2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

OSVVM: Advanced Verification for VHDL

Jim Lewis, SynthWorks VHDL Training, Tigard, OR, USA (*jim@synthworks.com*)

***Abstract*— To meet current product development schedules, we need to spend less time writing tests and finish design verification sooner. To accomplish this, modern verification methodologies have added randomization and functional coverage. Open Source VHDL Verification Methodology (OSVVM) is a set of VHDL packages that simplify implementation of functional coverage and randomization. OSVVM uses these packages to create an Intelligent Coverage verification methodology that is a step ahead of other verification methodologies, such as SystemVerilog's UVM.**

Keywords—VHDL; Functoinal Coverage ; Constrained Random ; Advanced Verification ; OSVVM

I. Introduction

Open Source VHDL Verification Methodology (OSVVM) provides package based functional coverage and randomization utilities that layer on top of your transaction level modeling (tlm) based VHDL testbench. Using these you can create either basic Constrained Random tests or more advanced Intelligent Coverage based random tests. This simplified approach allows you to utilize advanced randomization techniques when you need them and easily mix in directed, algorithmic, and file-based test generation techniques. Best of all, OSVVM is free and works in most VHDL simulators.

II. Randomization Basics

A. Why Randomize?

The key benefit of randomization is generating realistic stimulus in a timely fashion (to code). Randomization tends to be ideal for generating large variety of similar items, such as modes, sequences, processor instructions, and network packets.

The key point here is generating realistic stimulus. Figure 1 shows the number of words in a FIFO viewed as an analog waveform for a directed test. The output "diffcount" is the calculated number of words in the FIFO. The high points represent FIFO full and the low points are FIFO empty. Correlating these values with the directed test stimulus, we can see that we have filled the FIFO, emptied the FIFO, done simultaneous reads and writes, attempted FIFO write while the FIFO is full, and attempted a FIFO read while the FIFO is empty. In short, we have tested all of the items in our test plan.

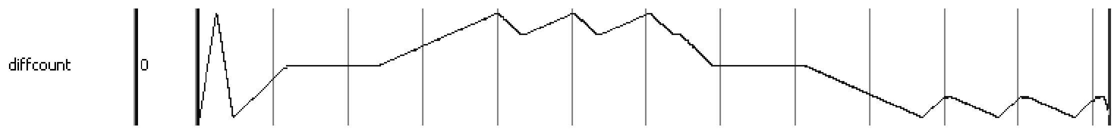

Figure 1: Directed test of FIFO

Figure 2 shows the number of words in a FIFO viewed as an analog waveform for a constrained random test. With a random test, we can no longer use the test conditions to "assume" that we have attempted a FIFO read while the FIFO is empty, instead we must measure these conditions. The signal ReadEmptyCovCnt measures this. The signal WriteFullCovCnt counts the number of times a FIFO write was attempted while the FIFO was full. Using these signals, we can show that all of the items in our test plan are covered.

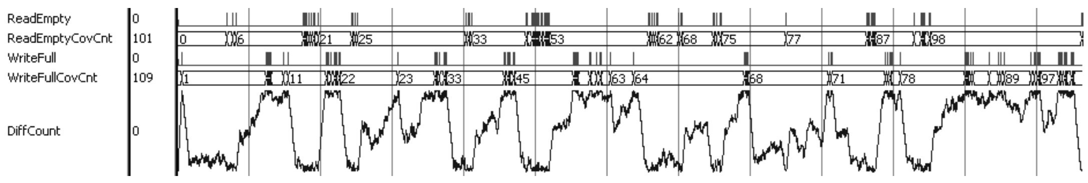

Figure 2: Constrained random test of FIFO

Reflecting back on Figures 1 and 2, one can see there is a significant difference in the quality of the tests. While the directed test hits all of the required points, the constrained random test wanders the space more and is more likely to expose unexpected bugs in the design.

B. Basic Randomization

The package RandomPkg contains randomization functions. One function randomly selects a value within a range of values, such as 0 to 15 as shown in the assignment to Data1 below. A variation of this, shown in the assignment to Data2, randomly selects a value within a range (here 0 to 15) except it excludes values (here 5 and 11).

```
Data1 := RV.RandInt(Min => 0, Max => 15) ;
Data2 := RV.RandInt(0, 15, (5,11) );  -- except 5 & 11
```

Another function randomly selects values within a set of values, such as (1, 2, 3, 5, 7, 11) as shown in the assignment to Data3 below.

```
Data3 := RV.RandInt( (1,2,3,5,7,11) );
```

Some problems, such as weighting for stimulus generation percentages, are best represented using a discrete, weighted distribution. The function DistInt uses an integer_vector input to specify randomization weights. The input to DistInt is an integer_vector of weights. The return value is the index of the selected weight. For a literal value, it will return a value between 0 to N-1, where N is the number of weights specified. As a result, in the assignment to Data4, the likelihood of a 0 is 70%, 1 is 20% and 2 is 10%.

```
Data4 := RV.DistInt(  ((7, 2, 1)) ;
```

RandomPkg also supports similar functions that return types unsigned, signed, std_logic_vector, time, real, integer_vector (for sets), real_vector, and time_vector. For more information see the RandomPkg_user_guide.

By itself, these functions are not constrained random, they simply generate random values.

C. Constrained Random

Constrained random tests are generated by using the randomization code patterns. The following code segment uses the DistInt function to randomly dispatch transactions to a UART.

```
StimGen : while TestActive loop      -- Repeat until done
 case RV.DistInt( (70, 11, 11, 6, 2) ) is
  when 0  =>     -- Nominal case   -- 70%
   TestMode := UARTTB_NO_ERROR ;
   Data     := RV.RandSlv(0, 255, Data'length) ;
   Idle     := RV.DistInt( (9,1) ) ;

  when 1  =>     -- Parity Error   -- 11 %
   TestMode := UARTTB_PARITY_ERROR ;
   Data     := RV.RandSlv(0, 255, Data'length) ;
   Idle     := RV.RandInt(2, 15) ;

  when 2 =>      -- Stop Error  -- 11%
   . . .
  when 3 =>      -- Stop and Parity Error 6%
   . . .
```

```
        when 4 =>     -- Break Error  -- 2%
          . . .
        when others =>  report "DistInt Failed"  severity failure ;
      end case ;
      UartRxScoreboard.Push( (Data, TestMode) ) ;
      DoUartTxTransaction(UartTxRec, Data, TestMode, Idle) ;
    end loop ;
```

Constrained random tests are not the strong point of OSVVM. Since there is no solver, it is up to the test coder to ensure that a balance of vectors are generated. Even with SystemVerilog and a powerful solver, achieving coverage closure can be a challenging problem.

Instead, OSVVM focuses on using its Intelligent Coverage based randomization, which randomizes across coverage holes.

III. Functional Coverage

A. *What is Functional Coverage?*

Functional coverage is code that observes execution of a test plan. As such, it is code you write to track whether important values, sets of values, or sequences of values that correspond to design or interface requirements, features, or boundary conditions have been exercised.

Functional coverage is important to any verification approach since it is one of the factors used to determine when testing is done. Specifically, 100% functional coverage indicates that all items in the test plan have been tested. Combine this with 100% code coverage and it indicates that testing is done.

Functional coverage that examines the values within a single object is called either item (OSVVM and 'e') or point (SystemVerilog) coverage. Using item coverage, one might look at is different transfer sizes across a packet based bus. For example, the test plan may require that transfer sizes with the following size or range of sizes be observed: 1, 2, 3, 4 to 127, 128 to 252, 253, 254, or 255. Boundary conditions such as the smaller and larger are most important to look at individually since they are typically where errors occur.

Functional coverage that examines the relationships between different objects is called cross coverage. An example of this would be examining whether an ALU has done all of its supported operations with every different input pair of registers.

While collecting functional coverage is simply programming, it is much easier accomplished using either the package based utilities (OSVVM) or language syntax (SystemVerilog).

B. *Why Not Just Code Coverage?*

VHDL simulation tools can automatically calculate a metric called code coverage (assuming you have licenses for this feature). Code coverage tracks what lines of code or expressions in the code have been exercised.

Code coverage cannot detect conditions that are not in the code. For example, in the packet bus item coverage example discussed previously, code coverage cannot determine that the required values or ranges have occurred – unless the code contains expressions to test for each of these sizes. Instead, we need to write functional coverage.

Code coverage only collects information on existing code. Hence, it can reach 100% and essential features can be missing (ie: the new design requires 3 programmable timers and it currently has only 2).

Code coverage runs based on delta cycles and may run several times per clock cycle. As a result, unless it is conditioned to update only at the active edge of clock, it tends to be optimistic in nature.

Even with these limitations, code coverage can be important as it detects untested code - such as something either missing from the test plan or extra undocumented features.

C. *Why Functional Coverage?*

With any randomized test environment, we need functional coverage to track what the test actually did.

For directed tests, historically we assumed functional coverage. The directed test ran, therefore, it correctly activated the necessary conditions. This worked ok for small designs, however, as design complexity increases, this is no longer sufficient to ensure you have properly exercised your design. For example, the new design has larger memory blocks, so the design team quadrupled them - will the test suite still reach all of the boundary conditions (FIFO write while full)?

D. Cross Coverage with CoveragePkg

Cross coverage examines the relationships between different objects, such as making sure that each register source has been used with an ALU. The ALU is shown in Figure 3. Note that the test plan will also be concerned about what values are applied to the adder. We are not intending to address that part of the test here.

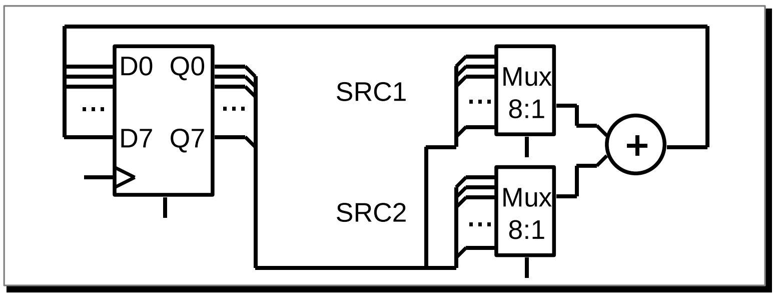

Figure 3: ALU

Cross coverage for SRC1 crossed SRC2 with can be visualized as a matrix of 8 x 8 bins as shown in Figure 4.

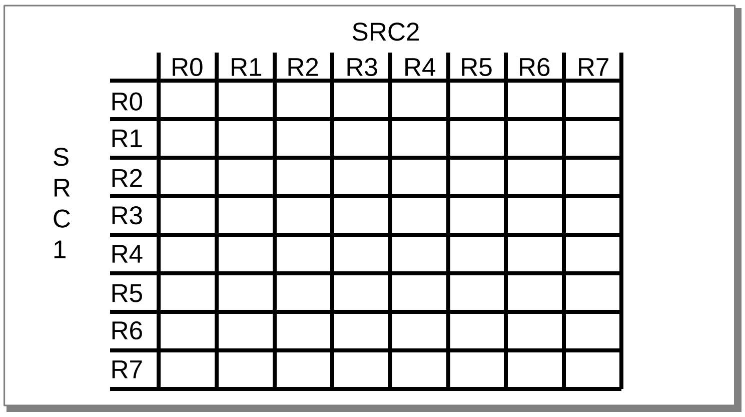

Figure 4: Visualizing cross coverage as a matrix

The basic steps to model functional coverage are declare the coverage object, create the coverage model, accumulate coverage, interact with the coverage data structure, and report the coverage.

Coverage is modeled using a data structure stored inside of a coverage object. The coverage object is created by declaring a shared variable of type CovPType, such as ACov shown below.

```
shared variable ACov : CovPType ;  -- Declare
```

Cross coverage is modeled using the method AddCross and two or more calls to function GenBin. AddCross creates the cross product of the set of bins (created by GenBin) on its inputs. The code below shows the call to create the 8 x 8 cross. Each call to GenBin(0,7) creates the 8 bins: 0, 1, 2, 3, 4, 5, 6, 7. The AddCross creates the 64 bins cross product of these bins. AddCross can support up to crossing 20 item bins.

```
ACov.AddCross( GenBin(0,7), GenBin(0,7) );
```

Coverage is accumulated using the method ICover. Since coverage is collected using sequential code, either clock based sampling or transaction based sampling (shown later in this example) can be used.

```
ACov.ICover( (Src1, Src2) ) ;  -- Accumulate
```

A test is done when functional coverage reaches 100%. The method IsCovered returns true when this occurs.

```
while not ACov.IsCovered loop    -- Done?
```

Finally, when the test is done, the method WriteBin is used to print the coverage results to OUTPUT (the transcript window when running interactively).

```
ACov.WriteBin ;  -- Report
```

Putting the entire example together, we end up with the following. Note for simplicity, coverage is collected in the stimulus generation process. Generally it is preferable to observe the coverage in separate model. However, in this case, where we have verified that "DoAluOp" drives this exact sequence, it will work ok.

```
architecture Test2 of tb is
  shared variable ACov : CovPType ;  -- Declare
begin
  TestProc : process
    variable RV : RandomPType ;
    variable Src1, Src2 : integer ;
  begin
    -- create coverage model
    ACov.AddCross( GenBin(0,7), GenBin(0,7) );  -- Model

    while not ACov.IsCovered loop    -- Done?
      Src1 := RV.RandInt(0, 7) ;     -- Uniform Randomization
      Src2 := RV.RandInt(0, 7) ;

      DoAluOp(TRec, Src1, Src2) ;    -- Transaction
      ACov.ICover( (Src1, Src2) ) ;  -- Accumulate
    end loop ;

    ACov.WriteBin ;  -- Report
    EndStatus(. . . ) ;
  end process ;
```

One thing to note at this point is that collecting coverage with OSVVM is just as concise as any language syntax.

IV. Intelligent Coverage is Faster

A. *Constrained Random Repeats Test Cases*

Just like solver based, constrained random, the previous example uses uniform randomization (code below) to select the register pairs for the ALU.

```
Src1 := RV.RandInt(0, 7) ;     -- Uniform Randomization
Src2 := RV.RandInt(0, 7) ;
```

Uniform randomization is only uniform for large sets of numbers. For small sets of numbers, like test generation, uniform randomization tends to repeat. Randomization theory tells us that to generate N test cases, it takes O(N*log N) randomizations.

Running the previous ALU testbench, it took 315 randomizations to generate all 64 unique pairs of registers. This is slightly less than 5X more iterations than the ideal case of 64 and is reasonably close to the predicted 64 * log(64). Figure 5 shows the resulting coverage matrix for this test. Note that some outputs actually reached 10 before others were covered (reached 1).

SRC2

SRC1

	R0	R1	R2	R3	R4	R5	R6	R7
R0	6	6	9	1	4	6	6	5
R1	3	4	3	6	9	5	5	4
R2	4	1	5	3	2	3	4	6
R3	5	5	6	3	3	4	4	6
R4	4	5	5	10	9	10	7	7
R5	4	6	3	6	3	5	3	8
R6	3	6	3	4	7	1	4	6
R7	7	3	4	6	6	5	4	5

Figure 5: Coverage results for constrained random style test

B. *Intelligent Coverage*

The goal of Intelligent Coverage randomization is to generate N unique test cases in N randomizations. Instead of writing separate constraints as is done with constrained random, Intelligent Coverage randomizes across the coverage holes and uses this information to generate the stimulus.

As a result, the steps for Intelligent Coverage randomization are write functional coverage, randomize using the functional coverage, and refine the values as necessary to generate the stimulus.

For the ALU example, we simply replace the two calls to RandInt with a single call to RandCovPoint (one of the Intelligent Coverage randomization methods). Note for cross coverage, RandCovPoint returns an integer_vector value. The code below shows the updated example.

```
architecture Test3 of tb is
  shared variable ACov : CovPType ;  -- Declare
begin
  TestProc : process
    variable RV : RandomPType ;
    variable Src1, Src2 : integer ;
  begin
    -- create coverage model
    ACov.AddCross( GenBin(0,7), GenBin(0,7) );  -- Model

    while not ACov.IsCovered loop    -- Done?
      (Src1, Src2) := ACov.RandCovPoint ; -- Intelligent Coverage Randomization

      DoAluOp(TRec, Src1, Src2) ;    -- Transaction
      ACov.ICover( (Src1, Src2) ) ;  -- Accumulate
    end loop ;

    ACov.WriteBin ;  -- Report
    EndStatus(. . . ) ;
  end process ;
```

This example completes in exactly 64 iterations, approximately 5X faster than constrained random.

C. *Refinement of Intelligent Coverage*

The process is not always this easy. Sometimes the value out of RandCovPoint will need to be further shaped by the stimulus generation process. Refinement involves writing code, and it can use either directed, algorithmic, file-based, or randomization methods.

The following is a contrived example that demonstrates the flexibility and capability of this approach. In this example, when a register pair on the diagonal (Src1 = Src2), then the code does two extra transactions, one to the diagonal before and one to the diagonal after the selected diagonal. In addition, it only collects coverage on the selected diagonal.

```
    while not ACov.IsCovered loop
        (Src1, Src2) := ACov.RandCovPoint ;
        if Src1 /= Src2 then
          DoAluOp(TRec, Src1, Src2) ;
          ACov.ICover( (Src1, Src2) ) ;
        else
          -- Do previous and following diagonal
          DoAluOp(TRec, (Src1-1) mod 8, (Src1-1) mod 8) ;
          DoAluOp(TRec,  Src1, Src1 ) ;
          DoAluOp(TRec, (Src1+1) mod 8, (Src1+1) mod 8) ;
          -- Can either record all or just selected diagonal
          ACov.ICover( (Src1, Src1) ) ;
        end if ;
      end loop ;
```

D. *Weighted Intelligent Coverage*

From some tests, we need to generate some conditions, transactions, or sequences more than others. To accomplish this we give different coverage bins different coverage goals. For the entire model to be covered, each bin much reach its specified coverage goal. When randomizing with coverage goals, the coverage goal is used as a randomization weight.

The following example reproduces the "weighted selection of test sequences" using intelligent coverage. Each coverage goal is specified as the first parameter to AddBins or AddCross. Each coverage bin is created with a separate call to AddBins. The notable difference between the Intelligent Coverage version is that it will reach coverage closure in exactly 100 iterations of the code - this is exactly the sum of the coverage goals.

```
    --  coverage goal is the first parameter
    Bin1.AddBins( 70, GenBin(0) ) ; -- Normal Handling, 70%
    Bin1.AddBins( 11, GenBin(1) ) ; -- Parity Error,    11%
    Bin1.AddBins( 11, GenBin(2) ) ; -- Stop Error,      11%
    Bin1.AddBins(  6, GenBin(2) ) ; -- Parity + Stop,    6%
    Bin1.AddBins(  2, GenBin(2) ) ; -- Break Error,      2%
    StimGen : while not Bin1.IsCovered loop
      iSequence := Bin1.RandCovPoint ;
      case iSequence is
        when 0  =>     -- Nominal case   -- 70%
          TestMode := UARTTB_NO_ERROR ;
          Data     := RV.RandSlv(0, 255, Data'length) ;
          Idle     := RV.DistInt( (9,1) ) ;

        when 1  =>     -- Parity Error   -- 11 %
          TestMode := UARTTB_PARITY_ERROR ;
          Data     := RV.RandSlv(0, 255, Data'length) ;
          Idle     := RV.RandInt(2, 15) ;

        when 2 =>      -- Stop Error  -- 11%
          . . .
        when 3 =>      -- Stop and Parity Error 6%
          . . .
        when 4 =>      -- Break Error  -- 2%
          . . .
        when others =>  report "DistInt Failed"  severity failure ;
      end case ;
      UartRxScoreboard.Push( (Data, TestMode) ) ;
      DoUartTxTransaction(UartTxRec, Data, TestMode, Idle) ;
    end loop ;
```

V. OSVVM IS MORE CAPABLE

OSVVM implements functional coverage in a data structure that is internal to the protected type (coverage object). This simplifies constructing a high fidelity model. Coverage can be constructed bin by bin as was done in the previous example. Coverage can be constructed algorithmically such as is the code below which gives all bins a coverage goal of 2 except for the diagonals which are 4.

```
TestProc : process
begin
  for i in 0 to 7 loop
    for j in 0 to 7 loop
      if i /= j then
        -- non-diagonal
        ACov.AddCross(2, GenBin(i),  GenBin(j));
      else
        -- diagonal
        ACov.AddCross(4, GenBin(i),  GenBin(j));
      end if ;
```

VI. COVERAGE CLOSURE

At the end of the day, to be done testing, we must reach coverage closure, or 100% functional coverage.

OSVVM focuses on coverage closure. The process is write functional coverage, randomize using functional coverage holes, and refine as necessary. For most cases, closure will happen by running the test long enough - the sum of the coverage goals.

Constrained random requires separate randomization constraints and functional coverage models. Coverage closure depends on the randomization constraints driving the inputs to the functional coverage conditions. After simulation, we merge all functional coverage databases and use tools to help us prune out tests that are not increasing the functional coverage.

VII. SUMMARY

OSVVM's Intelligent Coverage randomization is a simple, powerful, concise methodology for test generation. It consists of writing functional coverage, randomizing across coverage holes, and refining with directed, algorithmic, file-based or randomization methods.

OSVVM offers a number of benefits over SystemVerilog and 'e'.

- Readable by RTL engineers – no OO, transaction initiation with procedures, structural code for netlists.
- Works with your existing VHDL environment – including TLM
- Supports mixed approaches – directed, algorithmic, file-based, constrained and Intelligent Coverage random
- Functional coverage is modeled incrementally using sequential code. No OO required.
- Intelligent testbench capability (Intelligent Coverage) is built in and free
- Functional coverage, constrained random, and Intelligent Coverage can be refined with code.
- Packages are open source, and hence, extensible
- Fast test construction – primary focus is functional coverage
- Faster simulations – no redundant stimulus (log N) and no solver

REFERENCES

[1] Jim Lewis, “Functional Coverage using CoveragePkg,” January 2014, CoveragePkg_user_guide.pdf

[2] Jim Lewis, “Randomization using RandomPkg,” January 2014, RandomPkg_user_guide.pdf

[3] Jim Lewis, “VHDL Testbenches and Verification,” May 2014, Training Manual