Functional Coverage Using CoveragePkg

User Guide for Release 2014.01

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 Overview

The VHDL package, CoveragePkg, provides subprograms that facilitate implementation of functional coverage within VHDL. It is a core part of the Open Source VHDL Verification Methodology (OSVVM). While CoveragePkg is just a package, it offers a similar conciseness to the language syntax of other verification languages, such as SystemVerilog or 'e'. In addition it offers capability and flexibility that is a step ahead.

Functional coverage is code we write to track execution of a test plan. It is important to any verification approach since it is one of the factors in determining when testing is done. I will address this more in the section, "What is Functional Coverage and why do I need it?" In this section I will also address, "Why can't I just use code coverage?" and "Why you need functional coverage, even with directed testing."

Writing functional coverage is concise and flexible. The basics of writing functional coverage using coverage package are covered in the section, "Writing Functional Coverage using CoveragePkg."

One important, unique feature is the "Intelligent Coverage" that is built directly into the coverage data structure. This capability allows us to randomly select a hole in the current functional coverage to pass to the stimulus generation process. Using "Intelligent Coverage" helps minimize the number of test cases generated to achieve complete coverage - resulting in fewer simulation cycles and a higher velocity of verification. More details are provided in the section, "Intelligent Coverage is 5X or more faster than constrained random testing."

Functional coverage with CoveragePkg is captured incrementally using sequential code. This provides a great deal of flexibility and capability, and facilitates writing high fidelity functional coverage models. More details are provided in the section, "Flexibility and Capability."

CoveragePkg provides numerous methods (functions and procedures of a protected type) that provide a powerful capability. These methods are documented in the remaining part of the document. Note the package contains additional undocumented methods that are either experimental features or artifacts from older use models that

are maintained for backward compatibility. Use these at your own risk as they may be removed from future revisions.

This documentation is not a substitute for a great training class. CoveragePkg was developed and is maintained by Jim Lewis of SynthWorks. It evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and Verification class. This class includes additional packages that are not yet part of OSVVM. Please support our effort in supporting OSVVM by purchasing your VHDL training from SynthWorks.

All CoveragePkg features are supported now by many simulators. The only required features are protected types (VHDL-2002) and integer_vector (in package std.standard in VHDL-2008). In addition, since they are open source, the packages are free (download and usage) and will be updated on a regular basis.

2 Getting CoveragePkg

CoveragePkg is released under the Perl Artistic open source license. It is free (both to download and use - there are no license fees). You can download it from http://www.synthworks.com/downloads. It will be updated from time to time. Currently there are numerous planned revisions.

If you add features to the package, please donate them back under the same license as candidates to be added to the standard version of the package package. If you need features, be sure to contact us. I blog about the packages at http://www.synthworks.com/blog. We also support a user community and blogs through http://www.osvvm.org.

The STANDARD revision of this package requires VHDL-2008. It will work with a VHDL-2002 compliant simulator by uncommenting the VHDL-2008 compatibility packages.

Release notes are in the document CoveragePkg_release_notes.pdf.

3 What is Functional Coverage and why do I need it?

3.1 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 point (SystemVerilog) or item ('e') coverage. I prefer the term item coverage since point can also be a single value within a particular bin. One relationship we 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.

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.

Many think functional coverage is an exclusive capability of a verification language such as SystemVerilog. However, functional coverage collection is really just a matter of implementing a data structure.

CoveragePkg contains a protected type and methods (procedures and functions of a protected type) that facilitates creating the functional coverage data structure and writing functional coverage.

3.2 Why can't I just use 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 above, 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.

In the ALU cross coverage example above, code coverage cannot determine whether particular register pairs have been used together, unless the code is written this way. Generally each input to the ALU is selected independently of the other. Again, we need to write functional coverage.

Code coverage on a partially implemented design can reach 100%. It cannot detect missing features (oops forgot to implement one of the timers) and many boundary conditions (in particular those that span more than one block). Hence, code coverage cannot be used exclusively to indicate we are done testing.

In addition, code coverage is an optimistic metric. In combinational logic code in an HDL, a process may be executed many times during a given clock cycle due to delta cycle changes on input signals. This can result in several different branches of code being executed. However, only the last branch of code executed before the clock edge truly has been covered.

3.3 Test Done = Test Plan Executed and All Code Executed

To know testing is done, we need to know that both the test plan is executed and all of the code has been executed. Is 100% functional coverage enough?

Unfortunately a test can reach 100% functional coverage without reaching 100% code coverage. This indicates the design contains untested code that is not part of the test plan. This can come from an incomplete test plan, extra undocumented features in the design, or case statement others branches that do not get exercised in normal hardware operation. Untested features need to either be tested or removed.

As a result, even with 100% functional coverage it is still a good idea to use code coverage as a fail-safe for the test plan.

3.4 Why You Need Functional Coverage, even with Directed Testing

You might think, "I have written a directed test for each item in the test plan, I am done right?"

As design size grows, the complexity increases. A test that completely validates one version of the design, may not validate the design after revisions. For example, if the size a of FIFO increases, the test may no longer provide enough stimulus values to fill it completely and cause a FIFO Full condition. If new features are added, a test may need to change its configuration register values to enable the appropriate mode.

Without functional coverage, you are assuming your directed, algorithmic, file based, or constrained random test actually hits the conditions in your test plan.

Don't forget the engineers creed, "In the divine we trust, all others need to show supporting data." Whether you are using directed, algorithmic, file based, or constrained random test methods, functional coverage provides your supporting data.

3.5 What is "Coverage" then?

The word coverage can refer to functional coverage, code coverage, or property coverage (such as with PSL). Since this document focuses on functional coverage, when the word coverage is used by itself, it is functional coverage.

4 Writing Functional Coverage Using CoveragePkg

Functional coverage can be written using any code. CoveragePkg and language syntax are solely intended to simplify this effort. In this section, we will first look at implementing functional coverage manually (without CoveragePkg). Then we will look at using CoveragePkg to capture item and cross coverage.

4.1 Item (Point) Coverage done Manually

In this subsection we write item coverage using regular VHDL code. While for most problems this is the hard way to capture coverage, it provides a basis for understanding functional coverage.

In a packet based transfer (such as across an ethernet port), most interesting things happen when the transfer size is at or near either the minimum or maximum sized transfers. It is important that a number of medium sized transfers occur, but we do not need to see as many of them. For this example, let's assume that we are interested in tracking transfers that are either the following size or range: 1, 2, 3, 4 to 127, 128 to 252, 253, 254, or 255. The sizes we look for are specified by our test plan.

We also must decide when to capture (aka sample) the coverage. In the following code, we use the rising edge of clock where the flag TransactionDone is 1.

```
signal Bin : integer_vector(1 to 8) ;
. . .
process
begin
  wait until rising_edge(Clk) and TransactionDone = '1' ;
  case to_integer(unsigned(ActualData)) is
    when   1 =>          Bin(1) <= Bin(1) + 1 ;
    when   2 =>          Bin(2) <= Bin(2) + 1 ;
    when   3 =>          Bin(3) <= Bin(3) + 1 ;
    when   4 to 127 =>   Bin(4) <= Bin(4) + 1 ;
    when 128 to 252 =>   Bin(5) <= Bin(5) + 1 ;
    when 253 =>          Bin(6) <= Bin(6) + 1 ;
    when 254 =>          Bin(7) <= Bin(7) + 1 ;
```

```
      when 255 =>          Bin(8) <= Bin(8) + 1 ;
      when others =>
    end case ;
  end process ;
```

Any coverage can be written this way. However, this is too much work and too specific to the problem at hand. We could make a small improvement to this by capturing the code in a procedure. This would help with local reuse, but there are still no built-in operations to determine when testing is done, to print reports, or to save results and the data structure to a file.

4.2 Basic Item (Point) Coverage with CoveragePkg

In this subsection we use CoveragePkg to write the item coverage for the same packet based transfer sizes created in the previous section manually. Again, we are most interested in the smallest and largest transfers. Hence, for an interface that can transfer between 1 and 255 words we will track transfers of the following size or range: 1, 2, 3, 4 to 127, 128 to 252, 253, 254, and 255.

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 CovBin1 shown below.

```
architecture Test1 of tb is
  shared variable CovBin1 : CovPType ;
begin
```

Internal to the data structure, each bin in an item coverage model is represented by a minimum and maximum value (effectively a range). Bins that have only a single value, such as 1 are represented by the pair 1, 1 (meaning 1 to 1). Internally, the minimum and maximum values are stored in a record with other bin information.

The coverage model is constructed by using the method AddBins and the function GenBin. The function GenBin transforms a bin descriptor into a set of bins. The method AddBins inserts these bins into the data structure internal to the protected type. Note that when calling a method of a protected type, such as AddBins shown below, the method name is prefixed by the protected type variable name, CovBin1. The version of GenBin shown below has three parameters: min value, max value, and number of bins. The call, GenBin(1,3,3), breaks the range 1 to 3 into the 3 separate bins with ranges 1 to 1, 2 to 2, 3 to 3.

```
TestProc : process
begin
  --                    min, max, #bins
  CovBin1.AddBins(GenBin(1,   3,   3)); -- bins 1 to 1, 2 to 2, 3 to 3
  . . .
```

Additional calls to AddBins appends additional bins to the data structure. As a result, the call, GenBin(4, 252, 2), appends two bins with the ranges 4 to 127 and 128 to 252 respectively to the coverage model.

```
CovBin1.AddBins(GenBin(  4, 252, 2)) ; -- bins 4 to 127 and 128 to 252
```

Since creating one bin for each value within a range is common, there is also a version of GenBin that has two parameters: min value and max value which creates one bin per value. As a result, the call GenBin(253, 255) appends three bins with the ranges 253 to 253, 254 to 254, and 255 to 255.

```
CovBin1.AddBins(GenBin(253, 255)) ; -- bins 253, 254, 255
```

Coverage is accumulated using the method ICover. Since coverage is collected using sequential code, either clock based sampling (shown below) or transaction based sampling (by calling ICover after a transaction completes - shown in later examples) can be used.

```
-- Accumulating coverage using clock based sampling
loop
  wait until rising_edge(Clk) and nReset = '1' ;
  CovBin1.ICover(to_integer(unsigned(RxData_slv))) ;
end loop ;
end process ;
```

A test is done when functional coverage reaches 100%. The method IsCovered returns true when all the count bins in the coverage data structure have reached their goal. The following code shows the previous loop modified so that it exits when coverage reaches 100%.

```
-- capture coverage until coverage is 100%
while not CovBin1.IsCovered loop
    wait until rising_edge(Clk) and nReset = '1' ;
    CovBin1.ICover(to_integer(RxData_slv)) ;
end loop ;
```

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

```
-- Print Results
CovBin1.WriteBin ;
```

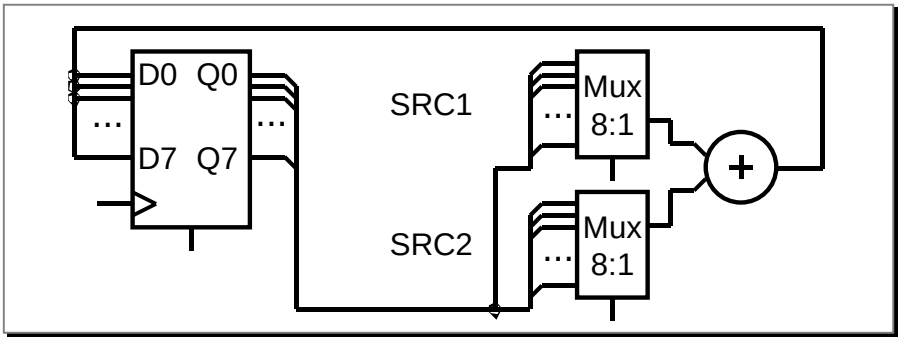
Putting the entire example together, we end up with the following.

```
architecture Test1 of tb is
  shared variable CovBin1 : CovPType ;  -- Coverage Object
begin
  TestProc : process
  begin
    -- Model the coverage
    CovBin1.AddBins(GenBin(  1,   3     )) ;
    CovBin1.AddBins(GenBin(  4, 252, 2)) ;
    CovBin1.AddBins(GenBin(253, 255   )) ;

    -- Accumulating Coverage
    -- clock based sampling
    while not CovBin1.IsCovered loop
      wait until rising_edge(Clk) and nReset = '1' ;
      CovBin1.ICover(to_integer(RxData_slv)) ;
    end loop ;

    -- Print Results
    CovBin1.WriteBin ;
    wait ;
  end process ;
```

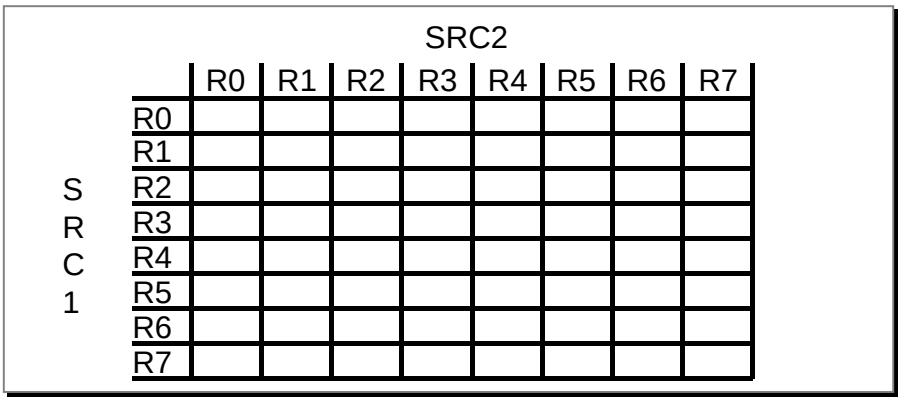
Note that when modeling coverage, we primarily work with integer values. All of the inputs to GenBin and ICover are integers; WriteBin reports results in terms of integers. This is similar to what other verification languages do.

4.3 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 hardware we are working with is as shown below. 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.



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



The steps for modeling cross coverage are the same steps used for item coverage: declare, model, accumulate, interact, and report. Collecting cross coverage only differs in the model and accumulate steps.

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. This can be visualized as the matrix shown previously.

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

AddCross supports crossing of up to 20 items. Internal to the data structure there is a record that holds minimum and maximum values for each item in the cross. Hence for the first bin, the record contains SRC1 minimum 0, SRC1 maximum 0, SRC2 minimum 0, and SRC2 maximum 0. The record also contains other bin information (such as coverage goal, current count, bin type (count, illegal, ignore), and weight).

The accumulate step now requires a value for SRC1 and SRC2. The overloaded ICover method for cross coverage uses an integer_vector input. This allows it to accept a value for each item in the cross. The extra set of parentheses around Src1 and Src2 in the call to ICover below designate that it is a integer_vector.

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

The code below shows the entire example. The shared variable, ACov, declares the coverage object. AddCross creates the cross coverage model. IsCovered is used to determine when all items in the coverage model have been covered. Each register is selected using uniform randomization (RandInt). The transaction procedure, DoAluOp,

applies the stimulus. ICover accumulates the coverage. WriteBin reports the coverage.

```
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 ;
```

5 Intelligent Coverage is 5X or more faster than constrained random

5.1 Constrained Random Repeats Test Cases

In the previous section we used uniform randomization (shown below) to select the register pairs for the ALU. Constrained random at its best produces a uniform distribution. As a result, the previous example is a best case model of constrained random tests.

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

The problem with constrained random testbenches is that they repeat some test cases before generating all test cases. In general to generate N cases, it takes "N * log N" randomizations. The "log N" represents repeated test cases and significantly adds to simulation run times. Ideally we would like to run only N test cases.

Running the previous ALU testbench, we get the following coverage matrix when the code completes. Note that some case were generated 10 time before all were done at least 1 time. It took 315 randomizations to generate all 64 unique pairs. This is slightly less than 5X more iterations than the 64 in the ideal case. This correlates well with theory as 315 ≈ 64 * log(64). By changing the seed value, the exact number of randomizations may increase or decrease but this would be a silly way to try to reduce the number of iterations a test runs.

		SRC2							
		R0	R1	R2	R3	R4	R5	R6	R7
SRC1	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

5.2 Intelligent Coverage

"Intelligent Coverage" is a coverage driven randomization approach that randomly selects a hole in the functional coverage and passes it to the stimulus generation process. Using "Intelligent Coverage" allows the stimulus generation to focus on missing coverage and reduces the number of test cases generated to approach the ideal of N randomizations to generate N test cases.

Let's return to the ALU example. The Intelligent Coverage methodology starts by writing functional coverage. We did this in the previous example too. Next preliminary stimulus is generated by randomizing using the functional coverage model. In this example, we will replace the uniform randomization with RandInt with a call to RandCovPoint (one of the Intelligent Coverage randomization methods). This is shown below. In this case, Src1 and Src2 are used directly in the test, so we are done.

```
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 ;
```

When randomizing across a cross coverage model, the output of RandCovPoint is an integer_vector. Instead of using the separate integers, Src1 and Src2, it is also possible to use an integer_vector as shown below.

```
variable Src : integer_vector(1 to 2) ;
. . .
Src := ACov.RandCovPoint ;      -- Intelligent Coverage Randomization
```

The process is not always this easy. Sometimes the value out of RandCovPoint will need to be further shaped by the stimulus generation process.

The Intelligent Coverage methodology works now and works with your current testbench approach. You can adopt this methodology incrementally. Add functional coverage today to make sure you are executing all of your test plan. For the tests that need help, use the Intelligent Coverage.

5.3 Intelligent Coverage reduces your work

The Intelligent Coverage methodology is different from what is done in a constrained random methodology. Rather than randomizing across holes in the functional coverage, the constrained random approach adds an equally complex set of randomization constraints to shape the stimulus. In many ways, the randomization constraints and functional coverage needed in a constrained random approach are duplicate views of the same information.

With Intelligent coverage, we focus on writing high fidelity coverage models. The constrained random step is reduced to a refinement step and only needs to focus on things that are not already shaped by the coverage. Hence, Intelligent Coverage methodology reduces (or eliminates) the work needed to generate test constraints.

6 Flexibility and Capability

OSVVM implements functional coverage as a data structure within a protected type. Using methods of the protected type allows both a concise capture of the model (as we saw in the previous examples) and a great degree of flexibility and capability.

The added flexibility and capability comes from writing the model incrementally using any sequential code (if, loop). As long as the entire model is captured before we start collecting coverage, we can use as many calls to AddBins or AddCross as needed. As a result, conditionally capturing coverage based on a generic is straight forward. In addition, algorithms that iterate using a loop are no more trouble than writing the code.

Additional flexibility and capability comes from being able to give each bin within a coverage model a different coverage goal. A coverage goal specifies the number of times a value from a particular bin needs to be observed before it is considered covered. The Intelligent Coverage randomization by default will use these coverage goals as randomization weights.

To demonstrate this flexibility, let's consider a contrived example based on the ALU. In this example, each SRC1 crossed with any SRC2 has a different coverage goal. In addition, it is an error if SRC1 and SRC2 are equal. The coverage goal for each bin is specified in the table below.

Coverage Goal	**Src1**	**Src2**
2	0	1, 2, 3, 4, 5, 6, 7
3	1	0, 2, 3, 4, 5, 6, 7
4	2	0, 1, 3, 4, 5, 6, 7
5	3	0, 1, 2, 4, 5, 6, 7
5	4	0, 1, 2, 3, 5, 6, 7
4	5	0, 1, 2, 3, 4, 6, 7
3	6	0, 1, 2, 3, 4, 5, 7
2	7	0, 1, 2, 3, 4, 5, 6
Illegal	Src1 = Src2	

To model the above functional coverage, we use a separate call for each different coverage goal. The function, IllegalBin, is used to mark the bins with Src1 = Src2 illegal. This is shown below.

```
architecture Test4 of tb is
  shared variable ACov : CovPType ;               -- Declare Cov Object
begin
  TestProc : process
    variable Src1, Src2 : integer ;
  begin
    -- Capture coverage model
    ACov.AddCross(  2,  GenBin (0),  IllegalBin(0) & GenBin(1,7)) ;
    ACov.AddCross(  3,  GenBin (1),  GenBin(0)   & IllegalBin(1) & GenBin(2,7)) ;
    ACov.AddCross(  4,  GenBin (2),  GenBin(0,1) & IllegalBin(2) & GenBin(3,7)) ;
    ACov.AddCross(  5,  GenBin (3),  GenBin(0,2) & IllegalBin(3) & GenBin(4,7)) ;
    ACov.AddCross(  5,  GenBin (4),  GenBin(0,3) & IllegalBin(4) & GenBin(5,7)) ;
    ACov.AddCross(  4,  GenBin (5),  GenBin(0,4) & IllegalBin(5) & GenBin(6,7)) ;
    ACov.AddCross(  3,  GenBin (6),  GenBin(0,5) & IllegalBin(6) & GenBin(7)) ;
    ACov.AddCross(  2,  GenBin (7),  GenBin(0,6) & IllegalBin(7) ) ;

    while not ACov.IsCovered loop             -- Done?
      (Src1, Src2) := ACov.RandCovPoint ;     -- Randomize with coverage

      DoAluOp(TRec, Src1, Src2) ;             -- Do a transaction
      ACov.ICover( (Src1, Src2) ) ;           -- Accumulate
    end loop ;

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

Note that all of the methods we have discussed have all of the methods have additional overloading that is documented in the CoveragePkg users guide (CoveragePkg_user_guide.pdf).

7 Declaration of the Coverage Object - CovPType

A coverage model is contained within a CovPType typed shared variable. Using a protected type allows both access to the structure from multiple processes and hides details of the model within the data structure.

The shared variable declaration for the coverage object is commonly put in the architecture of the design as shown below.

```
architecture Test3 of tb is
  shared variable ACov : CovPType ;               -- Declare Cov Object
begin
```

8 Basic Bin Description

The functions GenBin, IllegalBin, and IgnoreBin are used to create bins of type CovBinType. These bins are used as inputs to the methods, AddBins and AddCross, that create the coverage data structure. Using these functions replaces the need to know the details of type CovBinType.

8.1 Basic Type for Coverage Bins

The output type of the functions GenBin, IllegalBin, and IgnoreBin is CovBinType. It is declared as an array of the record type, CovBinBaseType. This is shown below. Note the details of CovBinBaseType are not provided as they may change from time to time.

```
type CovBinBaseType is record
  . . .
end record ;
type CovBinType is array (natural range <>) of CovBinBaseType ;
```

8.2 Creating Count Bins - GenBin

The following are five variations of GenBin. The ones with AtLeast and Weight parameters are mainly intended to for use with constants.

```
function GenBin(Min, Max, NumBin : integer ) return CovBinType ;
function GenBin(Min, Max : integer) return CovBinType ;
function GenBin(A : integer) return CovBinType ;
```

The version of GenBin shown below has three parameters: min value, max value, and number of bins. The call, GenBin(1, 3, 3), breaks the range 1 to 3 into the 3 separate bins with ranges1 to 1, 2 to 2, 3 to 3.

```
--                      min, max, #bins
CovBin1.AddBins(GenBin(1,   3,   3)); -- bins 1 to 1, 2 to 2, 3 to 3
```

If there are less values (between max and min) than bins, then only "max - min + 1" bins will be created. As a result, the call GenBin(1,3,20), will still create the three bins: 1 to 1, 2 to 2 and 3 to 3.

```
CovBin2.AddBins( GenBin(1, 3, 20) ) ; -- bins 1 to 1, 2 to 2, and 3 to 3
```

If there are more values (between max and min) than bins and the range does not divide evenly among bins, then each bin with have on average (max - min + 1)/bins. Later bins will have one more value than earlier bins. The exact formula used is (number of values remaining)/(number of bins remaining). As a result, the call GenBin(1, 14, 4) creates four bins with ranges 1 to 3, 4 to 6, 7 to 10, and 11 to 14.

```
CovBin2.AddBins( GenBin(1, 14, 4) ) ; -- 1 to 3, 4 to 6, 7 to 10, 11 to 14
```

Since creating one bin per value in the range is common, there is also a version of GenBin that has two parameters: min value and max value which creates one bin per value. As a result, the first call to AddBins/GenBin can be shortened to the following.

```
--                     min, max
CovBin1.AddBins(GenBin(1,   3));  -- bins 1 to 1, 2 to 2, and 3 to 3
```

GenBin can also be called with one parameter, the one value that is contained in the bin. Hence the call, GenBin(5) creates a single bin with the range 5 to 5. The following two calls are equivalent.

```
CovBin3.AddBins( GenBin(5) ) ;
CovBin3.AddBins( GenBin(5,5,1) ) ;  -- equivalent call
```

8.3 Creating Illegal and Ignore Bins - IllegalBin and IgnoreBin

When creating bins, at times we need to mark bins as illegal and flag errors or as ignored actions and not to count them.

The functions IllegalBin and IgnoreBin are used to create illegal and ignore bins. One version of IllegalBin and IgnoreBin has three parameters: min value, max value, and number of bins (just like GenBin).

```
--         min, max, NumBin
IllegalBin( 1,   9,   3) -- creates 3 illegal bins: 1-3, 4-6, 7-9
IllegalBin( 1,   9,   1) -- creates one illegal bin with range 1-9
IgnoreBin ( 1,   3,   3) -- creates 3 ignore bins: 1, 2, 3
```

There are also two parameter versions of IgnoreBin and IllegalBin that creates a single bin. Some examples of this are illustrated below. While this is different from the action of the two parameter GenBin calls, it matches the common behavior of creating illegal and ignore bins.

```
--         min, max
IllegalBin( 1,   9) -- creates one illegal bin with range 1-9
IgnoreBin ( 1,   3) -- creates one ignore bin with range 1-3
```

There are also one parameter versions of IgnoreBin and IllegalBin that creates a single bin with a single value. Some examples of this are illustrated below.

```
--          AVal
IllegalBin(  5  ) -- creates one illegal bin with range 5-5
IgnoreBin (  7  ) -- creates one ignore bin with range 7-7
```

8.4 Predefined Bins - ALL_BIN, …, ALL_ILLEGAL, ZERO_BIN, ONE_BIN

The following are predefined bins.

```
constant ALL_BIN     : CovBinType := GenBin(integer'left, integer'right, 1) ;
constant ALL_COUNT   : CovBinType := GenBin(integer'left, integer'right, 1) ;
constant ALL_ILLEGAL : CovBinType := IllegalBin(integer'left, integer'right, 1) ;
constant ALL_IGNORE  : CovBinType := IgnoreBin(integer'left, integer'right, 1) ;
constant ZERO_BIN    : CovBinType := GenBin(0) ;
constant ONE_BIN     : CovBinType := GenBin(1) ;
```

8.5 Combining Bins Using Concatenation - &

Since GenBin, IllegalBin, and IgnoreBin all return CovBinType, their results can be concatenated together. As a result, the following calls to GenBin creates the bins: 1 to 1, 2 to 2, 3 to 3, 2 to 127, 128 to 252, 253 to 253, 254 to 254, and 255 to 255.

```
CovBin1.AddBins(GenBin(0, 2) & GenBin(3, 252, 2) & GenBin(253, 255));
```

Calls to GenBin, IllegalBin, and IgnoreBin can also be combined. As a result the following creates the four separate legal bins (1, 2, 5, and 6), a single ignore bin (3 to 4), and everything else falls into an illegal bin.

```
CovBin2.AddBins( GenBin(1,2) & IgnoreBin(3,4) & GenBin(5,6) & ALL_ILLEGAL ) ;
```

9 Data Structure Construction

The coverage model data structure is created using the methods AddBins and AddCross.

9.1 Item (Point) Bins - AddBins

The method AddBins is used to add item coverage bins to the coverage data structure. Each time it is called new bins are appended after any existing bins. AddBins has additional parameters to allow specification of coverage goal (AtLeast) and randomization weight (Weight). By using separate calls to AddBins, each bin can have a different coverage goal and/or randomization weight.

```
procedure AddBins (CovBin : CovBinType) ;
```

9.2 Cross Coverage Bins - AddCross

The method AddCross is used to add cross coverage bins to the coverage data structure. Each time it is called new bins are appended after any existing bins. AddCross has additional parameters to allow specification of coverage goal (AtLeast) and randomization weight (Weight). By using separate calls to AddCross, each bin can have a different coverage goal and/or randomization weight.

```
procedure AddCross(
    Bin1, Bin2 : CovBinType ;
    Bin3, Bin4, Bin5, Bin6, Bin7, Bin8, Bin9, Bin10, Bin11, Bin12, Bin13,
    Bin14, Bin15, Bin16, Bin17, Bin18, Bin19, Bin20 : CovBinType := NULL_BIN
) ;
```

9.3 Controlling Reporting for Illegal Bins - SetIllegalMode

By default, illegal bins both count and flag an error. This behavior is controlled by the IllegalMode variable. The default value of the variable is ILLEGAL_ON. Setting IllegalMode to ILLEGAL_OFF, as shown below, suppresses printing of messages when an item is added to an illegal bin. Setting IllegalMode to ILLEGAL_FAILURE causes a severity failure message to be printed when an item is added to an illegal bin.

```
type IllegalModeType is (ILLEGAL_ON, ILLEGAL_FAILURE, ILLEGAL_OFF) ;
CovBin4.SetIllegalMode(ILLEGAL_OFF) ;  -- Illegal printing off
CovBin4.SetIllegalMode(ILLEGAL_ON) ;  --  Default:  Illegal printing on
```

9.4 Bin Size Optimization - SetBinSize

SetBinSize can help the creation of a coverage model be more efficient by pre-declaring the number of bins to be created in the coverage data structure. Use this for small bins to save space or for large bins to suppress the resize and copy that occurs when the bins automatically resize.

```
procedure SetBinSize (NewNumBins : integer) ;
```

10 Accumulating Coverage - ICover

The method ICover is used to accumulate coverage. For item (point) coverage, ICover accepts an integer value. For cross coverage, ICover accepts an integer_vector. The procedure interfaces are shown below. Since the coverage accumulation is written procedurally, ICover will support either clock based sampling or transaction based sampling (examples of both shown previously).

```
procedure ICover( CovPoint : in integer ) ;
procedure ICover( CovPoint : in integer_vector ) ;
```

Since the inputs must be either type integer or integer_vector, conversions must be used. To convert from std_logic_vector to integer, numeric_std_unsigned and numeric_std provide the following conversions.

```
CovBin3.ICover( to_integer(RxData_slv) ) ;  -- using numeric_std_unsigned (2008)
CovBin3.ICover( to_integer(unsigned(RxData_slv)) ) ;  -- using numeric_std
```

To convert either std_logic or boolean to integer, CoveragePkg provides overloading for to_integer.

```
CovBin3.ICover( to_integer(Empty) ) ;        -- std_logic
CovBin3.ICover( to_integer(Empty = '1') ) ;  -- boolean
```

To convert either std_logic_vector or boolean_vector to integer_vector (bitwise), CoveragePkg provides to_integer_vector functions.

```
CrossBin.ICover( to_integer_vector(CtrlReg_slv) ) ;  -- std_logic_vector
CrossBin.ICover( to_integer_vector((Empty='1')&(Rdy='1')) ) ;  -- boolean_vector
```

Since the language does not do introspection of aggregate values when determining the type of an expression, the boolean vector expression needs to be constructed using concatenation (as shown above) rather than aggregates (as shown below).

```
--! CrossBin.ICover( to_integer_vector( ((Empty='1'),(Rdy='1')) ));  -- ambiguous
```

11 Basic Randomization

Randomization is handled by either RandCovPoint and RandCovBinVal. The randomization is coverage target based. Once a count bin has reached its coverage goal it is no longer selected for randomization. The randomization results can be modified by using coverage goals, randomization weights, coverage targets, and randomization thresholds. These topics are discussed later in this document.

11.1 Randomly generating a value within a bin - RandCovPoint

RandCovPoint returns a randomly selected value (also referred to as a point) within the randomly selected bin. It returns integer_vector values for cross coverage bins, and integer or integer_vector for item (point) bins. The overloading for RandCovPoint is shown below.

```
impure function RandCovPoint return integer_vector ;
impure function RandCovPoint return integer ;
```

11.2 Randomly selecting a coverage bin - RandCovBinVal

RandCovBinVal returns a randomly selected bin value of type RangeArrayType. The type RangeArrayType and the function definitions are shown below. Note RangeArrayType may change in the future.

```
type RangeType is record
  min, max : integer ;
end record ;
type RangeArrayType is array (integer range <>) of RangeType;
impure function RandCovBinVal return RangeArrayType ;
```

11.3 Randomization, Illegal, and Ignore Bins

RandCovPoint and RandCovBinVal will never select a bin marked as illegal or ignore. However, if count bin intersects with a prior specified illegal or ignore bin then the illegal or ignore value may be generated by randomization. Currently care must be taken to avoid this. In revision 2013.04, if merging is enabled (see SetMerging) any count bin that is included in a prior illegal or ignore bin will be dropped.

12 Coverage Model Statistics

Coverage model statistics collecting methods allow us to check if the model is covered/testing is done (IsCovered), check if the model is initialized (IsInitialized), or check the current total coverage (GetCov).

12.1 Model Covered - Testing Done - IsCovered

The function IsCovered returns TRUE when all count bins have reached their coverage goal. This indicates that coverage is complete and testing is done. IsCovered is declared as follows. Just like ICover, IsCovered is called either at a sampling point of either the clock or a transaction.

```
impure function IsCovered return boolean ;  -- Uses CovTarget
impure function IsCovered ( PercentCov : real ) return boolean ;
```

12.2 Model Initialized - IsInitialized

The function IsInitialized returns a true when a coverage model has bins (has been initialized). IsInitialized is a useful check when constructing the coverage model in a separate process from collecting the coverage.

```
impure function IsInitialized return boolean ;
```

12.3 Number of Items Randomized - GetItemCount

The function GetItemCount returns the number of items that have been randomized in the coverage model.

```
impure function GetItemCount return integer ;
```

12.4 Total Coverage Goal - GetTotalCovGoal

The function GetTotalCovGoal returns the sum of each bins coverage. Coverage models with a simple relationship between the stimulus and the desired coverage will reach coverage closure in GetTotalCovGoal number of randomizations.

```
impure function GetTotalCovGoal return integer ;  -- uses CovTarget
impure function GetTotalCovGoal ( PercentCov : real ) return integer ;
```

12.5 Current Percent Coverage - GetCov

The function GetCov returns a type real value that indicates the current percent completion (0.0 to 100.0) of the coverage model. It has the following overloading.

```
impure function GetCov return real ;
impure function GetCov (PercentCov : real) return real ;
```

13 Reporting Coverage

Coverage results can be written as either all the bins (WriteBin) or just the bins that have not reached their coverage goal (WriteCovHoles). These results can either be printed to the std_output (file OUTPUT) or to a designated file. In addition, one or more lines of heading (SetMessage) may be printed before the results.

13.1 Reporting Bin Results - WriteBin

The procedure WriteBin prints out the coverage results with one bin printed per line. All count bins are printed. Illegal bins are printed if they have a non-zero count. Ignore bins are not printed. The weight field of the coverage bin is only printed when the weight is being use (see WeightMode). Its declaration and an example of usage is shown below.

```
procedure WriteBin ;
. . .
CovBin1.WriteBin ;
```

13.2 Reporting Coverage Holes - WriteCovHoles

WriteCovHoles prints out count bin results that are below a the coverage goal. Its declaration and an example usage is shown below.

```
procedure WriteCovHoles ;
. . .
CovBin1.WriteCovHoles ;
```

13.3 Setting Headings - SetMessage

The method SetMessage sets headings for WriteBin and WriteCovHoles. Each call to SetMessage creates a separate line in the output of either WriteBin or WriteCovHoles.

```
procedure SetMessage (NameIn : String ) ;
. . .
CovBin1.SetMessage("DMA") ;            -- first line of heading
CovBin1.SetMessage("Stat, WordCnt") ;  -- second line of heading
```

If the headings need to be cleared, use the method DeallocateMessage. It is called as follows.

```
CovBin1.DeallocateMessage ;            -- clears all headings
```

If the internal randomization seed has not yet been initialized, the first call to SetMessage will initialize the seed using the string value.

The method SetItemName is deprecated. It is currently maintained for backward compatibility and it simply calls SetMessage.

If SetMessage is not set, the value in SetName will be used instead.

13.4 Setting Coverage Model Name - SetName

The method SetName sets the name of the coverage model. The coverage model name is printed when an illegal bin is encountered. Additional calls to SetName will replace the previous value in SetName.

```
procedure SetName (NameIn : String ) ;
. . .
CovBin1.SetName("DMA Cov") ;            -- first line of heading
```

If the headings need to be cleared, use the method DeallocateName. It is called as follows.

```
CovBin1.DeallocateName ;            -- clears all headings
```

If the internal randomization seed has not yet been initialized, the first call to SetName will initialize the seed using the string value.

If SetName is not set, the first word in the first message (SetMessage) will be used instead.

13.5 Using Files - FileOpenWriteBin, WriteBin, and WriteCovHoles

Since a file parameter cannot be used with WriteBin and WriteCovHoles, either a file must be opened within the coverage model or an awkward set of string and File_Open_Kind parameters must be used.

If every WriteBin or WriteCovHoles writes to the same file, then FileOpenWriteBin can be use to open a file internal to the coverage model. The declaration of FileOpenWriteBin is shown below. When a file is open and WriteBin or WriteCovHoles is called without a file specification, then the opened file is used rather than OUTPUT.

```
procedure FileOpenWriteBin (FileName : string; OpenKind : File_Open_Kind ) ;
```

There is also a corresponding FileCloseWriteBin.

```
procedure FileCloseWriteBin ;
```

If several different files are used with WriteBin or WriteCovHoles, then the file name and open kind can be specified as a parameters. The following overloading is provided.

```
procedure WriteBin (FileName : string; OpenKind : File_Open_Kind := APPEND_MODE) ;
procedure WriteCovHoles ( FileName : string; OpenKind : File_Open_Kind :=
APPEND_MODE ) ;
```

Note, WRITE_MODE initializes and opens a file, so make sure to only use it on the first write to the file. For all subsequent writes to the same file use APPEND_MODE (hence it is the default). The following shows a call to WriteBin followed by a call to WriteCovHoles.

```
--                      FileName,       OpenKind
CovBin1.WriteBin      ("Test1.txt",    WRITE_MODE);
```

```
    CovBin1.WriteCovHoles ("Test1.txt",    APPEND_MODE);
```

14 Coverage Goals and Randomization Weights

Coverage goals and randomization weights are an important part of the Intelligent Coverage methodology. A coverage goal specifies how many times a value must land in a bin before the bin is considered covered. A randomization weight determines the relative number of times a bin will be selected in randomization. In VHDL, each bin within a coverage model may have a different coverage goal and randomization weight.

Up to this point, every coverage bin has a coverage goal of 1 and that value has been used as the randomization weight. However, some tests require coverage goal of other than one and some tests require a randomization weight that is different from the coverage goal. This section addresses how to set coverage goals and randomization weights using overloaded methods and functions in CoveragePkg.

14.1 Specifying Coverage Goals - AddBins, AddCross, and GenBin

A coverage goal can be set by using the AtLeast parameter of AddBins or AddCross. By default this coverage goal will also be used as the randomization weight. The declaration for these is shown below.

```
    procedure AddBins (AtLeast : integer ;  CovBin : CovBinType) ;
    procedure AddCross(
        AtLeast : integer ;
        Bin1, Bin2 : CovBinType ;
        Bin3, Bin4, Bin5, Bin6, Bin7, Bin8, Bin9, Bin10, Bin11, Bin12, Bin13,
        Bin14, Bin15, Bin16, Bin17, Bin18, Bin19, Bin20 : CovBinType := NULL_BIN
    ) ;
```

The GenBin function also has an AtLeast parameter. Its declaration is shown below.

```
    function GenBin(AtLeast, Min, Max, NumBin : integer ) return CovBinType ;
```

If a bin is an ignore or illegal bin, then the coverage goal is set to 0. If a bin is a count bin and a coverage goal is specified in more than one place, then the largest specified value is used. If a bin is a count bin and no coverage goal is specified then the coverage goal is set to 1.

14.2 Selecting Randomization Weights - SetWeightMode

By default, a coverage goal is used as the randomization weight. The coverage weight can also be set to use either a bin weight or remaining coverage as the randomization weight.

Selection of the randomization weight is done using SetWeightMode. The following table lists the current set of supported modes and how the randomization weight is calculated.

Mode	Weight
AT_LEAST	AtLeast
WEIGHT	Bin Weight
REMAIN	AtLeast - Count *
* Note AtLeast is adjusted if the coverage target /= 100 %	

The interface for procedure SetWeightMode is shown below.

```
type WeightModeType  is (AT_LEAST, WEIGHT, REMAIN) ;
procedure SetWeightMode (A : WeightModeType) ;
```

Note that there are additional undocumented features on SetWeightMode and WeightModeType. Use these at your own risk as they are subject to change in each revision. Their names and implementation were changed in revision 2013.04. If there is one that you have tried that is working better than documented options, please let me know.

14.3 Specifying Bin Weight - AddBins, AddCross, and GenBin

A bin's weight is used as the randomization weight when the WeightMode WEIGHT is selected by SetWeightMode. A bin's weight can be set by using the Weight parameter of AddBins or AddCross. If not specified, a bin's weight value will be 1. The declaration for these is shown below. Note this use of the Weight parameter also requires a coverage goal to be specified.

```
procedure AddBins (AtLeast, Weight  : integer ;  CovBin : CovBinType) ;
procedure AddCross(
    AtLeast : integer ;
    Weight  : integer ;
    Bin1, Bin2 : CovBinType ;
    Bin3, Bin4, Bin5, Bin6, Bin7, Bin8, Bin9, Bin10, Bin11, Bin12, Bin13,
    Bin14, Bin15, Bin16, Bin17, Bin18, Bin19, Bin20 : CovBinType := NULL_BIN
) ;
```

The GenBin function also has an Weight parameter. Its declaration is shown below.

```
function GenBin(AtLeast, Weight, Min, Max, NumBin : in integer)
      return CovBinType ;
```

If a bin is an ignore or illegal bin, then the bin weight is set to 0. If a bin is a count bin and a bin weight is specified in more place, then the largest specified value is used. If a bin is a count bin and no bin weight is specified then the coverage goal is set to 1. Likewise for the coverage goal.

15 Coverage Targets

For some tests, the AtLeast parameters will be used to set an initial coverage distribution. Later it may be desirable to use the same coverage distribution, but run it for much longer. Use of a coverage target allows the coverage goal to be scaled (increased or decreased) without having to change anything else in the coverage model. Hence, the effective coverage goal for a bin is the product of bin's AtLeast least value and the coverage model's coverage target value (specifically, AtLeast * CovTarget / 100.0).

15.1 Setting a Coverage Target - SetCovTarget

The method SetCovTarget sets the coverage model's coverage target (internally the CovTarget variable).

```
procedure SetCovTarget (Percent : real) ;
```

The coverage target is intended to scale the run time of a test without having to change a bin's AtLeast values. CovTarget is set to 100.0 initially. Setting the coverage target to 1000.0 will increase the run time 10X. Setting the coverage target to 50.0 will decrease the run time by 2X.

The versions of the following methods that do not have a PercentCov parameter use the CovTarget value: RandCovPoint, RandCovBinVal, IsCovered, and WriteCovHoles.

15.2 Overriding the Global Coverage Target - PercentCov

The methods that use CovTarget also have a version with a PercentCov parameter that overrides the CovTarget value. The following methods have a PercentCov parameter.

```
impure function RandCovPoint (PercentCov : real) return integer_vector ;
impure function RandCovPoint (PercentCov : real) return integer ;
impure function RandCovBinVal (PercentCov : real) return RangeArrayType ;
impure function IsCovered (PercentCov : real) return boolean ;
procedure WriteCovHoles ( PercentCov : real) ;
procedure WriteCovHoles ( FileName : string;  PercentCov : real ;
                OpenKind : File_Open_Kind := APPEND_MODE ) ;
```

16 Randomization Thresholds - SetThresholding and SetCovThreshold

Ordinarily randomization (using RandCovPoint or RandCovBinVal) can select any bin whose coverage target has not been reached. Thresholding modifies this by also excluding bins whose coverage exceeds the minimum coverage plus the threshold value (MinCov + threshold). Thresholding is intended to balance how a test converges to coverage closure. Thresholding only has meaning when coverage goals (AtLeast * CovTarget/100.0) are greater than 1.

The threshold value is set using SetCovThreshold. Thresholding is enabled by either SetCovThreshold or SetThresholding.

```
procedure SetThresholding (A : boolean := TRUE ) ;
procedure SetCovThreshold (Percent : real) ;
```

By setting a coverage threshold of 0.0, the notion of cyclic randomization is extended to work across a coverage model.

17 Handling Overlapping Bins

17.1 LastIndex - Count bins overlapping with other counts

When RandCovPoint or RandCovBinVal is called, the bin index that generates it is logged in the LastIndex variable. When ICover is called, it searches for the value in the bin whose index is currently stored in the LastIndex variable. This way if bins overlap, it insures that the bin that generated the value is the bin whose count value is incremented.

17.2 Bin Merging

17.2.1 Count Bins Contained in an Illegal or Ignore Bin

Bin merging is an experimental feature that drops a count bin if it is contained in a previously defined ignore or illegal bin. Merging is off by default and can be enabled or disabled with the SetMerging procedure shown below.

```
procedure SetMerging (A : boolean := TRUE ) ;
```

Currently bin merging also merges count bins when they have identical bin values. Merging of count bins is expensive. Since this feature is correctly handled by LastIndex, it may be removed in the future. If you need count bins to be merged, please contact the package author.

17.2.2 Count Bins Overlapping with an Illegal or Ignore Bin

Count bins overlapping with a previous ignore or illegal bin are problematic. When the count bin is selected for randomization, it may generate an illegal value due to the overlap.

This may be addressed in a future version. For now it is up to the user understand overlap and to avoid this.

17.3 Multiple Matches with ICover - SetCountMode

By default, ICover searches for the point in the bin pointed to by LastIndex. If not found there, it searches the bins in order. This mode should satisfy most use models.

SetCountMode is an experimental feature that can be used to change the default behavior. SetCountMode sets the internal CountMode variable. The default mode, described above, is COUNT_FIRST. If the CountMode is set to COUNT_ALL, each matching bin is counted. The following shows how to set the CountMode.

```
type CountModeType   is (COUNT_FIRST, COUNT_ALL) ;
CovBin4.SetCountMode(COUNT_ALL) ;    -- Count all matching bins
CovBin4.SetCountMode(COUNT_FIRST) ;  -- default.   Only count first matching bin
```

Caution: this experimental feature may be removed from future versions if it impacts run time. If you have need for COUNT_ALL, please contact the package author.

18 Initializing the Seeds - InitSeed, SetSeed, and GetSeed

Intelligent coverage uses pseudo random number generation as its basis. As such, for a given randomization seed value it will generate the same sequence of numbers every time a simulation is run. This is important as it means that when a bug is found and fixed, the fix can be validated since the same test sequence that caused the bug will be generated.

On the other hand, it also means that if a design has two identical interfaces and the testbench uses the two identical coverage models to generate tests that they will both see the same test sequence. This is not desirable since it is unlikely to generate interesting interactions between the two interfaces. As a result, it is desirable that each coverage model is given a different initial seed value. This is simple to do.

The InitSeed method initializes a coverage model's internal randomization seed. The following example shows the method overloading and an example call. One easy way to generate a unique seed value for each coverage bin is to use the string value generated by 'path_name applied to the coverage object as shown. Note that the method SetMessage will also call InitSeed with its parameter if the seed is not already set.

```
procedure InitSeed (S  : string ) ;
procedure InitSeed (I  : integer ) ;
. . .
CovBin1.InitSeed( CovBin1'path_name ) ; -- string
```

The methods GetSeed and SetSeed are intended for saving and restoring the seeds. In this case the seed value is of type RandomSeedType, which is defined in RandomBasePkg. RandomBasePkg also defines procedures for reading and writing RandomSeedType values (see RandomPkg Users Guide for details).

```
procedure SetSeed (RandomSeedIn : RandomSeedType ) ;
impure function GetSeed return RandomSeedType ;
```

Note that the time it takes to achieve coverage closure with open loop randomization methods, such as SystemVerilog's constrained random, may depend heavily on the initial seed value. Hence, within the SystemVerilog community some may try out different seeds when running simulations just to see if they can improve run times. This is not necessary with OSVVM's Intelligent Coverage methodology since it only selects from coverage holes.

19 Interacting with the Coverage Data Structure

19.1 Basic Bin Information

```
impure function GetNumBins return integer ;
impure function GetMinIndex return integer ;
impure function GetMinCov return real ;
impure function GetMinCount return integer ;
impure function GetMaxIndex return integer ;
impure function GetMaxCov return real ;
impure function GetMaxCount return integer ;
impure function GetErrorCount return integer ;
```

The function GetNumBins returns the number of bins in the coverage model. Bin values are numbered from 1 to NumBins.

The functions GetMinIndex and GetMaxIndex return the index of the first bin with the minimum and maximum percent coverage of a bin. The functions GetMinCov and GetMaxCov return the minimum and maximum percent coverage of a bin. The functions GetMinCount and GetMaxCount return the minimum and maximum count in a bin.

The function GetErrorCount sums up the count in each of the error bins and returns the resulting value. Generally GetErrorCount is called at the end of a testbench for coverage models that have bins marked as illegal.

```
TestErrCount := CovBin1.GetErrorCount + (Other_Error_Sources) ;
```

19.2 Getting Coverage Point Values

In addition to RandCovPoint, there are also the following that return coverage value that is within a particular coverage bin. Integer return values are for bins with a single item. Integer_vector are for single item or cross coverage bins.

```
impure function GetPoint ( BinIndex : integer ) return integer ;
impure function GetPoint ( BinIndex : integer ) return integer_vector ;
impure function GetMinPoint return integer ;
impure function GetMinPoint return integer_vector ;
impure function GetMaxPoint return integer ;
impure function GetMaxPoint return integer_vector ;
```

The function GetPoint returns a random point within the addressed bin (BinIndex). The functions GetMinPoint and GetMaxPoint return a random point within the first bin with minimum and maximum percent coverage.

19.3 Getting Coverage Bin Values

In addition to RandCovBinVal, there are also the following that return coverage bin value. The return value has type RangeArrayType.

```
impure function GetBinVal ( BinIndex : integer ) return RangeArrayType ;
impure function GetMinBinVal  return RangeArrayType ;
impure function GetMaxBinVal  return RangeArrayType ;
```

The function GetBinVal returns the bin value of the addressed bin (BinIndex). The functions GetMinBinVal and GetMaxBinVal return the bin value of the first bin with minimum and maximum percent coverage.

19.4 Getting Last Randomization Information

The method GetLastIndex returns the index value the bin last selected for randomization. The method GetLastBinVal returns the bin value of the bin indexed by LastIndex. The overloading is as follows.

```
impure function GetLastIndex  return integer ;
impure function GetLastBinVal  return RangeArrayType ;
```

19.5 Getting Coverage Holes

The following functions return information about coverage holes.

```
impure function CountCovHoles return integer ;
impure function CountCovHoles (PercentCov : real) return integer ;
impure function GetHoleBinVal(ReqHoleNum : integer := 1) return RangeArrayType ;
impure function GetHoleBinVal(PercentCov : real) return RangeArrayType ;
impure function GetHoleBinVal(ReqHoleNum : integer ; PercentCov : real)
  return RangeArrayType ;
```

The function CountCovHoles returns the number of holes that are below the PercentCov parameter. CountCovHoles without a PercentCov parameter returns the number of holes that are below the CovTarget value.

```
--                                  PercentCov
NumHoles   := CovBin1.CountCovHoles( 100.0 ) ;
```

GetHoleBinVal gets the ReqHoleNum bin with a coverage value less than the PercentCov value. The following call to GetHoleBinVal gets the 5th bin that has less than 100% coverage. Note that ReqHoleNum must be between 1 and CountCovHoles. GetHoleBinVal without a PercentCov parameter uses CovTarget in its place.

```
--                               ReqHoleNum,   PercentCov
TestData := CovBin1.GetHoleBinVal(     5,          100.0  ) ;
```

20 Coverage Database Operations

A coverage model can be written and read using WriteCovDb and ReadCovDb. Using these allows results to be accumulated across multiple tests, and hence, things like test configurations can be covered and randomized. The method declarations are shown below. Like WriteBin file parameters cannot be used, so WriteCovDb and ReadCovDb use parameters that specify the file name as a string and the file open mode.

```
procedure WriteCovDb (FileName : in string;
                      OpenKind : File_Open_Kind := WRITE_MODE ) ;
procedure ReadCovDb (FileName : string ; Merge : boolean := FALSE) ;
```

WriteCovDb saves the coverage model and internal variables into a file. The following shows a call to WriteCovDb. Generally WriteCovDb is called once per test. As a result, WRITE_MODE is the default.

```
--                   FileName,   OpenKind
CovBin1.WriteCovDb( "CovDb.txt", WRITE_MODE ) ;
```

The procedure method ReadCovDb reads the coverage model and internal variables from a file. If the optional Merge parameter is set to TRUE, the values read will be merged with the current coverage model. The following shows a call to ReadCovDb.

```
--                  FileName
CovBin1.ReadCovDb( "CovDb.txt", TRUE );
```

21 Bin Clearing and Deconstruction

The procedure SetCovZero sets all the coverage counts in a coverage bin to zero. This allows the counts to be set to zero after reading in a coverage database. A simple call to it is shown below.

```
CovBin1.SetCovZero ;  -- set all counts to 0
```

The procedure Deallocate deallocates the entire database structure and sets the internal variables back to their defaults.

```
CovBin1.Deallocate ;
```

22 Creating Bin Constants

Constants are used for two purposes. The first is to create a short hand name for an item (point) bin (normal constant stuff) and then use that name later in composing the coverage model. The second is to create the entire coverage model in the constant to facilitate reuse of the model.

22.1 Item (Point) Bin Constants - CovBinType

In a previous model, we constructed a cross coverage model using the following call to AddCross.

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

One step of refinement is to create an item bin constant for the register addresses, such as REG_ADDR shown below. The type of REG_ADDR is CovBinType. Since constants can extract their range based on the object assigned to them, it is easiest to leave CovBinType unconstrained.

```
constant REG_ADDR : CovBinType := GenBin(0, 7) ;
```

Once created the constant can be used in for further composition, such as shown below. Just like normal constants, this increases both the readability and maintainability of the code.

```
ACov.AddCross(REG_ADDR, REG_ADDR); -- Model
```

Since each element in an item bin may require different coverage goals or weights, additional overloading of GenBin were added. These are shown below.

```
function GenBin(AtLeast, Weight, Min, Max, NumBin : integer ) return CovBinType ;
function GenBin(AtLeast, Min, Max, NumBin : integer ) return CovBinType ;
```

As demonstrated earlier, item bins can be composed using concatenation. The following example creates two bins: 0 to 31 with coverage goal of 5, and 32 to 63 with coverage goal of 10.

```
constant A_BIN : CovBinType := GenBin(5, 0, 31, 1) & GenBin(10, 32, 63, 1) ;
```

22.2 Writing an Cross Coverage Model as a Constant - CovMatrix?Type

To capture a cross coverage model in a constant requires some additional types and functions. The following methodology is based the language prior to VHDL-2008 and requires a separate type definition for each size of cross coverage model. Currently up to a cross product of 9 separate items are supported by the following type. In VHDL-2008 where composites are allowed to have unconstrained elements, this will be reduced to a single type (and cross products of greater than 9 can be easily supported).

```
type CovMatrix2Type is array (natural range <>) of CovMatrix2BaseType;
type CovMatrix3Type is array (natural range <>) of CovMatrix3BaseType;
type CovMatrix4Type is array (natural range <>) of CovMatrix4BaseType;
type CovMatrix5Type is array (natural range <>) of CovMatrix5BaseType;
type CovMatrix6Type is array (natural range <>) of CovMatrix6BaseType;
type CovMatrix7Type is array (natural range <>) of CovMatrix7BaseType;
type CovMatrix8Type is array (natural range <>) of CovMatrix8BaseType;
type CovMatrix9Type is array (natural range <>) of CovMatrix9BaseType;
```

The function GenCross is used to generate these cross products. We need a separate overloaded function for each of these types. The interface that generates CovMatrix2Type and CovMatrix9Type are shown below.

```
function GenCross(  -- cross 2 item bins - see method AddCross
  constant AtLeast : integer ;
  constant Weight  : integer ;
  constant Bin1, Bin2 : in CovBinType
) return CovMatrix2Type ;

function GenCross(AtLeast : integer ; Bin1, Bin2 : CovBinType)
  return CovMatrix2Type ;
function GenCross(Bin1, Bin2 : CovBinType) return CovMatrix2Type ;

function GenCross(  -- cross 9 item bins - intended only for constants
  constant AtLeast : integer ;
  constant Weight  : integer ;
  constant Bin1, Bin2, Bin3, Bin4, Bin5, Bin6, Bin7, Bin8, Bin9 : in CovBinType
) return CovMatrix9Type ;
function GenCross(
  AtLeast : integer ;
  Bin1, Bin2, Bin3, Bin4, Bin5, Bin6, Bin7, Bin8, Bin9 : CovBinType
) return CovMatrix9Type ;
function GenCross(
  Bin1, Bin2, Bin3, Bin4, Bin5, Bin6, Bin7, Bin8, Bin9 : CovBinType
) return CovMatrix9Type ;
```

Now we can write our constant for our simple ALU coverage model.

```
constant ALU_COV_MODEL : CovMatrix2Type := GenCross(REG_ADDR, REG_ADDR);
```

When we want to add this to our coverage data structure, we need methods that handle types CovMatrix2Type through CovMatrix9Type. This is handled by the overloaded versions of AddBins shown below.

```
procedure AddBins (CovBin : CovMatrix2Type) ;
procedure AddBins (CovBin : CovMatrix3Type) ;
procedure AddBins (CovBin : CovMatrix4Type) ;
procedure AddBins (CovBin : CovMatrix5Type) ;
procedure AddBins (CovBin : CovMatrix6Type) ;
procedure AddBins (CovBin : CovMatrix7Type) ;
procedure AddBins (CovBin : CovMatrix8Type) ;
procedure AddBins (CovBin : CovMatrix9Type) ;
```

To create the coverage data structure for the simple ALU coverage model, call AddBins as shown below.

```
ACov.AddBins( ALU_COV_MODEL ); -- Model
```

This capability is here mostly due to evolution of the package. Keep in mind, the intent is to create readable and perhaps reusable coverage models.

GenCross also allows specification of weights. In a similar manner to AddCross, we can build up our coverage model incrementally using constants and concatenation. This is shown in the following example.

```
architecture Test4 of tb is
  shared variable ACov : CovPType ;                -- Declare Cov Object
  constant ALU_BIN_CONST : CovMatrix2Type :=
      GenCross(1, GenBin (0), GenBin(1,7)) &
      GenCross(2, GenBin (1), GenBin(0)   & GenBin(2,7)) &
      GenCross(3, GenBin (2), GenBin(0,1) & GenBin(3,7)) &
      GenCross(4, GenBin (3), GenBin(0,2) & GenBin(4,7)) &
      GenCross(5, GenBin (4), GenBin(0,3) & GenBin(5,7)) &
      GenCross(6, GenBin (5), GenBin(0,4) & GenBin(6,7)) &
      GenCross(7, GenBin (6), GenBin(0,5) & GenBin(7)) &
      GenCross(8, GenBin (7), GenBin(0,6) ) ;
begin
  TestProc : process
    variable RegIn1, RegIn2 : integer ;
  begin
    -- Capture coverage model
    ACov.AddBins( ALU_BIN_CONST ) ;

    while not ACov.IsCovered loop                  -- Interact
      -- Randomize register addresses -- see RandomPkg documentation
      (RegIn1, RegIn2) := ACov.RandCovPoint ;

      DoAluOp(TRec, RegIn1, RegIn2) ;              -- Do a transaction
      ACov.ICover( (RegIn1, RegIn2) ) ;            -- Accumulate
    end loop ;

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

23 Reuse of Coverage

There are a couple of ways to reuse a coverage model. If the intent is to reuse and accumulate coverage across tests, then the only way to accomplish this is to use WriteCovDb and ReadCovDb. If the intent is to just reuse the coverage model itself, then either a constant or a subprogram can be used. The calls to ICover generally are simple enough that we do not try to abstract them.

24 Compiling

We compile CoveragePkg into a library named "osvvm". Be sure to use the VHDL-2008 switch when you compile it. Your programs will need to reference CoveragePkg as follows.

```
library osvvm ;
  use osvvm.CoveragePkg.all ;
```

25 CoveragePkg vs. Language Syntax

The basic level of item (point) coverage that can be captured with CoveragePkg is similar to when can be captured with IEEE 1647, 'e'. CoveragePkg and 'e' allow an item bin to consist of either a single value or a single range. SystemVerilog extends this to allow a value, a range, or a collection of values and ranges. While this additional capability of SystemVerilog is interesting, it did not seem to offer any compelling advantage that would justify the additional complexity required to specify it to the coverage model.

For cross coverage, both SystemVerilog and 'e' focus on first capturing item coverage and then doing a cross of the items. There is some capability to modify the bins contents within the cross, but at best it is awkward. On the other hand, CoveragePkg allows one to directly capture cross coverage, bin by bin and incrementally if necessary. Helper functions are provided to simplify the process. This means for simple things, such as making sure every register pair of an ALU is used, the coverage is captured in a very concise syntax, however, when more complex things need to be done, such as modeling the coverage for a CPU, the cross coverage can be captured on a line by line basis.

As a result, with CoveragePkg it is easier to capture high fidelity coverage within a single coverage object. A high fidelity coverage model in a single coverage object is required to do Intelligent Coverage.

26 Deprecated Methods

In the original design of the coverage feedback and randomization functions, there was no coverage goal or weight. Instead, each bin has a weight of 1 and the coverage goal is determined by the AtLeast parameter in the function calls. These functions are shown below. In this implementation, all bins had the same coverage goal. Usage of the AtLeast parameter has been subsumed by the real valued PercentCov parameter. In addition, each bin now has the capability to have a different coverage goal and weight. With different coverage goal values, PercentCov has replaced the AtLeast parameter. The functionality of the AtLeast parameter has been subsumed by has been subsumed by the PercentCov parameter. A PercentCov parameter of 200.0 is equivalent to an AtLeast parameter of 2.

```
impure function GetMinCov return integer ;
impure function GetMaxCov return integer ;
impure function CountCovHoles ( AtLeast : integer ) return integer ;
impure function IsCovered ( AtLeast : integer ) return boolean ;
impure function GetHoleBinVal ( ReqHoleNum : integer := 1 ; AtLeast : integer )
    return RangeArrayType ;
impure function RandCovBinVal ( AtLeast : in integer ) return RangeArrayType ;
impure function RandCovPoint (AtLeast : in integer ) return integer_vector ;

procedure WriteCovHoles ( AtLeast : in integer ) ;
procedure WriteCovHoles ( FileName : string;  AtLeast : in integer ; OpenKind :
File_Open_Kind := APPEND_MODE ) ;
```

27 Future Work

CoveragePkg.vhd is a work in progress and will be updated from time to time.

Some of the plans for the next revision are:

- Revise bin merging. It is still an experimental feature and is off by default.
- Add a global coverage settings protected type.
  - Add capability for global on/off for coverage collection.
  - Set defaults for CountMode, IllegalMode, WeightMode, CovThresholdPercent in the global coverage model.
- Remove OrderCount (was for development purposes only).
- Consider overloading AddBins to subsume AddCross - that way AddBins is the only method needed to create the coverage data structure.

If you have ideas that you would like to see, please contact me at jim@synthworks.com.

28 Other Packages - RandomPkg

CoveragePkg is part of the Open Source VHDL Verification Methodology (OSVVM) packages. In addition to the CoveragePkg, our randomization packages (RandomPkg, RandomBasePkg, SortListPkg_int) are in OSVVM. The most current versions are always available at http://www.SynthWorks.com/downloads. Over time we will also be releasing other packages. With time, we hope simulation vendors will distribute the OSVVM libraries with their tools.

29 About CoveragePkg

CoveragePkg was developed and is maintained by Jim Lewis of SynthWorks VHDL Training. It evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and verification class. It is part of the Open Source VHDL Verification Methodology (OSVVM), which brings leading edge verification techniques to the VHDL community.

Please support our effort in supporting CoveragePkg and OSVVM by purchasing your VHDL training from SynthWorks.

CoveragePkg is released under the Perl Artistic open source license. It is free (both to download and use - there are no license fees). You can download it from http://www.synthworks.com/downloads. It will be updated from time to time. Currently there are numerous planned revisions.

If you add features to the package, please donate them back under the same license as candidates to be added to the standard version of the package. If you need features, be sure to contact us. I blog about the packages at http://www.synthworks.com/blog. We also support a user community and blogs through http://www.osvvm.org.

Release notes are in the document CoveragePkg_release_notes.pdf.

30 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has twenty-eight years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue. Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.

31 References

[1] Jim Lewis, VHDL Testbenches and Verification, student manual for SynthWorks' class.

[2] Andrew Piziali, Functional Verification Coverage Measurement and Analysis, Kluwer Academic Publishers 2004, ISBN 1-4020-8025-5

[3] IEEE Standard for System Verilog, 2005, IEEE, ISBN 0-7381-4811-3

[4] IEEE 1647, Standard for the Functional Verification Language 'e', 2006

[5] A Fitch, D Smith, "Functional Coverage - without SystemVerilog!", DVCON 2010

32 When Code Coverage Fails

While code coverage is generally a useful metric, there are some cases where it does not accomplish what we want.

To help understand the issue, consider the following process. If SelA, SelB, and SelC all are 1 when Clk rises, then all of the lines of code execute and the code coverage is 100%. However, only the assignment, "Y <= A" has an observable impact on the output. The assignments, "Y <= C" and Y <= B" are not observable, and hence, are not validated.

```
PrioritySel : process (Clk)
begin
  if rising_edge(Clk) then
    Y <= "00000000" ;
    if (SelC = '1') then
      Y <= C ;
    end if ;
    if (SelB = '1') then
      Y <= B ;
    end if ;
    if (SelA = '1') then
      Y <= A ;
    end if ;
  end if ;
end process ;
```

In combinational logic, this issue only becomes worse. If we change the above process as shown below, then it runs due to any change on its inputs. It still has the issues shown above. In addition, the process now runs and accumulates coverage based on any signal change. Signals may change multiple times during a given clock period due to differences in delays - either delta cycle delays (in RTL) or real propagation delays (in gate simulations or from external models).

```
PrioritySel : process (SelA, SelB, SelC, A, B, C)
begin
  Y <= "00000000" ;
  if (SelC = '1') then
    Y <= C ;
  end if ;
  if (SelB = '1') then
    Y <= B ;
  end if ;
  if (SelA = '1') then
    Y <= A ;
  end if ;
end process ;
```

Since functional coverage depends on observing conditions in design, it may cover all of the gaps. There are also additional tools that address this issue with code coverage.