Functional Coverage Using CoveragePkg

User Guide for Release 2024.03

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 an API that provides a powerful capability. This API is documented in the remaining part of the document.

Caution, the package also contains additional undocumented subprograms that are either experimental features or artifacts from older use models. 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' Advanced VHDL Testbenches and Verification class. Please support our effort in supporting OSVVM by purchasing your VHDL training from SynthWorks.

All CoveragePkg uses protected types (VHDL-2002) and integer_vector (VHDL-2008). As a result, most simulators should be able to support it.

Please note while the packages are Apache open source, this document's copyright is similar to the creative commons license CC BY-ND 4.0. You may use and distribute exact copies of this document. However, without written permission, the copyright does not permit you to use these materials (in particular examples) to develop your own training – lets be fair we have to earn a living.

2 Getting CoveragePkg

CoveragePkg is released under the Apache open source license. It is free (both to download and use - there are no license fees). You can download it from https://github.com/OSVVM/OSVVM. 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 and do a pull request. 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.

This package requires VHDL-2008.

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 an API 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 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.

5 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.

6 Basic Item (Point) Coverage with CoveragePkg

In this section 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

- Reference Packages
- Declare a Coverage ID
- Construct the Coverage Model
- Model Coverage
- Accumulate coverage
- Detecting when Done
- Print Results

6.1 Reference Packages

The simplest way to include OSVVM utility library in your tests is reference the OSVVM context declaration.

```
library OSVVM ;
  context osvvm.OsvvmContext ;
```

6.2 Declare a Coverage ID

A Coverage ID is a handle to the coverage object that is created internal to CoveragePkg. A Coverage ID has the type CoverageIDType. CoverageIDType is a regular type and can be held in a signal or variable.

```
architecture Test1 of tb is
  signal Cov1 : CoverageIDType ;
begin
```

6.3 Construct the Coverage Model

NewID allocates the initial coverage data structure. The data structure must be initialized before any other coverage API operation is used.

```
Cov1 <= NewID("Cov1") ;
wait for 0 ns ; -- Update Cov1
```

6.4 Model Coverage: Item Coverage

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 items are added to the coverage model by using the procedure AddBins and the function GenBin. GenBin transforms a bin descriptor into a set of bins. AddBins inserts these bins into the coverage data structure. 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
  AddBins(Cov1, GenBin(1,   3,   3)); -- bins 1 to 1, 2 to 2, 3 to 3
  . . .
```

Additional calls to AddBins appends 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.

```
AddBins(Cov1, 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.

```
AddBins(Cov1, GenBin(253, 255)) ; -- bins 253, 254, 255
```

6.5 **Accumulate Coverage: Item Coverage**

Coverage is accumulated using the 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' ;
  ICover(Cov1, to_integer(RxData_slv)) ;
end loop ;
```

6.6 **Detecting when Done**

A test is done when functional coverage reaches 100%. 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%
loop
```

```
    wait until rising_edge(Clk) and nReset = '1' ;
    ICover(Cov1, to_integer(RxData_slv)) ;
    exit when IsCovered(Cov1) ;
end loop ;
```

6.7 **Print Results**

Finally, when the test is done, WriteBin is used to print the coverage results to the file specified by TranscriptPkg (either TranscriptFile, OUTPUT, or both if mirroring is enabled).

```
WriteBin(Cov1) ;
```

6.8 **Item Coverage, Complete Example**

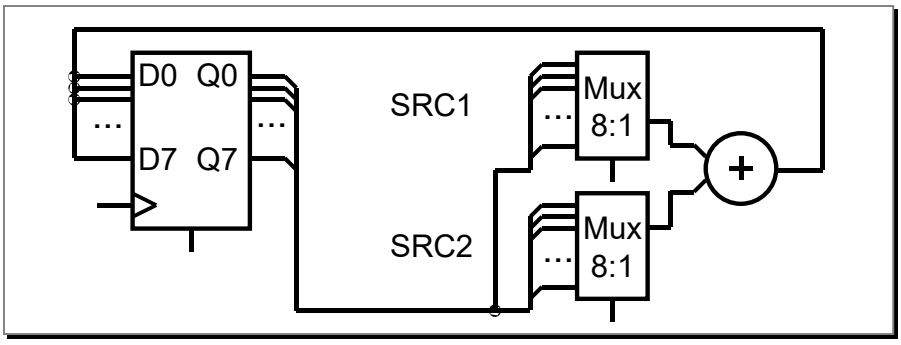
Putting the entire example together, we end up with the following.

```
architecture Test1 of tb is
  signal Cov1 : CoverageIDType ;  -- Coverage ID
begin
  TestProc : process
  Begin
    -- Construct the coverage model
    Cov1 <= NewID("Cov1") ;
    wait for 0 ns ; -- Update Cov1

    -- Model the coverage
    AddBins(Cov1, GenBin(  1,   3     )) ;
    AddBins(Cov1, GenBin(  4, 252, 2)) ;
    AddBins(Cov1, GenBin(253, 255   )) ;

    -- Accumulating Coverage
    -- clock based sampling
    loop
      wait until rising_edge(Clk) and nReset = '1' ;
      ICover(Cov1, to_integer(RxData_slv)) ;
      exit when IsCovered(Cov1) ;
    end loop ;

    -- Print Results
    WriteBin(Cov1) ;
    wait ;
  end process ;
```

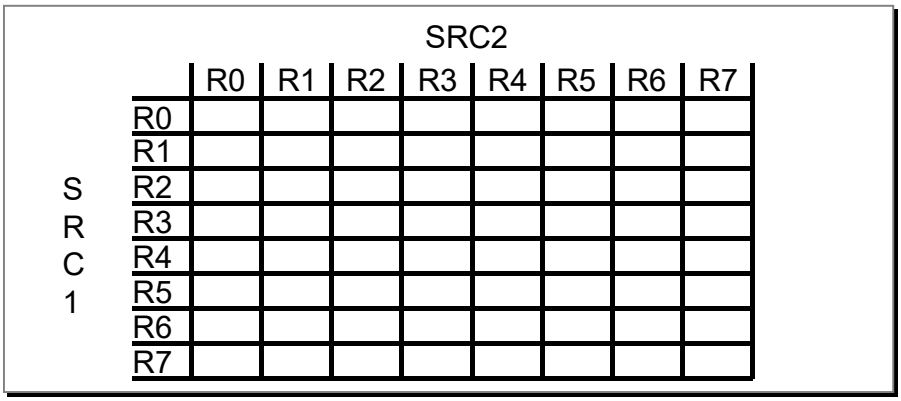
Note that when modeling coverage, we primarily work with integer values. All of the inputs to GenBin and ICover are integers; WriteBin reports results in terms of integers. This is similar to what other verification languages do.

7 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, construct, model, accumulate, detect done, and report. Collecting cross coverage only differs in the model and accumulate steps.

7.1 Model Coverage: Cross Coverage

Cross coverage is modeled using 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.

```
AddCross(ACov, 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).

7.2 Accumualte Coverage: Cross Coverage

The accumulate step now requires a value for SRC1 and SRC2. The overloaded ICover 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.

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

7.3 Cross Coverage, Complete

The code below shows the entire example. The signal, ACov, declares the coverage ID. NewID constructs the coverage model. AddCross adds cross coverage items to the coverage model. Each register is selected using uniform randomization (RandInt). The transaction procedure, DoAluOp, applies the stimulus. ICover accumulates the coverage. IsCovered determines when all items in the coverage model have been covered. WriteBin reports the coverage.

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

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

      DoAluOp(TRec, Src1, Src2) ;    -- Transaction
      ICover(ACov, (Src1, Src2) ) ;  -- Accumulate
      exit when IsCovered(ACov) ;    -- Done?
    end loop ;

    WriteBin(ACov) ;  -- Report
    ReportAlerts ;
  end process ;
```

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

8.1 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 – whether it is VHDL or other (SystemVerilog, e, …).

```
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 test cases 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.

SRC1 \ SRC2	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

8.2 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 GetRandPoint (one of the Intelligent Coverage randomization functions). 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
  signal ACov : CoverageIDType ;  -- Declare ID
begin
  TestProc : process
    variable RV : RandomPType ;
    variable Src1, Src2 : integer ;
  begin
    -- create coverage model
    AddCross(ACov, GenBin(0,7), GenBin(0,7) );  -- Model

    loop
      (Src1, Src2) := GetRandPoint(ACov) ; -- Intelligent Coverage Randomization

      DoAluOp(TRec, Src1, Src2) ;    -- Transaction
      ICover(ACov, (Src1, Src2) ) ;  -- Accumulate
      exit when IsCovered(ACov) ;    -- Done?
    end loop ;

    WriteBin(ACov) ;  -- Report
    ReportAlerts ;
  end process ;
```

When randomizing across a cross coverage model, the output of GetRandPoint 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 := GetRandPoint(ACov) ;      -- Intelligent Coverage Randomization
```

The process is not always this easy. Sometimes the value out of GetRandPoint will need to be further shaped by the stimulus generation process. This is particularly true randomizing an output value of a design and generating the desired input sequence to generate it.

The Intelligent Coverage methodology 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.

8.3 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.

9 Flexibility and Capability

OSVVM implements functional coverage as a data structure within a protected type. The protected type is hidden from view by a singleton that is accessed via the coverage API. Using subprograms of the coverage API 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 illegal for SRC1 to equal SRC2. 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

Coverage Goal	Src1	Src2
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
  signal ACov : CoverageIDType ;                -- Declare Cov Object
begin
  TestProc : process
    variable Src1, Src2 : integer ;
  begin
    ACov <= NewID("ACov") ;
    wait for 0 ns ; -- let ACov update

    -- Capture coverage model
    AddCross(ACov,  2,  GenBin (0),  IllegalBin(0) & GenBin(1,7)) ;
    AddCross(ACov,  3,  GenBin (1),  GenBin(0)   & IllegalBin(1) & GenBin(2,7)) ;
    AddCross(ACov,  4,  GenBin (2),  GenBin(0,1) & IllegalBin(2) & GenBin(3,7)) ;
    AddCross(ACov,  5,  GenBin (3),  GenBin(0,2) & IllegalBin(3) & GenBin(4,7)) ;
    AddCross(ACov,  5,  GenBin (4),  GenBin(0,3) & IllegalBin(4) & GenBin(5,7)) ;
    AddCross(ACov,  4,  GenBin (5),  GenBin(0,4) & IllegalBin(5) & GenBin(6,7)) ;
    AddCross(ACov,  3,  GenBin (6),  GenBin(0,5) & IllegalBin(6) & GenBin(7)) ;
    AddCross(ACov,  2,  GenBin (7),  GenBin(0,6) & IllegalBin(7) ) ;

    loop              -- Done?
      (Src1, Src2) := GetRandPoint(ACov) ;      -- Randomize with coverage

      DoAluOp(TRec, Src1, Src2) ;              -- Do a transaction
      ICover(ACov, (Src1, Src2) ) ;            -- Accumulate
      exit when IsCovered(ACov) ;              -- Done?
    end loop ;

    WriteBin(ACov) ;                            -- Report
    ReportAlerts ;
  end process ;
```

Note that the remainder of this document covers further details of the coverage API.

10 Coverage API Reference

The intent of the API is to make the creation of a coverage models easy. It is interesting to note that using the Coverage API is as concise as the language features of SystemVerilog. Going beyond that OSVVM has coverage introspection capability, such as intelligent coverage randomization, that is not available in SystemVerilog.

Behind the simple API are some rather interesting data structures. The important thing to note is that you do not need to understand the data structures to be able use CoveragePkg.

Caution, features not documented here are either considered experimental or have been deprecated (and may be removed in a future version). If you decide to use it and want it to stay around, be sure to let us know.

11 Constructing the Coverage Model

11.1 NewID and Supporting Types

11.1.1 NewID: Data Structure Constructor

NewID allocates and initializes the scoreboard data structure. It must be called before any other scoreboard operations are used.

```
impure function NewID (
  Name                : String ;
  ParentID            : AlertLogIDType          := OSVVM_COVERAGE_ALERTLOG_ID ;
  ReportMode          : AlertLogReportModeType  := ENABLED ;
  Search              : NameSearchType          := NAME_AND_PARENT_ELSE_PRIVATE ;
  PrintParent         : AlertLogPrintParentType := PRINT_NAME_AND_PARENT
) return CoverageIDType ;
```

NewID also creates an AlertLogID for the data structure using the value of the name parameter and initializes the randomization seed that is part of the coverage model (see InitSeed).

11.1.2 CoverageIDType

CoverageIDType is defined as follows.

```
type CoverageIDType is record
  ID : integer_max ;
end record CoverageIDType;
```

11.1.3 **AlertLogReportModeType**

```
type AlertLogReportModeType is (DISABLED, ENABLED, NONZERO) ;
```

When DISABLED, an AlertLogID does not print in ReportAlerts (text reports) or WriteAlertYaml (Yaml/HTML reports). When NONZERO, an AlertLogID does not print in ReportAlerts, but prints in WriteAlertYaml. Hence, NONZERO shortens the text reports to what is essential. When ENABLED, an AlertLogID prints in ReportAlerts and WriteAlertYaml.

None of these settings impact printing in Alert or Log.

DISABLED is intended to be used for data structures that are not used for self-checking and may produce errors on parameter checks. These errors are not lost as they are still accumulated in the parent ID.

11.1.4 **AlertLogPrintParentType**

```
type AlertLogPrintParentType is (PRINT_NAME, PRINT_NAME_AND_PARENT) ;
```

When PRINT_NAME, Alert and Log print the name of the AlertLogID. When PRINT_NAME_AND_PARENT, Alert and Log print the parent ID name, the ID separator, and then the ID name.

11.1.5 **NameSearchType**

```
type NameSearchType is (
    PRIVATE, NAME, NAME_AND_PARENT, NAME_AND_PARENT_ELSE_PRIVATE) ;
```

When NewID is called, first a search of existing IDs is done. If there is a match, that ID is returned, if there is not, a new one is created. Matching is determined by the ID, ParentID, and Search parameters. If the Search parameter is PRIVATE, no searching is done and a new ID is created. If the Search parameter is NAME, then any ID whose name matches is returned. If the Search parameter is NAME_AND_PARENT, then any ID whose name and ParentID match or NAME and stored Search parameter is NAME is returned.

12 **Basic Bin Description: GenBin, IllegalBin, and IgnoreBin**

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

12.1 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 ;
```

12.2 Creating Count Bins - GenBin

The following are three 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
AddBins(CovBin1, 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.

```
AddBins(CovBin2, 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.

```
AddBins(CovBin2,  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
AddBins(CovBin1, 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.

```
AddBins(CovBin3, GenBin(5) ) ;
```

```
AddBins(CovBin2, GenBin(5,5,1) ) ;  -- equivalent call
```

12.3 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
```

12.4 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) ;
```

12.5 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, 4 to 127, 128 to 252, 253 to 253, 254 to 254, and 255 to 255.

```
AddBins(CovBin1, GenBin(1, 3) & GenBin(4, 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.

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

13 Modeling Coverage

The coverage items are added to the coverage data structure using the procedures AddBins and AddCross.

13.1 Item (Point) Bins - AddBins

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 is defined for the following forms:

```
procedure AddBins (
  ID      : CoverageIDType ;
  Name    : String ;
  AtLeast : integer ;
  CovBin  : CovBinType
) ;
procedure AddBins (ID : CoverageIDType; Name : String;  CovBin : CovBinType) ;
procedure AddBins (ID : CoverageIDType; AtLeast : integer; CovBin : CovBinType ) ;
procedure AddBins (ID : CoverageIDType; CovBin : CovBinType  ) ;
```

The Name parameter associates a name with a coverage bin (or set of bins). The AtLeast parameter specifies a coverage goal for a coverage bin (or set of bins). In OSVVM, each coverage bin of a coverage model may have a different coverage goal.

13.2 Cross Coverage Bins - AddCross

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.

```
procedure AddCross(
  ID         : CoverageIDType ;
  Name       : string ;
  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
) ;
```

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

procedure AddCross(
  ID         : CoverageIDType ;
  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
) ;

procedure AddCross(
  ID         : CoverageIDType ;
  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 Name parameter associates a name with a coverage bin (or set of bins). The AtLeast parameter specifies a coverage goal for a coverage bin (or set of bins). In OSVVM, each coverage bin of a coverage model may have a different coverage goal.

13.3 Controlling Reporting for Illegal Bins - SetIllegalMode

By default, illegal bins both count and flag a severity error alert (see AlertLogPkg user guide). 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 alert to be printed when an item is added to an illegal bin.

```
type IllegalModeType is (ILLEGAL_ON, ILLEGAL_FAILURE, ILLEGAL_OFF) ;
procedure SetIllegalMode(ID : CoverageIDType; A : IllegalModeType) ;
```

Usage:

```
SetIllegalMode(CovBin4, ILLEGAL_OFF) ;  -- Illegal printing off
SetIllegalMode(CovBin4, ILLEGAL_ON) ;  --  Default:  Illegal printing on
```

13.4 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 (ID : CoverageIDType; NewNumBins : integer) ;
```

14 Accumulating Coverage

14.1 ICover

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(ID : CoverageIDType; CovPoint : integer_vector) ;
procedure ICover(ID : CoverageIDType; CovPoint : integer) ;
```

14.2 TCover

TCover is used to accumulate transition coverage. Transition coverage is accumulated over multiple samples of the coverage information. Transition coverage is modeled as cross coverage. Each time a value is received, it is accumulated internally. Once a minimum number of values are received, they are checked against the model.

```
procedure TCover (CoverID : CoverageIDType; A : integer) ;
```

14.3 Conversions to integer and 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.

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

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

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

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

```
ICover(CrossBin, to_integer_vector(CtrlReg_slv) ) ;  -- std_logic_vector
ICover(CrossBin, 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).

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

14.4 Conversions from integer and integer_vector

To support conversions from a point value back to a std_logic, Boolean, std_logic_vector, or Boolean_vector, the following conversions were added.

```
function to_boolean ( I : integer ) return boolean ;
function to_std_logic ( I : integer ) return std_logic ;
function to_boolean_vector ( IV : integer_vector ) return boolean_vector ;
function to_std_logic_vector ( IV : integer_vector ) return std_logic_vector ;
alias to_slv is to_std_logic_vector[integer_vector return std_logic_vector] ;
```

15 Basic Randomization

Randomization is handled by either GetRandPoint and GetRandBinVal. 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.

15.1 Randomly generating a value within a bin - GetRandPoint

GetRandPoint 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 GetRandPoint is shown below.

```
impure function GetRandPoint (ID : CoverageIDType) return integer ;
impure function GetRandPoint (ID : CoverageIDType) return integer_vector ;
```

Note that GetRandPoint is the new name for RandCovPoint and that RandCovPoint is deprecated.

15.2 Randomly selecting a coverage bin - GetRandBinVal

GetRandBinVal 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 GetRandBinVal (ID : CoverageIDType) return RangeArrayType ;
```

Note that GetRandBinVal is the new name for RandCovBinVal and that RandCovBinVal is deprecated.

15.3 Randomization, Illegal, and Ignore Bins

GetRandPoint and GetRandBinVal 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.

16 Coverage Model Statistics

Coverage model statistics collecting functions 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).

16.1 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(ID : CoverageIDType) return boolean ;
```

16.2 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 (ID : CoverageIDType) return boolean ;
```

16.3 Number of Items Randomized - GetItemCount

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

```
impure function GetItemCount (ID : CoverageIDType) return integer ;
```

16.4 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 (ID : CoverageIDType) return integer ;
```

16.5 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 (ID : CoverageIDType) return real ;
```

17 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).

Some of this is more complicated than desired since VHDL does not permit a file identifier to be passed to a method of a protected type.

17.1 Reports for a Single Coverage Model

17.1.1 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.

```
procedure WriteBin (ID : CoverageIDType) ;
. . .
WriteBin(CovBin) ;
```

17.1.2 Reporting Coverage Holes - WriteCovHoles

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

```
procedure WriteCovHoles (ID : CoverageIDType; LogLevel : LogType := ALWAYS ) ;
. . .
WriteCovHoles(CovBin1) ;
```

When the LogLevel parameter is specified (such as DEBUG) and that Level is not enabled within AlertLogPkg, then the WriteCovHoles will not print. The WriteCovHoles with a LogLevel parameter of "ALWAYS" always prints.

17.1.3 Specifying Files with FileOpenWriteBin

By default, WriteBin and WriteCovHoles print to the file specified by TranscriptPkg (either TranscriptFile, OUTPUT, or both if mirroring is enabled).

FileOpenWriteBin opens a file that is used by WriteBin and WriteCovHoles for all coverage models. If opened, this file will be used instead of the file specified by TranscriptPkg. The declaration of FileOpenWriteBin is shown below.

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

There is also a corresponding FileCloseWriteBin.

```
procedure FileCloseWriteBin ;
```

17.1.4 Specifying Files with WriteBin and WriteCovHoles

WriteBin and WriteCovHoles can also be called with a string based FileName parameter. If specified, this file will be used instead of the FileOpenWriteBin and TranscriptFile.

```
procedure WriteBin (
   ID       : CoverageIDType;
   FileName : string;
   OpenKind : File_Open_Kind := APPEND_MODE
) ;
procedure WriteCovHoles (
   ID       : CoverageIDType;
   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
WriteBin      (CovBin1, "Test1.txt",    WRITE_MODE);
WriteCovHoles (CovBin1, "Test1.txt",    APPEND_MODE);
```

17.1.5 Conditional Printing using LogLevels

If the LogLevel parameter is specified (such as DEBUG) and that Level is not enabled within AlertLogPkg, then the WriteBin or WriteCovHoles will not print. The WriteBin or WriteCovHoles without the LogLevel parameter always prints.

```
procedure WriteBin (
  ID       : CoverageIDType;
  LogLevel : LogType
) ;
procedure WriteBin (
  ID       : CoverageIDType;
  LogLevel : LogType;
```

```
    FileName : string;
    OpenKind : File_Open_Kind := APPEND_MODE
  ) ;
  procedure WriteCovHoles (
    ID       : CoverageIDType;
    LogLevel : LogType := ALWAYS
  ) ;
  procedure WriteCovHoles (
    ID       : CoverageIDType;
    LogLevel : LogType;
    FileName : string;
    OpenKind : File_Open_Kind := APPEND_MODE
  ) ;
```

17.1.6 Printing Message Headings - PrintToCovFIle

PrintToCovFile prints a string value to the same file used by WriteBin and WriteCovHoles. PrintToCovFile is intended for printing headings.

```
procedure PrintToCovFile(S : string) ;
```

17.1.7 Setting Defaults for WriteBin fields: SetReportOptions

WriteBin uses the following format when printing. This format is used to facilitate integration with requirements tracing tools.

```
{Prefix} [BinName] [PASSED|FAILED] [BinInfo] [Count]
```

The procedure SetReportOptions sets defaults for the WriteBin options for all coverage models. The interface for SetReportOptions is as follows.

```
procedure SetReportOptions (
      WritePassFail   : OsvvmOptionsType := COV_OPT_INIT_PARM_DETECT ;
      WriteBinInfo    : OsvvmOptionsType := COV_OPT_INIT_PARM_DETECT ;
      WriteCount      : OsvvmOptionsType := COV_OPT_INIT_PARM_DETECT ;
      WriteAnyIllegal : OsvvmOptionsType := COV_OPT_INIT_PARM_DETECT ;
      WritePrefix     : string := OSVVM_STRING_INIT_PARM_DETECT ; -- Deprecated
      PassName        : string := OSVVM_STRING_INIT_PARM_DETECT ; -- Deprecated
      FailName        : string := OSVVM_STRING_INIT_PARM_DETECT   -- Deprecated
    ) ;
```

With OSVVM release 2024.03, the default settings come from OsvvmSettingsPkg_default. If you want to change that default setting, copy OsvvmSettingsPkg_default.vhd to OsvvmSettingsPkg_local.vhd and make the changes you desire. This provides a simple way to make these values available to all test cases. For the deprecated parameters, WritePrefix, PassName, and FailName, usage of the constant is the only way to set these values now.

OsvvmOptionsType is defined in OsvvmGlobalPkg.vhd.

The default values for these parameters and the variable from OsvvmSettingsPkg that set them are shown in the table below.

Parameter	Constant	Default
WritePassFail	COVERAGE_WRITE_PASS_FAIL	FALSE
WriteBinInfo	COVERAGE_WRITE_BIN_INFO	TRUE
WriteCount	COVERAGE_WRITE_COUNT	TRUE
WriteAnyIllegal	COVERAGE_WRITE_ANY_ILLEGAL	FALSE
Deprecated WritePrefix	COVERAGE_PRINT_PREFIX	"%% "
Deprecated PassName	COVERAGE_PASS_NAME	"PASSED"
Deprecated FailName	COVERAGE_FAIL_NAME	"FAILED"

When all parameters are TRUE, the report will print with the following format.

```
%% State0  PASSED Bin:(0)   Count = 1  AtLeast = 1
%% State1  PASSED Bin:(1)   Count = 1  AtLeast = 1
%% State2  FAILED Bin:(2)   Count = 0  AtLeast = 1
%% State3  FAILED Bin:(3)   Count = 0  AtLeast = 1
```

The "%% " is the value of WritePrefix. The "State0", "State1", … are the names of the bins and if present always print. Next is the PassFail message. It will print "PASSED" if the count is greater than or equal to the goal (AtLeast value), otherwise, it prints "FAILED". The PassFail message is enabled using the WritePassFail field. The value printed when it passes or fails is controlled by the PassName and FailName fields. Printing of bin information may be redundant when a bin is named. This information can be disabled using the WriteBinInfo field. Printing of the Count (current coverage) and AtLeast (coverage goal) can be disabled with the WriteCount field. Nominally illegal bins are only printed when the have failed (something landed in that bin). The parameter, WriteAnyIllegal, can be used to enable printing of all illegal bins (including the ones with no values and hence pass).

17.2 **Reports for All Coverage Models**

The coverage models are implemented in a singleton data structure inside of CoveragePkg. This data structure allows us to iterate across all of the coverage models to gather information.

Generally only SetCovWeight is called directly in your testbenches. The others are called as a result of calling EndOfTestReports (see AlertLogPkg_User_Guide.pdf). EndOfTestReports generates YAML information for test reports, alert reports and coverage reports (by calling WriteCovYaml).

17.2.1 **WriteCovYaml: Create YAML coverage models**

WriteCovYaml creates a YAML representation of the internal singleton data structure that contains all of the coverage models. It also stores the settings applied to a coverage model.

```
procedure WriteCovYaml (
  FileName : string := "";
  OpenKind : File_Open_Kind := WRITE_MODE
) ;
```

Instead of calling WriteCovYaml directly, it is recommended to call EndOfTestReports at the end of a simulation (see AlertLogPkg_User_Guide). In addition, when a test is run using the OSVVM script simulate, the YAML file will automatically be converted to HTML (by script Cov2Html). Details of the OSVVM reports are shown in the "OSVVM Test Writer's User Guide" and the "Script User Guide" in the Documentation repository.

17.2.2 **ReadCovYaml: Read YAML coverage models**

ReadCovYaml reads in the YAML representation of the coverage models written out by WriteCovYaml.

```
procedure ReadCovYaml (
  FileName : string := "";
  Merge    : boolean := FALSE
) ;
```

If the Merge flag is TRUE, the model information is merged with an existing coverage model of the same name.

17.2.3 **GotCoverage: Are there any coverage models in CoveragePkg?**

The function GotCoverage returns TRUE when the singleton data structure has at least one coverage model.

```
impure function GotCoverage return boolean ;
```

17.2.4 **GetCov: Calculate coverage across all coverage models**

GetCov calculates the coverage across all coverage models. To do this, it sums up all the coverage counts and goals for a coverage model. It then multiplies these values by the CovWeight. The coverage is then calculated as the sum of the weighted counts divided by the sum of the weighted goals.

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

17.2.5 AffirmIfCovered and AlertIfNotCovered

AffirmIfCovered produces an ERROR alert if all coverage models are not covered (GetCov /= 100.0) and a PASSED log if all coverage models are covered. AlertIfNotCovered produces an ERROR alert if all coverage models are not covered (GetCov /= 100.0)

```
procedure AffirmIfCovered ;
procedure AlertIfNotCovered (Level : AlertType := ERROR) ;
```

17.2.6 GetTotalCovCount

Return the sum of the coverage counts for a coverage model. If the coverage count for a bin is larger than the coverage goal, it uses the value of the coverage goal instead.

```
impure function GetTotalCovCount(ID : CoverageIDType; PercentCov : real )
    return integer ;
impure function GetTotalCovCount(ID : CoverageIDType) return integer ;
```

17.2.7 GetTotalCovGoal

Return the sum of the coverage goals for a coverage model.

```
impure function GetTotalCovGoal (ID : CoverageIDType; PercentCov : real )
    return integer ;
impure function GetTotalCovGoal (ID : CoverageIDType) return integer ;
```

17.2.8 SetCovWeight / GetCovWeight: Weight of a Model

SetCovWeight sets the CovWeight for a coverage model. GetCovWeight returns the CovWeight for a coverage model. CovWeight is used in calculating the total coverage across all coverage models. See GetCov.

```
procedure       SetCovWeight       (ID : CoverageIDType; Weight : integer) ;
impure function GetCovWeight       (ID : CoverageIDType) return integer ;
```

By default CovWeight is 1. Set CovWeight to 0 to ignore a coverage model when calculating the coverage across the coverage models in the CoveragePkg singleton data structure.

17.3 Setting Bin Names

Each bin can be named. The bin name is specified as the first parameter to AddBins and AddCross. This means to use names, one must specify each bin individually (bin by bin). The intent behind bin names it to correlate a requirement with a bin name and furthermore associate this with a pass or fail indication of the requirement.

```
procedure AddBins (
  ID      : CoverageIDType ;
  Name    : String ;
  AtLeast : integer ;
  CovBin  : CovBinType
) ;
```

```
procedure AddBins (ID : CoverageIDType; Name : String;  CovBin : CovBinType) ;

procedure AddCross(
  ID         : CoverageIDType ;
  Name       : string ;
  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
) ;

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

17.4 SetItemBinNames: Setting Item Bin Names

For a cross coverage model, the names for each item bin can be set by calling SetItemBinNames.

```
procedure SetItemBinNames (
    ID         : CoverageIDType ;
    Name1      : String ;
             Name2,  Name3,  Name4,  Name5,
    Name6,  Name7,  Name8,  Name9,  Name10,
    Name11, Name12, Name13, Name14, Name15,
    Name16, Name17, Name18, Name19, Name20 : string := ""
) ;
```

18 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.

18.1 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 (
  ID      : CoverageIDType ;
  Name    : String ;
  AtLeast : integer ;
  CovBin  : CovBinType
) ;
procedure AddBins (ID : CoverageIDType; AtLeast : integer; CovBin : CovBinType ) ;

procedure AddCross(
  ID         : CoverageIDType ;
  Name       : string ;
  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
) ;

procedure AddCross(
  ID         : CoverageIDType ;
  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.

18.2 Selecting Randomization Weights - SetWeightMode

By default, a coverage goal is used as the randomization weight.

Selection of the randomization weight is done using SetWeightMode. Currently the only supported (meaning not deprecated modes) are AT_LEAST and REMAIN. These are as specified in the following table. Also see SetCovTarget as it can scale the AtLeast value.

Mode	Weight
AT_LEAST	AtLeast
REMAIN	AtLeast - Count *

Mode	Weight
* Note AtLeast is adjusted if the coverage target /= 100 %	

The interface for procedure SetWeightMode is shown below.

```
type WeightModeType  is (AT_LEAST, REMAIN, <remaining are deprecated>) ;
procedure SetWeightMode (ID : CoverageIDType; WeightMode : WeightModeType) ;
```

Note that any undocumented features are deprecated and likely to be removed.

19 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).

19.1 Setting a Coverage Target - SetCovTarget

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

```
procedure SetCovTarget (ID : CoverageIDType; 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 subprograms that do not have a PercentCov parameter use the CovTarget value: GetRandPoint, GetRandBinVal, IsCovered, and WriteCovHoles.

19.2 Overriding the Global Coverage Target - PercentCov

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

```
impure function GetRandPoint (ID : CoverageIDType; PercentCov : real )
  return integer ;
impure function GetRandPoint (ID : CoverageIDType; PercentCov : real )
  return integer_vector ;
impure function GetRandBinVal (ID : CoverageIDType; PercentCov : real )
  return RangeArrayType ;
impure function IsCovered(ID : CoverageIDType; PercentCov : real) return boolean;
```

```
procedure WriteCovHoles (ID : CoverageIDType; PercentCov : real) ;
procedure WriteCovHoles (ID : CoverageIDType;
                         LogLevel : LogType; PercentCov : real) ;
procedure WriteCovHoles (ID : CoverageIDType;
  FileName : string; PercentCov : real; OpenKind : File_Open_Kind := APPEND_MODE);
procedure WriteCovHoles (ID : CoverageIDType; LogLevel : LogType;
  FileName : string; PercentCov : real; OpenKind : File_Open_Kind := APPEND_MODE);

impure function GetTotalCovGoal (ID : CoverageIDType; PercentCov : real)
  return integer ;
impure function GetCov (ID : CoverageIDType; PercentCov : real ) return real;
impure function CountCovHoles (ID : CoverageIDType; PercentCov : real)
  return integer ;
```

20 Randomization Thresholds - SetThresholding and SetCovThreshold

Ordinarily randomization (using GetRandPoint or GetRandBinVal) 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 (ID : CoverageIDType; A : boolean := TRUE ) ;
procedure SetCovThreshold (ID : CoverageIDType; Percent : real)  ;
```

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

21 Handling Overlapping Bins

21.1 LastIndex - Count bins overlapping with other counts

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

21.2 Bin Merging

21.2.1 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 (ID : CoverageIDType; 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.

21.2.2 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.

21.3 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) ;
procedure SetCountMode (ID : CoverageIDType; A : CountModeType) ;

SetCountMode(CovBin4, COUNT_ALL) ;    -- Count all matching bins
SetCountMode(CovBin4, 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.

22 Working with Randomization Seeds

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.

22.1 NewID Sets the Randomization Seed

NewID sets the randomization seed using the string value specified in the call.

```
Cov1 <= NewID("Cov1") ;
```

If a verification component is used more than once, you may want to use the instance name as the ID name. The example assumes the entity name is VC1.

```
Cov1 <= NewID(VC1'instance_name) ;
```

22.2 InitSeed Sets the Randomization Seed

InitSeed hashes its string or integer parameter and sets the randomization seed. Note that NewID already called InitSeed with its string parameter, so most of the time, calling InitSeed separately should be unnecessary.

```
procedure InitSeed (ID : CoverageIDType; S : string;
    UseNewSeedMethods : boolean := TRUE) ;
procedure InitSeed (ID : CoverageIDType; I : integer;
    UseNewSeedMethods : boolean := TRUE ) ;
```

If are trying to reproduce an older test that used the previous version of CoveragePkg, set UseNewSeedMethods to FALSE, otherwise, leave it as TRUE.

```
InitSeed(Cov1, VC1'instance_name) ;
```

22.3 Saving and Restoring the Randomization Seed

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 (ID : CoverageIDType; RandomSeedIn : RandomSeedType ) ;
impure function GetSeed (ID : CoverageIDType) return RandomSeedType ;
```

23 Using an AlertLogID

Alerts signaled by CoveragePkg use an internal AlertLogID. The AlertLogID is set using the name in the call to NewID.

23.1 GetAlertLogID

GetAlertLogID returns the current AlertLogID used to report alerts.

```
impure function GetAlertLogID (ID : CoverageIDType) return AlertLogIDType ;
```

24 Interacting with the Coverage Data Structure

GetRandPoint is the primary way of getting the next stimulus point from the coverage model. However, from time to time we need to use other methods – such as incrementing or minimum. This section provides non-random ways of getting index, bin, or point values.

In general access starts by having bin index. From a bin index, a BinVal can be looked up in the data structure. From a BinVal, a point can be randomly selected within the BinVal.

24.1 Getting Bin Index Values

A bin index is a reference to a location in the coverage data structure. These are numbered from 1 to NumBins. Internal to the data structure, the last index used for stimulus generation is stored in LastStimGenIndex, and the last index used for either stimulus generation or for coverage recording (ICover) is stored in LastIndex.

```
impure function GetNumBins (ID : CoverageIDType) return integer ;
impure function GetRandIndex (ID : CoverageIDType; CovTargetPercent : real)
    return integer ;
impure function GetRandIndex (ID : CoverageIDType) return integer ;
impure function GetLastIndex (ID : CoverageIDType) return integer ;
impure function GetIncIndex (ID : CoverageIDType) return integer ;
impure function GetMinIndex (ID : CoverageIDType) return integer ;
impure function GetMaxIndex (ID : CoverageIDType) return integer ;
impure function GetNextIndex (ID : CoverageIDType; Mode : NextPointModeType)
    return integer ;
impure function GetNextIndex (ID : CoverageIDType) return integer ;
```

GetNumBins returns the number of bins in the coverage model.

GetLastIndex returns the index that was last used for either used for stimulus generation (in the case of GetRandIndex, GetIncIndex, GetMinIndex, GetMaxIndex, or GetNextIndex) or coverage collection (in the case of ICover).

GetRandIndex randomly selects the next index in the fashion described for GetRandPoint / GetRandBinVal in previous sections.

GetIncIndex gets the bin immediately after the last bin generated using an incrementing pattern (LastStimGenIndex + 1).

GetMinIndex (GetMaxIndex) returns the index of the first bin with the minimum (maximum) percent coverage of a bin.

GetNextBinVal selects the bin using the internal setting of the mode. The mode can be RANDOM, INCREMENT, or MIN (for GetRandBinVal, GetIncBinVal, or GetMinBinVal). The mode can be set with SetNextPointMode. Alternately a mode value can be specified on the call to GetNextBinVal.

```
procedure SetNextPointMode (A : NextPointModeType) ;
```

24.2 **Getting Coverage Point Values**

These functions access a point within a coverage bin. For cross coverage these return an integer_vector value. For item coverage, these return either an integer or integer_vector value (depending on the target of the assignment).

```
impure function GetRandPoint (ID : CoverageIDType) return integer ;
impure function GetRandPoint (ID : CoverageIDType) return integer_vector ;
impure function GetRandPoint (ID : CoverageIDType; PercentCov : real )
  return integer ;
impure function GetRandPoint (ID : CoverageIDType; PercentCov : real )
  return integer_vector ;
impure function GetPoint(ID : CoverageIDType; BinIndex : integer) return integer;
impure function GetPoint(ID : CoverageIDType; BinIndex : integer)
  return integer_vector ;
impure function GetIncPoint (ID : CoverageIDType) return integer ;
impure function GetIncPoint (ID : CoverageIDType) return integer_vector ;
impure function GetMinPoint (ID : CoverageIDType) return integer ;
impure function GetMinPoint (ID : CoverageIDType) return integer_vector ;
impure function GetMaxPoint (ID : CoverageIDType) return integer ;
impure function GetMaxPoint (ID : CoverageIDType) return integer_vector ;
impure function GetNextPoint (ID : CoverageIDType; Mode : NextPointModeType)
return integer ;
impure function GetNextPoint (ID : CoverageIDType; Mode : NextPointModeType)
return integer_vector ;
impure function GetNextPoint (ID : CoverageIDType) return integer ;
impure function GetNextPoint (ID : CoverageIDType) return integer_vector ;
```

GetRandPoint randomly selects a value in the bin value determined by GenRandIndex.

GetPoint returns a point within the bin indexed by the specified BinIndex.

GetIncPoint randomly selects a value in the bin value determined by GenIncIndex.

GetMinPoint randomly selects a value in the bin value determined by GenMinIndex.

GetMaxPoint randomly selects a value in the bin value determined by GetMaxIndex.

GetNextPoint randomly selects a value in the bin value determined by GetNextIndex.

24.3 **Getting Coverage Bin Values**

The following that return coverage bin value using type RangeArrayType.

```
impure function GetRandBinVal (ID : CoverageIDType) return RangeArrayType ;
impure function GetRandBinVal (ID : CoverageIDType; PercentCov : real )
  return RangeArrayType ;

impure function GetBinVal (ID : CoverageIDType; BinIndex : integer ) return
RangeArrayType ;


impure function GetLastBinVal (ID : CoverageIDType) return RangeArrayType ;
impure function GetIncBinVal (ID : CoverageIDType) return RangeArrayType ;
impure function GetMinBinVal (ID : CoverageIDType) return RangeArrayType ;
impure function GetMaxBinVal (ID : CoverageIDType) return RangeArrayType ;
impure function GetNextBinVal (ID : CoverageIDType; Mode : NextPointModeType)
return RangeArrayType ;
impure function GetNextBinVal (ID : CoverageIDType) return RangeArrayType ;
impure function GetHoleBinVal (ID : CoverageIDType; ReqHoleNum : integer ;
PercentCov : real  ) return RangeArrayType ;
impure function GetHoleBinVal (ID : CoverageIDType; PercentCov : real  ) return
RangeArrayType ;
impure function GetHoleBinVal (ID : CoverageIDType; ReqHoleNum : integer := 1 )
return RangeArrayType ;
```

The function GetBinVal returns the bin value indexed by the specified BinIndex.

GetRandBinVal returns the bin value determined by GenRandIndex.

GetLastBinVal returns the bin value determined by GetLastIndex.

GetIncBinVal returns the bin value determined by GenIncIndex.

GetMinBinVal returns the bin value determined by GenMinIndex.

GetMaxBinVal returns the bin value determined by GetMaxIndex.

GetNextBinVal returns the bin value determined by GetMaxIndex.

24.4 **Basic Bin Information**

```
impure function GetMinCov (ID : CoverageIDType) return real ;
```

```
impure function GetMinCount (ID : CoverageIDType) return integer ;
impure function GetMaxCov (ID : CoverageIDType) return real ;
impure function GetMaxCount (ID : CoverageIDType) return integer ;
```

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.

24.5 Getting the ERROR Count

GetErrorCount returns the coverage for bins that were marked as ILLEGAL bins. If all went well, this should be zero. The error count is recorded in the AlertLogPkg data structure.

```
impure function GetErrorCount (ID : CoverageIDType) return integer ;
```

This information can also be acquired by doing the following:

```
ErrCnt := GetAlertCount( GetAlertLogID(Cov1)) ;
```

24.6 Getting Coverage Bin Name

The function GetBinName returns the name of a bin.

```
impure function GetBinName (
    ID          : CoverageIDType;
    BinIndex    : integer;
    DefaultName : string := ""
) return string ;
```

24.7 Getting Coverage Holes

The following functions return information about coverage holes.

```
impure function CountCovHoles (ID : CoverageIDType) return integer ;
impure function CountCovHoles (ID : CoverageIDType; PercentCov : real)
    return integer ;

impure function GetHoleBinVal (
  ID         : CoverageIDType;
  ReqHoleNum : integer ;
  PercentCov : real
) return RangeArrayType ;
impure function GetHoleBinVal (
  ID         : CoverageIDType;
  PercentCov : real
) return RangeArrayType ;
impure function GetHoleBinVal (
  ID         : CoverageIDType;
  ReqHoleNum : integer := 1
) 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   := CountCovHoles(CovBin1, 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 := GetHoleBinVal(CovBin1,     5,          100.0  ) ;
```

25 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 subprogram 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 ReadCovDb (
  ID       : CoverageIDType;
  FileName : string;
  Merge    : boolean := FALSE
) ;
procedure WriteCovDb (
  ID : CoverageIDType;
  FileName : string;
  OpenKind : File_Open_Kind := WRITE_MODE
) ;
```

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

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

26 Bin Clearing and Deconstruction

The procedure ClearCov 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.

```
ClearCov(CovBin1) ;  -- set all counts to 0
```

27 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.

27.1 Item (Point) Bin Constants - CovBinType

In previous examples, we added bins to a cross coverage model using a call to AddCross.

```
AddCross(ACov, 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.

```
AddCross(ACov, 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) ;
```

27.2 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 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);
```

Use one of the versions of AddCross shown below add the constant to the coverage data structure.

```
procedure AddCross(ID : CoverageIDType; CovBin : CovMatrix2Type; Name : String := "");
```

```
procedure AddCross(ID : CoverageIDType; CovBin : CovMatrix3Type ; Name : String := "");
procedure AddCross(ID : CoverageIDType; CovBin : CovMatrix4Type ; Name : String := "");
procedure AddCross(ID : CoverageIDType; CovBin : CovMatrix5Type ; Name : String := "");
procedure AddCross(ID : CoverageIDType; CovBin : CovMatrix6Type ; Name : String := "");
procedure AddCross(ID : CoverageIDType; CovBin : CovMatrix7Type ; Name : String := "");
procedure AddCross(ID : CoverageIDType; CovBin : CovMatrix8Type ; Name : String := "");
procedure AddCross(ID : CoverageIDType; CovBin : CovMatrix9Type ; Name : String := "");
```

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

```
AddCross (ACov, 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 AtLeast. 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
  signal ACov : CoverageIDType ;                -- 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
    ACov <= NewID("ACov") ;
    wait for 0 ns ; -- Allow ACov to update
    -- Capture coverage model
    AddCross (ACov, ALU_BIN_CONST ) ;

    loop
      -- Randomize register addresses -- see RandomPkg documentation
      (RegIn1, RegIn2) := GetRandPoint(ACov) ;

      DoAluOp(TRec, RegIn1, RegIn2) ;             -- Do a transaction
      ICover(ACov, (RegIn1, RegIn2) ) ;           -- Accumulate
      exit when IsCovered(ACov) ;                 -- done
    end loop ;

    WriteBin(ACov) ;                               -- Report
    ReportAlerts ;
  end process ;
```

28 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.

29 Compiling CoveragePkg

See Script_user_guide.pdf for the current compilation directions.

30 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.

31 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.

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

32 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 Apache open source license. It is free (both to download and use - there are no license fees). You can download it from https://www.GitHub.com/OSVVM/OSVVM. 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 OSVVM_release_notes.pdf.

33 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus 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.

34 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

35 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.