Dilemma of opting one out of many methodologies and mantra of going back to simple basics

By Swapnil Sapre, hardware engineer in the R&D Services unit of MindTree Ltd. (Bangalore, India)

WHY THIS ARTICLE?

One of SystemVerilog's noticeable features is that it is basically a "design language" that has been extended with verification capabilities. This might be an advantage or not, depending on who you're asking, but obviously, if you only want to use a limited number of verification capabilities to gradually extend your existing Verilog testbench, SystemVerilog might be a good solution.

This is a bottom-up approach where you want to retain your old verilog code (typically transactors) and add stuff like randomization and maybe coverage on top. It is normally recommended, though, that you seriously consider a top-down approach (read: lose your old testbench), because a top-down approach provides a comprehensive solution to the verification problem (courtesy VMM, VMM, AVM and now UVM).

But in today's scenario in which there are too many methodologies to choose from and still we are not sure if we are going to use them for long or again there is going to be a migration of environments, and how are EDA vendors going to charge and support us with compatibility and interoperability libraries across simulators- we're going to discuss about the former approach which is a more pragmatic one in certain cases.

This bottom-up approach gives more confidence to individuals at the concept level and gives more room for experimentation to make use of OOD concepts rather than memorizing the classes that are available in methodology libraries and manuals.

This paper discusses Random constraint-based verification and explains how random verification can complement the directed verification for the generic designs.

This also describes how to create a simple and robust constrained random-based verification environment step by step with the help of SystemVerilog.

The Constrained Random Techniques (CRT) can be effectively used for verifying large complex designs. As CRT can automatically generate a large number of test cases, it can hit corner cases faster and help in reaching conditions that would normally not be easily reached with traditional methods. These features are built over and above an already existing legacy Verilog environment.

Random verification for generic designs is implemented by Transaction-based Models or Bus Functional Models.

1.0 INTRODUCTION

"Reuse" is a term that is frequently associated with verification productivity. When faced with writing a verification environment from scratch, or modifying an existing one, the choice will often be to stick with what is familiar and already in existence.

"Methodology" lays a foundation for a robust verification environment which is capable of handling complex verification needs and speeding up the verification process.

When a verification environment is needed for a new design, or for a design revision with significant changes, it is important to objectively look at the shortcomings of the existing verification environment and expected productivity gain with the new methodology and determine the best solution.
In most cases, we need to find an optimum balance between re-usability of our legacy Verilog environment and the resource utilization along with limited timelines in adopting the new methodology. This can be accomplished by reusing the knowledge/legacy code from an earlier project along with an upgrade to a new methodology provided with the verification language that is SystemVerilog.

There are already a few verification methodologies available such as VMM and OVM which help in building a robust verification flow. But keeping in our limited resources and stringent timelines, we may have to focus on implementing a simpler flow based on Constraint Random Techniques (CRT), which helps in generating the interested test scenarios automatically. This all is performed and implemented with in-built features available with SystemVerilog.

This document demonstrates the introduction of Constraint Random Verification with SystemVerilog while re-using the legacy Verilog verification environment (keeping what we knew best).

2.0 DESIGN UNDER TEST

The Figure below shows the top level view of our design under test.
3.0 DIRECTED VERIFICATION OF DUT-LEGACY/EXISTING ENVIRONMENT

Figure below shows the legacy Verilog-based verification environment which was earlier used to verify the functionality of the platform.

The legacy verification suite of scenarios consisted of a group of ARM-based assembly patterns and Verilog-based BFM scenarios.

The BFM-based Verilog patterns are written for verifying the integration and other checks which were not possible through assembly patterns. The BFM in Verilog was replacing the CPU-1, CPU-2 and was generating the manually requested transactions.

This traditional approach of verifying the designs by writing the Verilog/VHDL testbench leads the designers to completely rely on developing a directed environment and handwritten directed test cases. These directed tests provide explicit stimulus to the design inputs, run the design in simulation, and check the behavior of the design against expected results. This approach may provide adequate results for small, simple designs but it is still a manual and somewhat error-prone method. In addition, directed tests were not able to catch obscure defects due to features that nobody thought of. Moreover, these traditional methods have very limited and cumbersome random capability.

3.1 WHY ADVANCED VERIFICATION ENVIRONMENTS?

With increase in complexity and size of design, there is higher and higher demand on exhaustive functional verification. These demands are necessitating the development of new verification technologies, such as, constrained random verification, scoreboarding and functional coverage, to achieve exhaustive functional verification goal.
These development methods for reusable verification environment are much easier and helpful in constraining verification to find out the corner cases and hidden bugs which are left undetected with conventional directed approach.

### 4.0 CONSTRAINT RANDOM-BASED VERIFICATION

Now, before starting the implementation of a Constraint Random Verification environment, there were few points of consideration.

**Language**

SystemVerilog would be the first choice to be used since it is an IEEE standard as well as easy to learn, for those who are already familiar with Verilog. It provides some additional constructs for the randomization implementation and Object Oriented techniques for improving the Verification environment.

**Approach**

**REUSING OR CREATING THE TESTBENCH FROM SCRATCH**

We implement the “constraint driven coverage-based randomization” in SystemVerilog by reusing the Verilog-based Transactor Tasks (Bus Functional Models) and utilizing SystemVerilog constructs as discussed below.
The above Figure shows the layers added in the existing verification environment to implement the Constraint Random Verification environment.

The flow for preparing the test plan remains the same as before for the directed testing except that, now the focus is on implementing the random transactions and data streams which are valid for the DUT. The commands are random to the extent that they cover the corner case scenarios that cannot be thought of during the directed verification. Cross-coverage of these transaction types is then performed, in order to ensure that all the combinations for op-codes and error conditions are exercised.

5.0 BUILDING BLOCKS

5.1 TRANSACTION-BASED STREAM (PACKET) GENERATOR
The Transactor generates high-level transactions like read/write with certain burst/size on some PORT. The term transactor is used to identify components of the verification environment that interface between two levels of abstractions for any transaction.

```
task XFR; // Transactor task for unit AHB packet generation

input [3:0] hrqst; // assert hrqst or not
input hwrite; //
input [2:0] hresp; // expected hresp behavior
input [1:0] htrans; //
input hlock; //
input [2:0] hburst; //
input [2:0] hsize; //
input hunalign; // hunalign
input [7:0] hbstrb; // hbstrb
input [31:0] haddr; // haddr
input [DMSB:0] hdata; // see below
input [DMSB:0] hmask; // AND w/ actual/expected data before comparing
input [5:0] hprot; //
input [5:0] hsideband; // hsideband
input [3:0] hmaster; // alternate master number
input [3:0] slot; //
input [80*8:1] comment; //

endtask;

//XFR(hrqst,hwrite,{xhresp2,1'b0,xhresp0},htrans,hlock,hburst,hsize,hunalign,hbstrb,haddr,hdata,hmask,k,hprot,6'h0,hmaster,slot,comment);

Code snippet for the packet generator
```

### 5.2 FILTER/CONSTRAINT BLOCK

Purely random test generations are not very useful because of the following two reasons:

1. Generated scenarios may violate the assumptions, under which the design was constructed.
2. Many of the scenarios may not be interesting, thus wasting valuable simulation time.
Hence, the random generators should be constrained to generate the required stimulus sequences. Constraining the generators may involve defining sequences of data, but it also may involve coordinating multiple independent data streams onto a single physical channel/port or parallel channels/ports, each stream itself being made up of data sequence patterns.

The ability to constrain the generated data to create detailed stimulus scenarios tends to require more complex randomization process. It becomes more efficient to take few complex stimulus sequences as directed stimulus, and leave the bulk of the data generation to a simple randomization process.

This Filter/Constraint block generates the valid AHB transactions and allowed instructions for the platform. Below is the snippet of the code implementation of the constraint block.

```plaintext
class MainClass;

    rand bit [31:0] Address;
    rand bit [7:0] Strobe;
    rand bit [63:0] Data;

endclass

class Constraints_Block1 extends MainClass;

    constraint C_M00 { (Address [3:0])%4 == 0 ;}
    constraint C_M01 { Address [27:4] == 24'h000000 ;}
    constraint C_M05 { Address [31:28] == 4'h8 ;}
    constraint C_M03 { Address [2] == 1 -> Strobe == 8'hf0 ;}

```

constraint C_M04 {Address[2]==0 -> Strobe==8'h0f;}

endclass
Constraints_Block1 Block1 = new();

class Constraints_Block2 extends MainClass;

constraint C_M00 {(Address[3:0])%4==0 ;}
constraint C_M01 {Address [25:4]==24'h000000 ;}
constraint C_M02 {Address[27:26]==4'h1 ;}
constraint C_M05 {Address[31:28]==4'h8 ;}
constraint C_M03 {Address[2]==1 -> Strobe==8'hf0;}
constraint C_M04 {Address[2]==0 -> Strobe==8'h0f;}

endclass
Constraints_Block2 Block2 = new();

class Constraints_Block3 extends MainClass;

constraint C_M00 {(Address[3:0])%4==0 ;}
constraint C_M01 {Address [25:4]==24'h000000 ;}
constraint C_M02 {Address[27:26]==4'h2 ;}
constraint C_M05 {Address[31:28]==4'h8 ;}
constraint C_M03 {Address[2]==1 -> Strobe==8'hf0;}
constraint C_M04 {Address[2]==0 -> Strobe==8'h0f;}

endclass
Constraints_Block3 Block3 = new();

Code snippet for the constraint block

5.3 COMMAND DRIVER BLOCK

This block generates transactions, either individually or in streams. Individual meaning unit AHB packet and stream meaning multiple AHB packets for different transactor interfaces. Note that each of
these commands may be derived several times or in several flavors till the Functional Coverage reaches 100.

```verbatim

task Command2_BLOCK2_AIPS();

begin

$display("******COMMAND 2 SELECTED*******");
->event2;

cmd2=cmd2+1;

XFR( `XFER, `WR, `OK, `NSEQ, `NLCK, `SNGL, `WORD, `AL, BLOCK2.Strobe, BLOCK2.Address, BLOCK2.Data, 64'hFFFFFFFFFFFFFFFF, `nnnnPD, 6'h00, `MST1, `SLT0, "Setup: 0" );

XFR( `XFER, `WR, `OK, `NSEQ, `NLCK, `SNGL, `WORD, `AL, AIPS.Strobe, AIPS.Address, AIPS.Data, 64'hFFFFFFFFFFFFFFFF, `nnnnPD, 6'h00, `MST1, `SLT0, "Setup: 0" );

endtask

Code snippet for the command driver block
```

The Command (Cn) is a combination of 1 or more number of unit bus packets (Pn). Each packet (Pn) is targeted for different transactor interface.
5.4 SEQUENCER

Sequencer throws the command or a set of commands, to the DUT, depending on the weight or the probability of the scenario (command) at that point of time. Feedback of the Functional Coverage is used to determine the probability of the n\textsuperscript{th} scenario and thus the output of the sequencer.

```
randcase
  100-L1: Command1_BLOCK1_AIPS();
  100-L2: Command2_BLOCK2_AIPS();
  100-AI: Command3_AIPS();
endcase
```

```
L1=cvr.cross_cover.LL1.get_inst_coverage();

L2=cvr1.cross_cover.LL2.get_inst_coverage();
```
A1= cvr2.cross_cover.A01.get_inst_coverage();

Code snippet for the sequencer block

The below diagram shows, how a Sequencer generates the command considering its current weight which is initialized to ‘0’ at the beginning.

Where—

F.C.\(_{nth}\) - Functional Coverage of the \(n\)th behavior (set).

Weight - Probability of selecting \(n\)th behavior (set).

Scenario - Set of valid random instructions (BFM)

Sequencer - Engine that triggers behaviors (set) based on its current weight.

Sequencer methodology overview

5.5 FUNCTIONAL COVERAGE BLOCK.
To ensure that we hit every possible cross-coverage point it is required that we achieve a high functional coverage of the DUT. Also, any areas that were initially missed during random testing are easily highlighted by the functional coverage results.

The functional verification requirements are translated into a functional coverage model to automatically track the progress of the verification project. A functional coverage model is implemented using a combination of covergroup. The choice depends on the nature of the available data sampling interface and the complexity of the coverage points.

Functional coverage is the primary director of the Verification strategy. It determines how well the test bench is fulfilling your verification objectives and measures the thoroughness of the verification process. A functional coverage model is composed of several functional coverage groups. The bulk of the functional coverage model for a particular design under verification will be implemented as a functional aspect of the verification environment.

The output of the Functional Coverage block is given as a feedback to the Sequencer which in turn decides the selection of the n° scenario.

The following points are kept in consideration for developing the coverage groups:-

• Coverage groups for the stimulus generated by the Generator.
• Coverage groups for the stimulus driven onto DUT.
• Coverage groups for the response received from the DUT.

class coverage_for_BLOCK2;
    covergroup cross_cover @(posedge testbench.arm_clk);
    option.per_instance =1;
    type_option.goal = 100;
    L_20:coverpoint BLOCK2.Address
{  
    bins Addr_cover_value[] =  
        {32'h88000000,32'h88000004,32'h88000008,32'h8800000C};  
    // illegal_bins bad = {32'h88000000};  
}

L_21:coverpoint BLOCK2.Strobe  
{
    bins Strobe_cover_value[] = {8'h0f,8'hf0};
}

LL2:cross L_20,L_21;

endgroup

function new();

cross_cover= new;

defunction : new
endclass

Code snippet for a cover group

5.6 RESPONSE AND PROTOCOL CHECKER (ASSERT PROPERTIES)

Assertions constantly check the DUT for correctness. These look at external and internal signals. The testbench uses the results of these assertions to see if the DUT has responded correctly.

In directed tests, the response can be hardcoded in parallel with the stimulus. Thus, it can be implemented in a more distributed fashion, in the same program that implements the stimulus. However, it is better to treat the response checking as an independent function.

By separating the checking from the stimulus, all symptoms of failures can be verified any time.

6.0 CONCLUSION

Keeping in view the limited human resource and the stringent project deadlines, developing an object-oriented verification environment in SystemVerilog over the existing environment felt advisable. Inclusion of the Constraint Random Verification significantly reduces the effort and time required to verify the complex behaviors. Along with reusability, SystemVerilog usage also can be implemented for developing the testbench from scratch with the layered testbench approach as discussed.

The experience using SystemVerilog so far has provided us with an environment that is:
• Maintainable – The Common look and feel between related class types make it easy for team members to float from one functional area to another. The code is very modular with well defined ways to communicate between transactors.

• Controllable – The modular approach allow us to be more precise in determining expected values, thus minimizing false fails.

• Reusable – Core-level checkers, classes, tasks can be reused for system-level verification. In addition, the SystemVerilog skills developed on this project can be used in any future verification project that uses a high level verification language (HVL).

7.0 RECOMMENDATIONS

Adopted methodology provided a robust verification architecture that produces more modular code with a higher degree of reusability. Code produced for one portion of the project can be used in other environments.

8.0 REFERENCES

∞ http://www.inno-logic.com/education-systemverilog-methodology.htm
∞ http://verificationguild.com/modules.php
∞ ARM Synopsys Verification Methodology Manual (VMM) for SystemVerilog
∞ http://www.eda.org/sv
∞ http://www.eda.org/sv-ieee 1800

About the author:

Swapnil Sapre (swapnil_sapre@mindtree.com)

Swapnil Sapre is a guest blogger on Think Verification. He is a Module Lead at MindTree and has 7+ years of experience in verification. He has been providing consultation to various organizations through his expertise in SystemVerilog, Verilog, and VHDL. He also has many other areas of interest such as astrophysics, advertising and politics. Here's Swapnil's interesting paper where he shows how SystemVerilog can be adopted to enhance legacy test benches with constrained random capabilities to verify an ARM SoC system.