Typical testbenches are not structured to alter tests in real time, nor exercise the ability to create functions that are scalable and allocate values to parameters in real time (see Figure 1). However, OVM provides this built-in layer of abstraction for development ease.

The goal of this paper is to suggest using many of the OVM constructs embedded in the methodology, as well as a few additional modifications to achieve the desired functional coverage faster. As designers become fluent in the methodology, they can continue to make further modifications to each module to achieve better results.

II. ADVANCED OVM CONCEPTS

To identify some of the concepts to achieve the functional coverage, we present the advanced concepts behind OVM. While presenting the concepts, we shall present the framework of the modules and the modifications to the existing SystemVerilog. The concepts presented are for verification engineers and designers interested in expanding their verification knowledge.

The list below presents the advanced components and terms necessary to create models achieving fast functional coverage to reduce simulation time.

- Class – Similar to C++ classes and objects, these instances contain variables and functions that can be called and reused throughout the environment.
- Sequence – A complex bi-directional stimulus and reusable class that can perform creation and randomization of transactions at runtime.
- Sequencer [3] – The class that synchronizes the transactions between the sequences and the driver.
- Coverage – The frequency or degree to which an event that is recorded in the system has been tested. The data is recorded into Coverage Collectors. Coverage is interpreted by the verification engineer from the requirements document and implemented in the OVM Coverage Scoreboard component.
- Coverage Scoreboard – Analysis component that records the frequency and determines the completeness of testing using Coverage Collectors.
- Agent – Consists of three main class objects: Driver, Sequencer and Monitor. This class object allows engineers the ability to contain a specific protocol into a smaller environment that can be instantiated in multiple locations for reusability.

III. SYSTEMVERILOG FOR VERIFICATION

SystemVerilog provides engineers a more complex environment to further enhance their tests. Typical testbenches are simple, allocating the resources to directed tests and verification via waveform or some form of basic comparison

logic. Since earlier designs were smaller, this method was sufficient for designs. However, as designs grow more complex, this method is not feasible any more. This is due to FPGAs and ASICs having a much larger gate count and more complex functionality.

SystemVerilog deeply integrates Object Oriented Programming (OOP) to create complex reusable programming modules that aid in the development and verification of large devices. Due to the complexity of OVM and OOP components, designers require voluminous stages of planning and management of objects in order to reuse and parameterize the models, where each object can inherit the characteristics and variables from the parent model. Thus setting the OOP coding framework during the initial development phases is crucial to the achievement of advanced verification and future successes.

Unlike typical RTL testbenches, OVM environments with OOP are easily ported and reused amongst various systems, each having minor differences such as variable types, sizes and naming conventions. In some aspects, it may be required to go through several iterations of the same code to achieve the greatest results. With this OVM framework, developers are capable of obtaining highly complex, reusable and effective verification models.

IV. ACHIEVING FUNCTIONAL COVERAGE

Designs are growing, the industry is changing and the need for products to enter the market prior to the competitors is immensely imperative to the success of the company. Deploying bug free products is a necessity to retain existing and prospective customers, whether they are the CEOs or investors. Paul Wilcox [5] notes that finding out of phase defects in the field is detrimental to the company and could cost millions of dollars of damage by adding additional engineering costs and loss of business, thus stressing verifications significance in the design cycle.

Paul Wilcox [5] also states that in the past, verification was performed by the design team once the code was developed. With the designer creating the testbench, the focus of the testing typically focuses on the requirements using directed tests. A. Kumar and C. Kumar [3] explains that if the team is the sole verification engineers, the engineers tend to focus on the requirements designated in the specification thus leaving the interpretation of the specification left to one team rather than two or more. Without a second teams review, the requirements may not be implemented correctly and could leave to eventual system errors.

It is common to perform directed tests, and corner case testing if time is allotted, to verify functionality. In Figure 1, we see that the testbench involves a task list that stimulates the Device Under Test (DUT) to verify the design functionality which may not provide enough stimulus to the design. Mentor Graphics tests using OVM [6] contains stimulus to use SystemVerilog's constrained and unconstrained random stimulus. Although random testing is available, Paul Wilcox [5] states this does not guarantee testing will complete all of the coverage. Constrained random testing may not cover everything since it is more related to directed tests and requires the engineer to become familiar with the design.

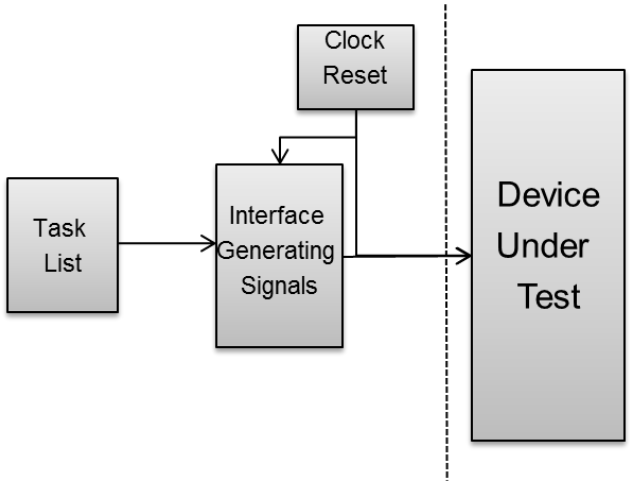

Figure 1. Traditional Verification Diagram. This traditional method is one dimensional and allow for testing only using stimulus without feedback.

Instead, random testing allows the engineer who verifies the DUT to handle corner cases not originally tested by the designer. This allows for test cases that typically lead to recovery logic, a state machine reset or error handling in Intellectual Property (IP) modules.

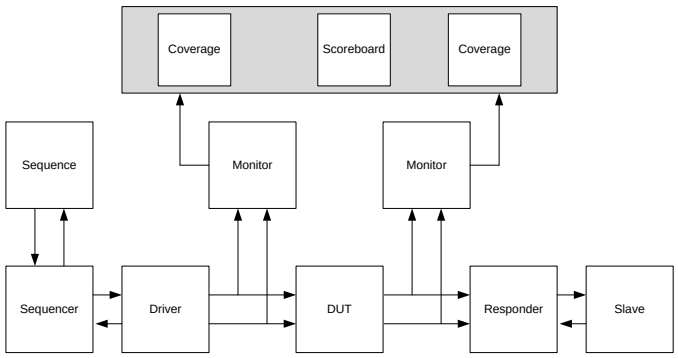

Figure 2. OVM Environment Testbench Diagram. Above is a typical representation of the OVM environment where the sequence receives feedback from the sequencer and driver.

Random and constrained random testing may take several hours to cover all of the scenarios necessary to achieve full functional coverage. However, implementing the sequencer to

used feedback from the driver via a response packet, test times can drop significantly.

To determine whether all of the functionalities are tested, engineers use coverage scoreboards to analyze the results based on reports and coverage collector outputs. This typically requires the verification engineer to have some understanding of the system level requirements.

A. COVERAGE ANALYSIS

Sequences typically are driven unidirectional without feedback to dynamically alter the tests to verify functionality and increase code coverage faster. Several languages such as VHDL and Verilog are capable of creating such dynamic sequences. However, OVM and SystemVerilog utilize built in library functions to perform the same tests.

OVM allows for easy development of transactions between the stimulus and the driver, allowing for the driver to provide feedback to the stimulus to allow for dynamic change in tests. Another feature available in OVM is the ability to have a response from different modules such as the coverage scoreboard.

Figure 3 shows a simple diagram of the coverage scoreboard's return path to the sequence. In many OVM environments, the driver transmits a response packet to the stimulus. Using coverage data over driver data allows the designer to know exactly what operations have been exercised. As shown in Figure 4, driver data does not provide very many details about the operation other than function X was issued to command an operation.

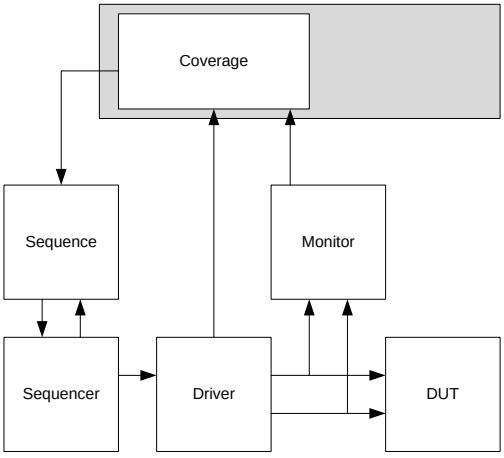

Figure 3. OVM Environment Test bench Diagram. The proposed method allows the sequence to receive feedback from the responder rather than the driver. This allows for the sequence to dynamically change the stimulus in runtime based on what has been covered.

Also, as seen in Figure 4, coverage blocks add far more information that is relevant to the sequence. For instance, say we are issuing commands, in any order, to a FIFO. A verification engineer wants to capture write full and read empty errors. The driver should not be aware of these flags.

Instead, the monitor captures, translates and forwards the data to the coverage scoreboard. From there, the coverage scoreboard information stating how many times the error flag was captured is placed into a packet and sent back to the sequence, where the sequence will appropriately adjust itself to send more writes in order to force the coverage scoreboard to meet its error flag goal.

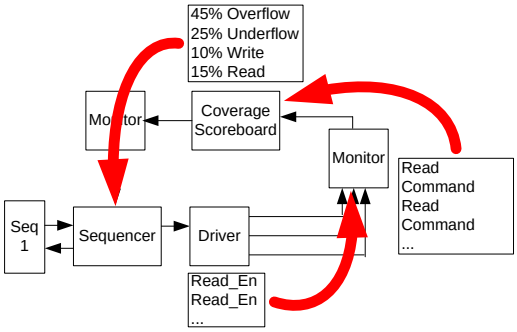

Figure 4. Diagram of the Data Passed between Components. As data progresses from driver to monitor and coverage scoreboard, the data becomes more detailed thus potentially leading to superior and clear-cut stimulus into the system.

Functional coverage data from the coverage scoreboard class object can influence the stimulus drastically, thus allowing the test to complete much faster as seen in the FIFO example. To dynamically change the tests in respect to the functional coverage, verification engineers must account for the return packet and determine what stimulus is required to achieve full functional coverage. Verification engineers must research the specification document to interpret the expected results and set the appropriate actions for the sequence.

V. IMPLEMENTATION

To understand the design's coverage analysis, structural details of the proposed coverage analysis response method are presented. Afterwards, we discuss some aspects of the language and implementation.

A. COVERAGE ANALYSIS RESPONSE

In the proposed design, we consider the built in functions of the OVM libraries to return the data from the coverage scoreboard back to the sequence. Two packets are required for the operation to occur, the first is the packet necessary for the driver and the second is the return data. The two packets are illustrated below with the send_packet and rsp_packet:

```
//Class/Module Name
class ex_sequence0 extends ovm_sequence #(send_packet,rsp_packet );

 //Factory
 `ovm_sequence_utils(ex_sequence0, ex_sequencer)

 //Packets
 send_packet snd;
 rsp_packet rsp;
```

With this modification, the sequence can incorporate the rsp_packet into the sequence algorithm to dynamically adjust the stimulus. In the environment, this allows for two-way communication from the sequencer to the sequence as shown in figure 5. Duolos states [7] that to connect the sequence and sequencer, the macro `ovm_sequence_utils is used to add the sequence to the list belonging to the given sequencer.

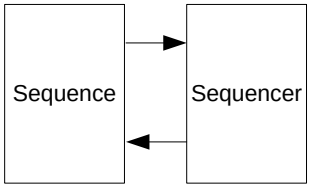

Figure 5. Connection of the Sequence to the Sequencer. This is an automatic connection between the two components.

Next, designers need to add the incoming port from the sequencer with the extension shown below:

```
ex_driver extends ovm_driver #(rcv_packet);
```

This allows for incoming packets from the sequencer. To attach the Sequencer to the driver, a connection in the OVM agent is required to connect the seq_item_export of the sequencer to the driver seq_item_port as shown below.

```
driver.seq_item_port.connect(ex_sequencer.seq_item_export)
```

This allows for transactions from the sequence through the sequencer to the Driver and now allows the flow of data from the sequence to the driver depicted in figure 6.

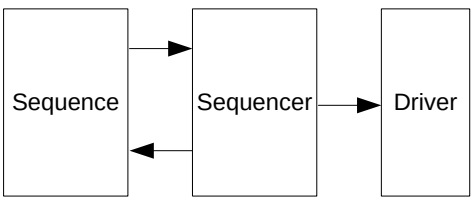

Figure 6. Connection of the Sequence, Sequencer and the Driver. Currently the Sequence can flow data to the driver, but cannot receive data.

To transmit a response packet back to the sequence from the coverage scoreboard, we require several small additions. A Monitor is required to gather data from the low level pins of the virtual interface to interpret the data. Also, in order to hold the data until the function is ready, an optional tlm_analysis_fifo can be used. Next, the Monitor packages and transmits the data through an analysis port as shown below:

```
class fifo_pass_through_monitor extends ovm_monitor  ;
…
  //PORTS/EXPORTS
  ovm_analysis_export #(rsp_packet) monitor_input_export;
  ovm_analysis_port   #(rsp_packet) my_rsp_ap;


  protected tlm_analysis_fifo #(rsp_packet) monitor_pt_in_fifo;
```

The Coverage Scoreboard requires two OVM ports: one to receive data from the Monitor and another to export the statistical data. The output statistical data requires a different packet type since this data format a higher level of abstraction containing integers and floating point values. A connection must be made in the connect function of the Agent. Figure 7 depicts the connection of the Sequence, Sequencer, Driver, Monitor and Coverage Scoreboard class objects.

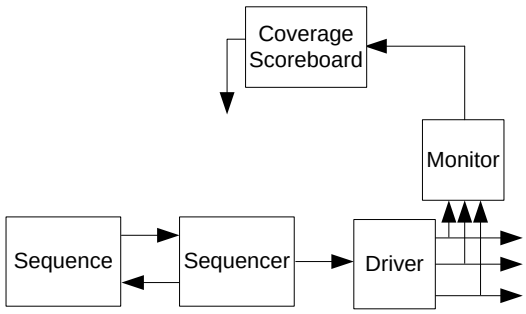

Figure 7. Connection of class objects from the Sequence to the Coverage Scoreboard.

To transmit data from the coverage scoreboard, data must flow through an intermediate passive Monitor. The pass-through Monitor has the built-in response port necessary to transmit the analytical data from the Coverage Scoreboard to the rsp_port of the sequencer. As shown earlier in Figure 4, the Coverage Scoreboard transmits a response packet to the Monitor which is then passed through to the sequencer through the rsp_port. The OVM pass-through Monitor requires a blocking port and an rsp_port.

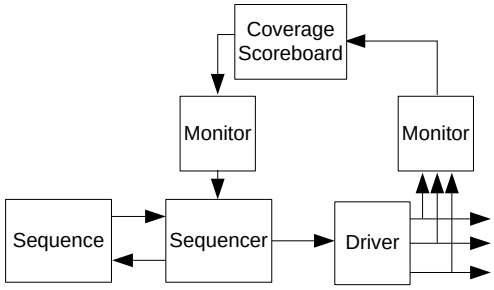

Figure 8. Connection of all the class objects to receive data from the Coverage Scoreboard to the Sequence.

The reason for having the pass-through monitor is due to the limitations of the OVM libraries. According to Mentor's Verification Academy reference manual [6], "the analysis export [is] used by drivers or monitors to send responses to the sequencer." This feature is unavailable to other OVM components. To write data from the monitor back to the sequencer, the Monitor must extend the ovm_driver and declare the response packet that will connect to the rsp_port. Writing data to this port can be performed by the below:

```
rsp_port.write(response_packet)
```

In the response packet, the analytical data can vary depending on the designer's requirements and the data necessary to transmit back to the Sequence.

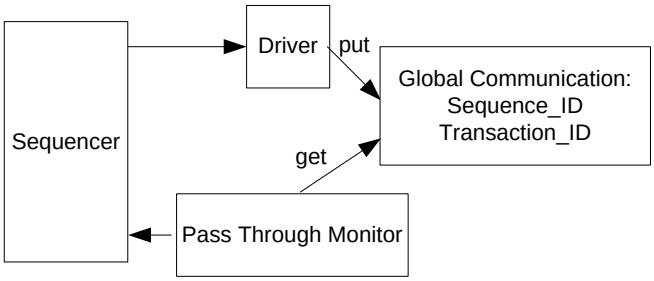

Figure 8. Global Communication Variables. Sequence_ID must be set on the rsp item to match the req item. This allows the message to return back to the proper sequence.

The events for the Sequence will follow standard wait for grant, send and wait for done sequence depicted by figure 9. However, in the middle of this sequence, the sequence_id and transaction_id must be saved and read by the Pass-through Monitor in order to return the rsp_packet to the correct sequence. In figure 8, one method of storing the information is through a global class storing the variables. Using this technique, designers can perform a simple handshaking to transmit and receive packets in the Sequence, as well as return the rsp_packet to the correct sequence.

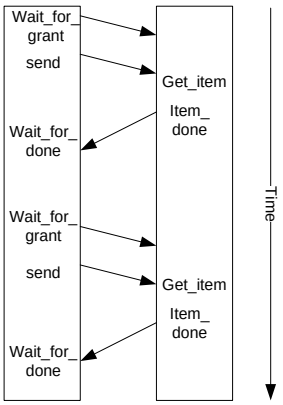

Figure 9. Timeline of events for the Sequence. The Sequence will continue to perform the same operations whether it is the feedback from the Driver or pass-through Monitor.

From this foundation, the designer must create software to properly transmit and receive data correctly between each class object while following the OVM standards. The Monitor between the Driver and the FIFO continues to grab data; therefore the designer must consider how much data to send to the Coverage Scoreboard. Also, the designer must consider when to transmit data from the Coverage Scoreboard to the Monitor.

The Coverage Scoreboard is continuously sampling the packets from the Monitor, thus a flag signifying the end of an action is required. This end of action flag can then trigger a transmission packet to the pass-through Monitor and allow the Sequence to continue from the wait for done function.

VI. EXPERIMENTS

For the two proposed methods, the tests are performed on a standard FIFO with first-word-fall-through, a read and write clock for various rates and asynchronous resets using the QuestaSim Simulator. A full OVM environment is setup to monitor the stimulus into the FIFO, as well as a monitor the outputs from the FIFO as shown in figure 10.

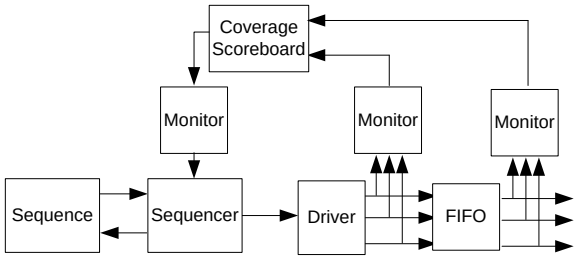

Figure 10. OVM Test Environment with Coverage Scoreboard Analysis Data Feedback.

A Coverage Scoreboard is implemented to capture at least 200 successful reads and writes, 200 write full and read empty flags and 200 overflow and underflow flags (see Table 1). Each error flag can be captured multiple times on a given instruction.

TABLE 1. Summary of Flags for Code Coverage.

Test	Flag Count
Read Commands Issued	200
Read Not Empty Error	200
Read Program Not Empty Error	200
Read Underflow Error	200
Write Commands Issued	200
Write Not Full Error	200
Write Program Not Full	200
Write Overflow Error	200

Two tests scenarios are used for comparison depicted in table 2, the first consisting of using only random stimulus and the second using random stimulus with responses from the coverage scoreboard.

TABLE 2. Summary of Stimulus and Clock Rates.

Test	Clock Rate #1	Clock Rate #2	Stimulus
1	1000 MHz	1000 MHz	Random
2	1000 MHz	1000 MHz	Coverage Feedback

The stimulus packet contains a data array, a command register for read or write and a burst size. Stimulus for both methods will either choose from read or write; issuing various sizes of read or write burst commands.

For this test, the coverage analysis data feedback method will issue random stimulus as its driving input with little logic to determine the coverage bin containing the least frequency counts. The coverage analysis data feedback method will use the least frequency counts and issue a random data burst of read or write commands, while the complete random stimulus will issue read or write data bursts at random without order or constraints.

A. ANALYSIS

Looking at the results in table 3, the tests concluded a savings of more than 5000 cycles. The tests completed the coverage requirements stated in table 1 in a short time. The actual time for the simulation resulted in the following simulation seconds:

TABLE 3. Summary of Results.

Test	Simulation Time	Coverage (%)
1	41626ns	100
2	25918ns	100

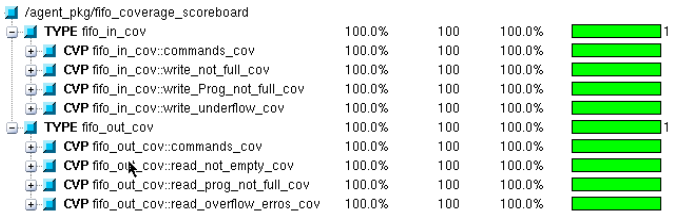

Figure 4. Coverage Group Report. Complete Coverage with a goal of 100% was achieved at 41626 nanoseconds for complete random stimulus, while Complete Coverage for the Coverage Feedback method was 25918 nanoseconds.

Shown in figure 4 is the complete coverage of the Coverage Scoreboard. Both tests were able to complete the minimum 200 count requirements. Due to the burst size not containing any constraints, each flag captured received well above 200 due to the packets continuing to issue the random burst size.

TABLE 4. Summary of Actual Flags Captured for Code Coverage.

Test	Random	Coverage Feedback
Read Commands Issued	4544	1530
Read Not Empty Error	3402	388
Read Program Not Empty Error	3402	388
Read Underflow Error	3395	383
Write Commands Issued	5687	4925
Write Full Error	1994	1232
Write Program Full	5407	4645
Write Overflow Error	1993	1231

The actual flag counts captured were also significantly less using the coverage feedback method. This is due to the less redundancy of the random stimulus entering the same commands several times. The amount of errors captured was reduced significantly, whereby meeting the Coverage Scoreboard requirements early in the simulation.

VII. CONCLUSIONS

Examining the design of the two methods, the introduction of the coverage analysis for response showed that the amount of simulation time can be reduced dramatically. Our initial design was fairly simple, checking the coverage of a few flags and acting on the coverage with the least frequency counts.

Both coverage feedback and random stimulus completed the tests in a reasonable time, however with more modifications the design can easily save hundreds more cycles by adding to the code complexity. The coverage feedback currently determines whether the entire read or write coverage bins have been satisfied. Instead, more logic could have been implemented to force specific error flags that would cause read empty errors, write full errors, etc.

In the future, our tests will compare the effects of increasing the logic to force specific functions using coverage feedback; as well as another test to compare coverage feedback against driver feedback.

REFERENCES

[1] IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language, IEEE *Std* 1800™-2005, IEEE Computer Society, 2005.

[2] S. Sutherland, S. Davidmann and P. Flake, SystemVerilog for Design, Norwell, MA: Kluwer Academic Publishers, 2004.

[3] J. Aynsley, "Easier UVM for Functional Verification by Mainstream users," Duolous, Ringwood, U.K.

[4] A. Kumar and C. Kumar, "Functional Coverage Analysis of OVM Based Verification of H.264 CAVLD Slice Header Decoder," Jharkhand, India,

[5] P. Wilcox, Professional Verification: A Guide to Advanced Functional Verification, Norwell, MA: Kluwer Academic Publishers, 2004.

[6] Mentor Graphics. Verification Academy: UVM / OVM Verification Methodology. Mentor Graphics., OR. [Online]. Available: http://verificationacademy.com/verification-methodology

[7] Duolos. Getting Started with OVM. [Online]. Available: http://www.doulos.com/knowhow/sysverilog/ovm/tutorial_2/