OPEN SOURCE VHDL VERIFICATION METHODOLOGY
User's Guide

Rev. 1.2

2012-01-05

Abstract

This document describes OPEN SOURCE VHDL VERIFICATION METHODOLOGY (OS-VVM) that combines two VHDL packages: RANDOMPKG and COVERAGEPKG.

OS-VMM allows flexible, user friendly generation of random numbers and implementation of functional coverage in VHDL.

The key feature of this methodology is the ability to control random stimulus generation based on the current coverage results.

Table of Contents

Introduction

Modern digital designs, especially those describing complete systems, frequently require the use of random stimulus generation and functional coverage in their verification.

Random stimulus is needed to model behavior of the environment in which implemented design is supposed to work or (together with functional coverage) to provide fast alternative to directed tests. Programming languages usually provide basic features supporting random number generation and users have to do some serious coding to get the stream of random numbers they really need.

Many verification procedures created for large designs produce large amount of output data that is hard to analyze, no matter if directed tests or constrained random tests were implemented. The concept of Functional Coverage addresses this issue by enabling description and verification of coverage goals based on the values of critical variables collected during simulation. Some languages offer built-in facilities supporting functional coverage, others (like VHDL) require manual coding to achieve the same results.

Open Source VHDL Verification Methodology provides two packages created by Jim Lewis, Director of VHDL Training at SynthWorks: **RANDOMPKG** and **COVERAGEPKG**. The packages can support stand-alone randomization and functional coverage, but the strongest feature of OS-VMM is the ability to control random stimulus generation based on the current status of the functional coverage.

Random Number Generation Background

Excellent ***Random Number Generator*** (RNG), also known as ***True Random Number Generator*** (TRNG) is the key source of data in numerous applications. The TRNG generating numbers from a given value range should meet two requirements:

- **No bias**: all numbers in the range should appear with equal probability.
- **No period**: there should be no repeating patterns in the generated stream of numbers.

Pseudo-Random Number Generator (PNRG) keeps the ***"no bias"*** requirement, but reduces the second requirement to ***"very long period"***, i.e. it generates very long, repeating sequence of equally distributed random numbers.

One very popular class of PRNG is called ***Linear Congruential Generator*** (LCG) and creates next random number by applying simple linear formula to the previously generated number. It means that initial number called ***seed*** must be specified to start the entire sequence of random numbers.

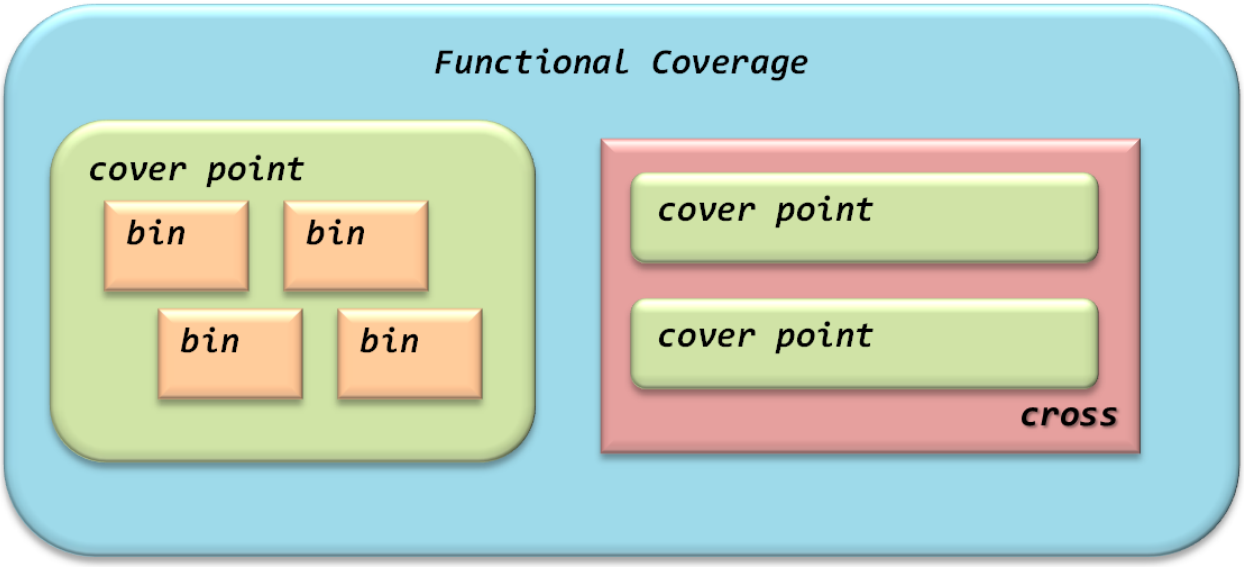
VHDL is equipped with LCG-based **UNIFORM** function available in the **MATH_REAL** package. Since the function combines two LCGs, it requires two positive integer seeds and it generates pseudo-random real numbers normalized to (0.0, 1.0) interval.

LGCs are considered sufficient for generation of simulation stimulus, but should not be used for cryptography and advanced statistical applications. The **RANDOMPKG** package uses **MATH_REAL.UNIFORM** by default, but can be modified to incorporate user-provided, better quality (but slower) algorithms, such as *Mersenne Twister*.

Functional Coverage Background

Functional Coverage is a kind of metric showing how much of the design specification was exercised. The quality of coverage results depends heavily on the test plan, i.e. 100% functional coverage means 100% of features that were selected for testing. Closely related code coverage is usually a tool function

and can be fully automated; Functional Coverage requires careful preparation of the test and analysis of the results, so it cannot be fully automated.



Functional coverage data is collected by sampling values during simulation.

- ***Cover points*** are one-dimensional expressions (sometimes just variables) sampled during coverage data collection.
- ***Bins*** are the ranges of values within cover point for which coverage data is accumulated, i.e. detection of any value from the range increments coverage count for the bin.
- ***Crosses*** are multi-dimensional expressions that can be treated as Cartesian product of cover points or just collections of multi-dimensional bins. They collect cross-coverage data in the shape of pairs, triples, etc. of values sampled from multiple variables.

As an example, let's consider VHDL testbench collecting 8-bit data from 8 × 8 matrix of sensors. If both sensor data and sensor indices are generated randomly, we can create:

- cover point for sensor data with 16 bins of equal size (0..15, 16..31, 32..47, etc.),
- cross for index data with 64 one-value bins ((0,0), (0,1),…,(1,0), (1,1), etc.)

We can specify minimal coverage count required for each bin. Global coverage goal will be achieved if all bins reached required minimal count.

Protected Types

All packages and other pieces of code presented in this document rely on ***protected types*** – VHDL concept similar to ***classes*** known from other programming languages.

Protected types facilitate ***encapsulation*** in VHDL code:

- combination of pieces of data (***properties***) and operations that can be performed on them (***methods***) into one object,
- hiding of implementation-specific data from the end-user.

Users that never worked with protected types can easily imagine them as special version of record types that allow procedures and functions in addition to regular data fields.

Here's sample declaration and body of a protected type:

```
type FlagsPT is protected              -- declaration
    procedure set;                          -- procedure method
    procedure reset; ;                      -- procedure method
    impure function is_set return BOOLEAN; ; -- function method
end protected FlagsPT;

type FlagsPT is protected body         -- body
    variable flag : BOOLEAN := False;     -- private property
    procedure set is
    begin
        flag := True;
    end;
    procedure reset is
    begin
        flag := False;
    end;
    impure function is_set return BOOLEAN is
    begin
        return flag;
    end;
end protected body FlagsPT;
```

The public part of protected type is described in **protected..end protected** section (declaration); only items visible here are directly accessible to variables of the protected type.

The protected type body (**protected body..end protected body** section) is the implementation of the protected type. It contains full descriptions of all methods mentioned in the declaration and private data fields and subprograms accessible only via methods.

Protected types can be used with variables only: either local to processes or subprograms or shared variables declared in the architectures. Public items from the type declaration can be accessed using **variable_name.method_name** notation.

```
test : process
        variable myflag : FlagsPT;
    begin
        report "!!! Setting 'myflag'!!!";
        myflag.set;
        if myflag.is_set then
            report "'myflag' is set.";
        end if;
        report "!!! Resetting 'myflag'!!!";
        myflag.reset;
        if not myflag.is_set then
            report "'myflag' is set now.";
        end if;
        wait;
    end process;
```

Please note that the end user of the **FlagsPT** type is shielded from the internal representation of the flag value. We can change it from Boolean to Integer or Bit (with appropriate modifications of methods) and the process using variable **myflag** will still look and work the same.

Protected types in OS-VVM shield end user from quite arcane data structures and subprograms supporting randomization and coverage.

Preparing OS-VVM library

RANDOMPKG Package Files

The **RANDOMPKG** package consists of three VHDL sources:

- **SORTLISTPKG_INT.VHD**
- **RANDOMBASEPKG.VHD**
- **RANDOMPKG.VHD**

The first two files contain auxiliary declarations for the main package described in the third file. In normal circumstances only the contents of **RANDOMPKG.VHD** should be referenced in end-user designs.

COVERAGEPKG Package Files

The **COVERAGEPKG** package is described in one **COVERAGEPKG.VHD** VHDL source.

Since the package supports intelligent randomization of stimulus, it must have the access to the packages **RandomPkg** and **RandomBasePkg** mentioned in the previous section. Typically both random packages and the coverage package are compiled to the same library.

Compiling Packages

Sources of both packages should be compiled in the order specified above, using options enabling support of VHDL 2008 standard (if available). The minimal language version required for successful compilation is VHDL 2002.

It should be always possible to compile sources to the same library as the files of the design that use OS-VVM packages. In this case, design units can reference those packages like this:

```
use work.RandomPkg.all;
use work.CoveragePkg.all;
```

If the package is to be used frequently in multiple different designs, users should consider compiling package files to a separate binary library OSVVM that can be referenced without compilation. In Aldec simulators the following commands can be used in the console:

```
alib -global OSVVM $aldec/vlib/OSVVM/OSVVM.lib
set worklib OSVVM
acom -2008 SortListPkg_int.vhd RandomBasePkg.vhd RandomPkg.vhd
acom -2008 CoveragePkg.vhd
```

The first command creates new, global library **OSVVM** in standard tool folder. The second command makes that library a working library for the next command. The third and fourth commands compile sources, assuming that they are located in the current directory.

Check the tool documentation if you are using different vendor simulator.

Using RANDOMPKG Package

The package allows generation of random numbers of **real**, **integer**, **standard_logic_vector**, **unsigned** and **signed** type. Distribution and range of generated values can be specified, together with exclusions (unwanted values). It is also possible to specify series of values from which one should be selected randomly; each value in the series can have unique weight that decides if it is selected more or less frequently than the others.

To ensure uniqueness of random number sequences and to allow repeatable simulation results the package provides subprograms that control RNG seed.

Special Data Types

To handle 'series of numbers' parameters, the package defines two types:

- **DistRecType** – a record with integer fields **Value** and **Weight**;
- **DistType** – an unconstrained array **DistRecType** elements.

To allow specification, storage and retrieval of random distribution type used by RNG functions, special enumeration type **RandomDistType** is declared:

```
type RandomDistType is (NONE, UNIFORM, FAVOR_SMALL, FAVOR_BIG, NORMAL, POISSON);
```

For practical applications NONE value has the same effect as UNIFORM, but allows checking if current distribution has default value (NONE) or was changed by the code.

While UNIFORM, NORMAL and POISSON distributions behave as described in the literature, FAVOR_BIG and FAVOR_SMALL use square root function internally to get expected non-uniform results.

To specify not only distribution type, but also its numerical parameters, the following **RandomParmType** record type is declared:

```
type RandomParmType is record
  Distribution : RandomDistType ;
  Mean         : Real ;
  StdDeviation : Real ;
end record ;
```

After all auxiliary declarations, the package declares protected type **RandomPType** that handles RNG operations via its methods described in the following sections.

Please note that to avoid decreasing quality of generated random numbers users should declare variables of **RandomPType** type as local with the narrowest possible scope. Such variables should be used to generate stream of values for one specific application/design object and should not be shared.

Handling Seeds

The package automates handling of seeds required by RNG. To enable flexible initialization of the generators, dedicated **InitSeed** procedure method is provided in three versions: with S (**string**), I (**integer**) and IV (**integer_vector**) arguments. For some forms of constant values of seed the method call can be ambiguous, so it is recommended to qualify those values or specify both formal parameter name and its value. In case of multiple random variables (objects) used in the same design unit, it is recommended to use attribute specific to variable location for seeding RNG:

```
RNGvar.InitSeed(RNGvar'instance_name);
```

To store or transfer seed values between random variables, function method **GetSeed** and procedure method **SetSeed** were created; both are using **RandomSeedType** type to represent seed value.

```
RNGvarA.SetSeed(RNGvarB.GetSeed);
```

For users accustomed to SystemVerilog, **GetSeed** and **SetSeed** methods functionality is also available in equivalent **SeedRandom** method that can be called as bot function and procedure.

Distribution-Specific Methods

If the test code requires clear indication of the distribution of values generated by given method call, there are several dedicated methods that can be used.

If the required distribution is **Uniform**, **Favor_small** or **Favor_big**, methods with the distribution name can be used. All three are accept **Min** and **Max** arguments (**integer** or **real**) to specify desired value range and return random number of the argument type. There is also version of each function method that accepts third argument **Exclude** of **integer_vector** type to allow exclusion of some numbers from the specified range.

```
A := RandVarA.Uniform(1,6);               -- no exclusions
B := RandVarB.Favor_big(0,255,(44, 77)); -- two exclusions
```

The function method called **Normal** returns random value of **real** or **integer** type with normal distribution. The **real** type version requires **Mean** and **StdDeviation** arguments to specify distribution parameters and can also accept **Min** and **Max** arguments to specify value range. The **integer** type version requires **Mean**, **StdDeviation**, **Min** and **Max** arguments and can accept optional **Exclude** argument of **integer_vector** type to exclude some values from the generated range.

```
X := RandVarX.Normal(128.0, 16.0, 0.0, 256.0);
```

The method called **Poisson** returns random value of **real** or **integer** type with Poisson distribution. The **real** type version requires **Mean** distribution parameter and can also accept **Min** and **Max** arguments to specify value range. The **integer** type version requires **Mean**, **Min** and **Max** arguments and can accept optional **Exclude** argument of **integer_vector** type to exclude some values from the generated range.

```
Y := RandVarY.Poisson(16.0, 0, 63);
```

Please note that for **Normal** and **Poisson**, distribution parameters are always **real**, while **Min** and **Max** match the returned value type.

Universal Methods

The package is also equipped with a group of methods generating integral values of

- **integer**,
- **std_logic_vector**,
- **unsigned**,
- **signed**

data types.

Methods with names starting with **Rand-** accept value range or set of values restrictions of generated numbers.

Methods starting with **Dist-** accept series of values and weight or just weights. If just the vector containing **N** weights is specifies, it is assumed that they apply to the generated values from the **0..N-1** range. Numbers with greater weights are generated with higher probability than numbers with smaller weights.

While the type of generated value is suggested by remainder of function name (**RandInt**, **RandSlv**, **DistUnsigned**, **DistSigned**, etc.), the distribution used during generation is stored in the internal status variable and can be changed during the run of the test.

Dedicated procedure method **SetRandomParm** allows dynamic change of the distribution used by universal functions. Its argument is either solitary record of **RandomParmType** or a group of three values: enumeration value of **RandomDistType**, **Mean** and **Deviation** of **real** type.

Matching **GetRandomParm** function method returns current distribution as **RandomParmType** or **RandomDistType** value.

```
RandVar.SetRandomParm(NORMAL, 10.0, 2.0);
Dist := RandVar.GetRandomParm;
```

Arguments of universal function methods allow specification of the range (or set) of values that can be generated, exclusions and size of the generated vector value.

First arguments of all universal functions are obligatory and describe basic range or set of values that should be generated.

Universal Function Name	Basic Range or Set Arguments
Rand*	Range: Min, Max **integer** values
Rand*	Set of values: **integer_vector**
Dist*	N weights for integers from 0..N-1 range: **integer_vector**
DistVal*	Pairs of value and weight: **DistType**

Optional **integer_vector** argument that follows obligatory arguments specifies exclusions from the value range or set.

For functions returning vector value (***Slv**, ***Unsigned**, ***Signed**), the size of the vector must be specified as the last argument.

```
DataInt := RV.RandInt(0, 7);     -- Generate integer from the range 0 to 7.
DataInt := RV.RandInt(1, 13, (3, 7, 11)); -- range 1 to 13, exclude 3, 7, 11.
DataSlv := RV.RandSlv(0, 9, 5); -- std_logic_vector, range 0 to 9, 5-bit long.
DataInt := RV.RandInt( (-5, -1, 3, 7, 11), (-1, 7) ); -- set of 5 values,
                                -- two values excluded.
DataInt := RV.DistInt( (7, 2, 1) ); -- numbers 0, 1, 2 with weights 7, 2, 1.
DataInt := RV.DistValInt( ((1, 7), (3, 2), (5, 1)), (1=>3) ); -- set of three
                                -- values with weights, one value excluded.
```

Remaining universal methods can be treated as convenience functions, since they do not provide additional functionality, just different selection of arguments:

```
impure function RandReal(Min, Max: Real) return real ;
impure function RandReal return real ;              -- 0.0 to 1.0
impure function RandReal(Max: Real) return real ;   -- 0.0 to Max
impure function RandInt (Max : integer) return integer ;
impure function RandSlv (Size : natural) return std_logic_vector ;
impure function RandSlv (Max, Size : natural) return std_logic_vector ;
impure function RandUnsigned (Size : natural) return Unsigned ;
impure function RandUnsigned (Max, Size : natural) return Unsigned ;
impure function RandSigned (Size : natural) return Signed ;
impure function RandSigned (Max : integer;  Size : natural ) return Signed ;
```

Summary

The most typical flow of the **RANDOMPKG** package usage looks like this:

Stage	*Description*	
Attaching package	`use work.RandomPkg.all;` `use my_packages.RandomPkg.all;`	
Declaring object	`variable RndX : RandomPType;`	
Initialization	`RndX.InitSeed(RndX'instance_name);`	
Generation	`X := RndX.Uniform(1,6);` `X := RndX.Normal(5.0, 2.0, 1, 9);`	`RndX.SetRandomParm(NORMAL, 4.0, 1.0);` `X := RndX.RandInt(0, 8);`

- Universal methods default to Uniform distribution if **SetRandomParm** was not called.
- Supported distributions include **Uniform**, **Favor_small**, **Favor_big**, **Normal** and **Poisson**.
- Generated values are **real**, **integer**, **std_logic_vector**, **unsigned** or **signed**.
- Random variables should not be shared between different processes.
- If multiple random variables are used, they should be initialized with different seeds.

Using COVERAGEPKG Package

The package allows creation of data structures required for functional coverage collection and provides subprograms that manipulate data during preparation, data collection and reporting stages. Beginning users can quickly create coverage models by following simple coding pattern:

1. Declare coverage object.
2. Generate bins and add them to a cover point or cross structure in the object.
3. Collect coverage data (at any convenient sampling event).
4. Use coverage data to control stimulus randomization.
5. Check if coverage achieved. Repeat steps 3 & 4 if needed.
6. Write report.

Experienced VHDL programmers can easily add new functionality such as traversing coverage data structures and creating extensive reporting/post-processing routines.

Special Data Types

To simplify creation of bins as series of **min..max** value ranges, the package creates two auxiliary data types: **RangeType** record type and unconstrained array type of **RangeType** elements called **RangeArrayType**.

```
type RangeType is record
  min : integer ;
  max : integer ;
end record ;
type RangeArrayType is array (integer range <>) of RangeType ;
```

In addition to value range, full bin specification requires some additional pieces of information. While the package uses dynamic data structures internally to store this kind of information, it also provides some additional data types that help users to enter bin data manually.

```
type CovBinBaseType is record
  BinVal    : RangeArrayType(1 to 1) ;
  Action    : integer ;
  Count     : integer ;
```

```
    AtLeast   : integer ;
    Weight    : integer ;
  end record ;
  type CovBinType is array (natural range <>) of CovBinBaseType;
  -- 'Action' field values for 'CovBinBaseType'
    constant COV_COUNT   : integer := 1 ;
    constant COV_IGNORE  : integer := 0 ;
    constant COV_ILLEGAL : integer := -1 ;
```

The **CovBinBaseType** record type stores:

- bin value range in the **BinVal** field,
- number of bin hits in the **Count** field,
- counting action (increment) in the **Action** field,
- coverage goal (minimal required value of **Count**) in the **AtLeast** field,
- randomization weight in the **Weight** field.

The **CovBinBaseType** is the element type of the unconstrained array type **CovBinType** describing one-dimensional structures of cover points.

To support manual entry of *N* dimensional bins used in crosses, the package defines a family of array types named **CovMatrix*N*Type** with elements of **CovMatrix*N*BaseType** (with values of *N* in 2..9 range).

The dynamic coverage data structures and subprograms to operate them are packaged in a protected type **CovPType**. User of the package should declare one variable (object) of this type for each cover point or cross in the design. Variables can be local to the processes, but should be declared as shared variables in the architecture if coverage data collection and reporting/processing are done in different processes.

```
shared variable XCov : CovPType;
```

Generating Bins

To allow more flexible operations on bins, **GenBin** functions are defined as independent subprograms in the **CoveragePkg** package, not as the methods of **CovPType** protected type.

The signature of the base version of **GenBin** function is shown below:

```
function GenBin(
  constant AtLeast       : in integer ;  -- coverage goal
  constant Weight        : in integer ;  -- randomization weight
  constant Min, Max      : in integer ;  -- range of covered values
  constant NumBin        : in integer    -- number of bins in a range
) return CovBinType ;
```

Please note that the number of items in the array returned by the function call is equal **NumBin**.

There are overloaded versions of **GenBin** function created for easier bin creation:

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

Two argument version of the function creates **Max-Min+1** bins covering one value each. One argument version creates one bin covering argument value.

The **GenBin** function creates bins with **Action** field set to COV_COUNT (=1). If users need bins with values that should be ignored, they should use **IgnoreBin** function instead. It is available in 5 variants mirroring **GenBin**, but setting **Action** field to COV_IGNORE (=0). Please note that two argument version creates just one bin covering the entire range.

When users need bins that should trigger error when hit, they should use **IllegalBin** function. It does not set **AtLeast/Weight** fields and sets **Action** field to COV_ILLEGAL (=-1).

Results of all bin-generating function calls can be concatenated using "&" operator to create longer bin structures.

Creating Cover Points

To create dynamic, one-dimensional cover point data structure within coverage object, users should call **AddBins** method of the **CovPType** type. This procedure is available in 3 versions:

```
procedure AddBins (
  AtLeast : integer ;
  Weight  : integer ;
  CovBin : CovBinType
) ;
procedure AddBins (AtLeast : integer ;  CovBin : CovBinType) ;
procedure AddBins (CovBin : CovBinType) ;
```

The three-argument version of the procedure lets users to specify goal and randomization weight. The remaining two versions assume that goal and weight are 1 if not specified. The **CovBin** argument in the **AddBins** method call is usually specified as a call of **GenBin** function:

```
XCov.AddBins(2, 1, GenBin(0, 255, 16));
```

The **AddBins** method can be called incrementally, i.e. subsequent calls will add new bins to the ones created by the previous calls.

Creating Crosses

To create dynamic, multi-dimensional cross data structure within coverage object, users should call **AddCross** method of the **CovPType** type. This procedure is available in 3 versions similar to **AddBins**, but accepting from 2 to 20 arguments of CovBinType type (one for each dimension of the cross). Sample method call creating two-dimensional cross with 8 × 8 matrix of 1 value bins is shown below:

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

Slightly more complicated cross structure is can be created by this call:

```
XYCov.AddCross( 3, GenBin(0, 3, 2), GenBin(0, 3, 2) );
```

This time four bins are created: (0..1)×(0..1), (0..1)×(2..3), (2..3)×(0..1) and (2..3)×(2..3). Each bin covers four pairs of values and requires 3 hits to achieve coverage goal.

The **AddCross** method can also be called incrementally, e.g. to build cross structure row-by-row or column-by-column. The number of cross dimensions should not change in the incremental calls.

Sampling Data

The procedure method called **ICover** should be called to collect data whenever sampling event occurs. Depending on the design where coverage data is collected, the sampling event can be a clock edge, the end of bus transaction or some other, suitable event. The method accepts integer argument for objects containing cover points and integer vector argument for objects containing crosses. The use of conversion function may be required if sampled data is non-integer.

```
DCov.ICover(to_integer(Data));
XYCov.ICover((X, Y));
```

Checking Coverage Status

Users should call function method **IsCovered** to check if coverage goal was achieved.

```
impure function IsCovered ( PercentCov : real := 100.0 ) return boolean ;
```

If no argument is specified in the method call (or if it is set to 100.0), all individual bin goals must be achieved to guarantee **True** result. Specifying lower percentage threshold gives **True** results when some individual bin goals were not reached yet.

Most typical use of **IsCovered** method is the exit condition in the loop collecting coverage data:

```
while not DCov.IsCovered loop
 . . . -- data collection
```

If more data about coverage status is needed, the following function method can be used:

```
impure function CountCovHoles ( PercentCov : real := 100.0 ) return integer ;
```

The function returns the number of holes – bins that have not reached their goals.

Reporting

The detailed report of all bins contents can be printed to the console using **WriteBin** method call. The method is overloaded to allow writing the same report to a file; user has to specify output file name and (optionally) file open kind in this case.

```
DCov.WriteBin ("Dcovrep.txt", OpenKind => WRITE_MODE );
```

If open kind is not specified, new data will be appended at the end of current file contens.

To report only uncovered bins (coverage holes) the following procedure methods can be used:

```
procedure WriteCovHoles ( PercentCov : real := 100.0 ) ;
procedure WriteCovHoles ( FileName : string;  PercentCov : real := 100.0 ;
                          OpenKind : File_Open_Kind := APPEND_MODE ) ;
procedure WriteCovHoles ( AtLeast : in integer ) ;
procedure WriteCovHoles ( FileName : string;  AtLeast : in integer ;
                          OpenKind : File_Open_Kind := APPEND_MODE ) ;
```

If current coverage database must be saved and post processed (or reloaded) later, **WriteCovDb** method can be used:

```
DCov.WriteCovDb ("Dcovdb.txt", OpenKind => WRITE_MODE );
```

Intelligent Stimulus Randomization

One of the key features of the package is the ability to randomize stimulus based on current coverage results. The base function method created for this purpose is:

```
impure function RandCovPoint (PercentCov : real := 100.0) return integer_vector;
```

The operation of the function can be described like this:

- Create list of all bins with current coverage below specified **PercentCov** value.
- Use **RANDOMPKG** features to generate random value (**integer_vector**) that belongs to one of the uncovered bins.

Please note that the random number generator should be properly initialized (seeded) to guarantee high quality of generated numbers. To initialize generator, users should call **InitSeed** method once before the first call of **RandCovCoint**. The value of **instance_name** attribute of the coverage variable can be used as the reliable seed:

```
XYCov.InitSeed(XYCov'instance_name);
 . . .
while not XYCov.IsCovered loop
    wait until rising_edge(CLK);
    (X, Y) := XYCov.RandCovPoint;
     . . .
```

Without the use of intelligent coverage – just with randomly selected values from the set of N – we need approximately NlogN trials to select each value at least once. With intelligent coverage the number of trials is reduced to N. It means that in typical cases intelligent coverage reduces time to achieve coverage goal to less than 50%.

Summary

This document presented most important information needed to use the **COVERAGEPKG** package. Additional information about advanced features and support for legacy versions can be found in the official package documentation. The typical usage patterns of the package summed up here:

Stage	*Description*	
Attaching package	`use work.CoveragePkg.all;` `use my_packages.CoveragePkg.all;`	
Declaring object	`shared variable CovX, XYCov : CovPType;`	
Generating bins	`GenBin(0,7);  -- 8 bins, 1 value each` `GenBin(0,255,16); -- 16 equal size bins`	
Creating cover points/crosses	`CovX.AddBins(GenBin(0,31,8));` `CovX.AddBins(GenBin(32,47,1));`	`XYCov.AddCross(GenBin(0,7), GenBin(0,7));`
Sampling	`CovX.ICover(X);`	
Status Check	`if CovX.IsCovered then`	
Hole Count	`NotCov := CovX.CountCovHoles;`	
Randomize Stim.	`X := CovX.RandCovPoint; -- pick uncovered values`	
Print Report	`CovX.WriteBin; -- can be quite long...`	
Dump Database	`CovX.WriteCovDb ("covdb.txt", OpenKind => WRITE_MODE );`	

- Coverage object should be declared as shared variable if it is used in more than one process (e.g. data collection and reporting). It can be declared as local process variable otherwise.
- Sampling event is outside of the scope of this coverage package: any event-detecting condition can be used to trigger **ICover** call.
- Proper seeding of random number generator is needed when stimulus randomization is used.

FIFO Example Description

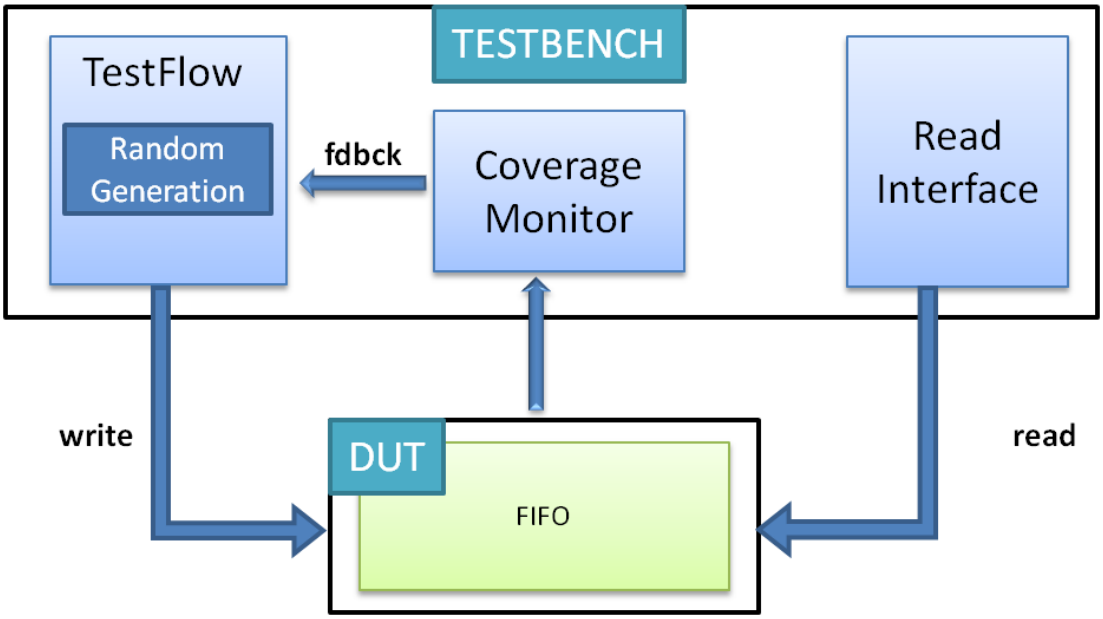
With the FIFO design we illustrate the usage of OS-VVM's Randomization and Coverage packages to build a self-checking, self-adjusting testbench.

FIFO Example Architecture

The FIFO design includes both DUT (Design Under Test), i.e. the FIFO memory model itself, and the testbench parts. With DUT being simple we put the focus on the testbench to demonstrate the usage of randomization and functional coverage OS-VVM's packages.

The testbench includes three main processes. The TestFlow process generates the burst write transactions for the FIFO. It randomizes the length of the burst write operation as well as the value of each word in those bursts. The CoverageMonitor process collects functional coverage statistics and also provides the feedback for random generation to increase the probability of hitting the uncovered points.

The ReadInf process will read the data from the FIFO whenever it is not empty.



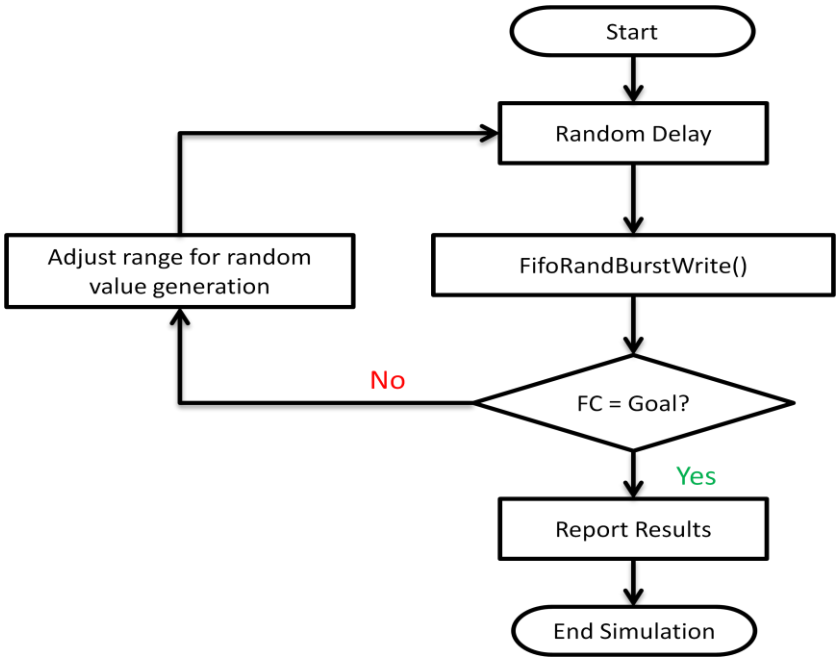
Test Flow

The TestFlow process operates at the transaction level. There is a random delay cycle before each burst transaction starts. The delay range is dynamically adjusted which helps to either empty out or fill the fifo.

FifoRandBurstWrite is a procedure that generates burst write transactions to the fifo.

At the end of each burst transaction the functional coverage (FC) is being calculated and compared to the goal, which is by default set to 100%. If the goal is not reached and the FC has not improved since

the last transaction then the random range for the delay cycle and the burst length will get adjusted, and then next cycle starts.



Random delay

The duration of the delay cycle is randomly generated using RandomPkg's RandInt (Min, Max : integer) function:

```
delay := RV.RandInt(mindelay, maxdelay);
wait for delay * clk_period;
```

RandInt function returns a random integer value in the given range. In this example we are using mindelay and maxdelay variables to control the duration of the wait cycle.

Burst Write Transaction

FifoRandBurstWrite procedure generates the bursts of the fifo write transactions. The data that is being written is generated using RandomPkg's *RandSlv(Min, Max, Size : natural)* function that returns a value of std_logic_type in the given range [min..max] and with the given size. In our example the 8-bit word values are generated in the range between 0 and 255.

```
for i in 1 to len loop
--Creating the random value to be sent to fifo
wordgen := RV.RandSlv(0, 255, 8);
--writing the word to fifo
FifoWriteWord(wordgen, FifoWrInf);
end loop;
```

FifoWriteWord function implements the write transaction to the fifo DUT. Each write transaction happens on the rising edge of a clock signal and takes one clock cycle.

```
procedure FifoWriteWord (
   word : in std_logic_vector(7 downto 0);
   signal FifoWrInf : out FifoWrInfType ) is
```

```
begin
   --setting the bus for fifo
   FifoWrInf.we <= '1';
   FifoWrInf.datain <= word;
   wait until rising_edge(clk);
   --clearing write enable
   FifoWrInf.we <= '0';
end procedure;
```

The FifoWriteWord transactions are applied to the DUT through the FifoWrInf variable of VHDL record type:

```
type FifoWrInfType is record
   we : std_logic;
   datain : std_logic_vector(data_width-1 downto 0);
end record;
signal FifoWrInf : FifoWrInfType;
```

Coverage Monitor

The CoverageMonitor process collects the functional coverage throughout the simulation. The functional coverage of the current example is based on the following three points:

1. fifo gets full – based on *full* signal
2. fifo gets empty – based on *empty* signal
3. write transaction while fifo is full: based on simultaneously active *we* and *full* signals

The following condition is considered illegal:

4. read attempt from the empty fifo

At the beginning of the CoverageMonitor procedure we create bins for each of the above conditions.

We only add bins for logic value '1' for the first two coverage points as we don't care for when those signals are low. This means that the point will be covered when the signals associated with those coverage points (see below) will get assigned value '1' during the simulation.

```
-- coverage point 1 - for fifo's full signal (High)
cp1_full.AddBins(GenBin(1));
-- coverage point 2- for fifo's empty signal (High)
cp2_empty.AddBins(GenBin(1));
```

The third coverage point has being added a cross condition with two bins. Both bins only care for logic value '1'. Coverage point 3 will be considered covered when two signals associated with the respective bins (see below) will simultaneously get value '1' during the simulation.

```
-- coverage point 3 - cross coverage for simultaneous FifoWrInfType.we and full
signal
cp3_cross_we_full.AddCross(GenBin(1), GenBin(1));
```

In the 4th coverage point we create illegal bin for the condition that never supposed to happen: reading from an empty fifo.

```
-- coverage point 4 - creating illegal bin when empty and re are high at the same
time
cp4_illegal_re_empty.AddBins(IllegalBin(1));
```

Once all bins have been added to the respective cover points the 'while' loop starts and runs until the first three points are covered:

```
--collecting coverage
MainCovLoop: while not (cp1_full.IsCovered and cp2_empty.IsCovered and
cp3_cross_we_full.IsCovered)  loop
```

Inside the while loop we will sample each point if the coverage goal has not been reached yet (*IsCovered* =False) and only if it is the first occurrence in the current transaction (status_notFull = '1'). When FIFO gets full (full = '1') we clear the status_notFull flag that will disable the coverage counter from incrementing coverage on the consecutive clock edges. In other words if fifo's full signal was high for N clock cycles, we want that to be considered as a single occurrence rather than N occurrences.

Then we check if the coverage goal has been reached (cp1_full.*IsCovered)* and display the respective message on the console with the time stamp and, later, detailed bin's information by the *WriteBin* procedure.

```
--check if cp1 is covered
if not (cp1_full.IsCovered = FALSE and status_notFull = '1') then
   --if not then sample it
   cp1_full.ICover(to_integer(full) ) ;
   if (full = '1') then
         status_notFull := '0';
         if (cp1_full.IsCovered) then
               Message("Covered condition *FIFO Full* @ " & time'image(now) );
         else
               Message("Hit condition *FIFO Full* @ " & time'image(now) );
         end if;
         cp1_full.WriteBin;
   end if;
end if;
```

The same checks and sampling are performed for the 2nd and 3rd cover points.

Timeout check

At the end of each cycle of the coverage monitor loop we check for the timeout and exit the loop if the current simulation time exceeds the TimeOut value:

```
exit MainCovLoop when now >= TimeOut;
```

The timeout check has been used to prevent the test from running forever when it cannot hit all the coverage points.

Reporting and stopping the simulation

Once either all points are covered or the timeout value is exceeded, the CoverageMonitor process exits the *MainCovLoop* loop, performs final reporting to the console and stops the simulation by assigning *end_sim* to '1'. All testbench processes suspend their execution when *end_sim* is assigned to '1' which stops the simulation.

Running the example in Active-HDL

Follow the steps below to compile and simulate the FIFO example in Aldec's Active-HDL version 8.3 or later.

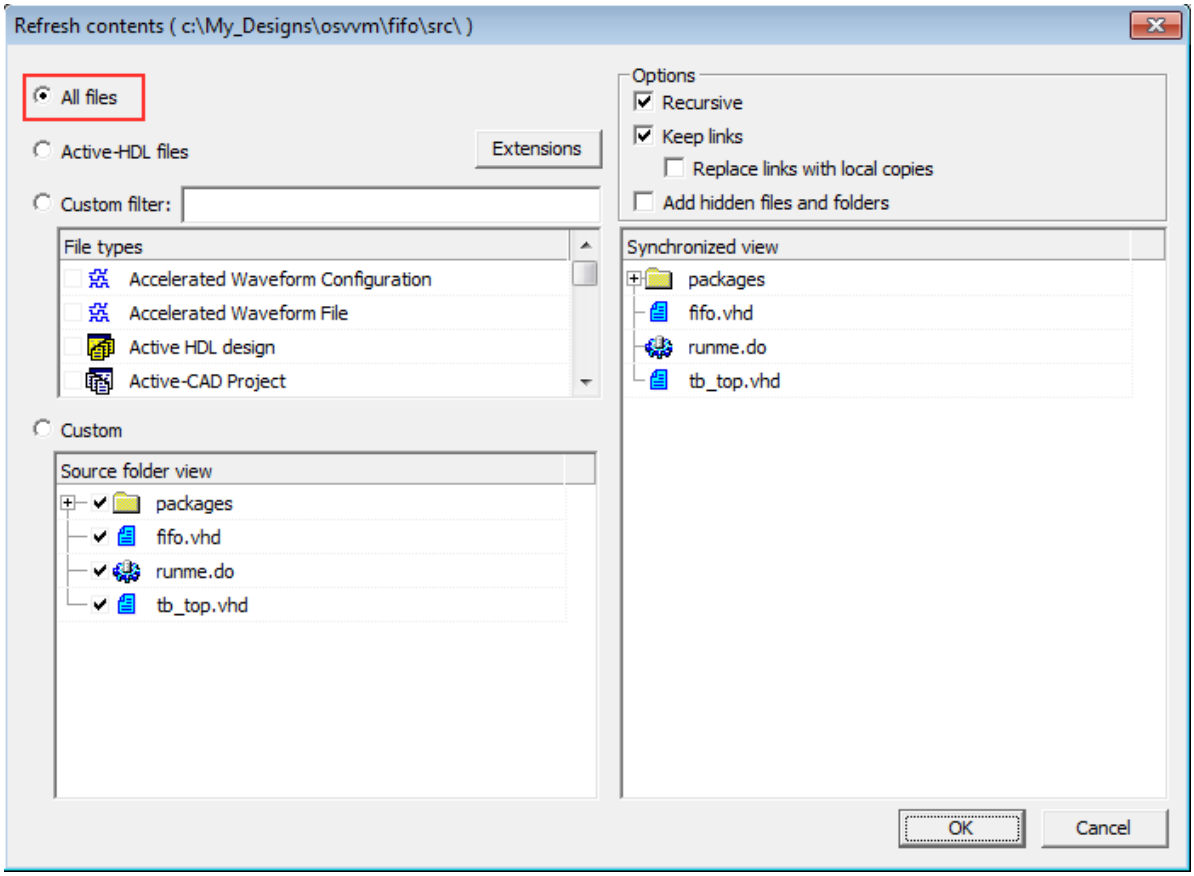
1. Launch Active-HDL and create a new workspace and design by going to menu File | New | Workspace. Check 'Create an Empty Design' option while going through the process. While creating a new design avoid using '-' in design name.

   E.g. Wrong Names: OS-VVM, fifo-1

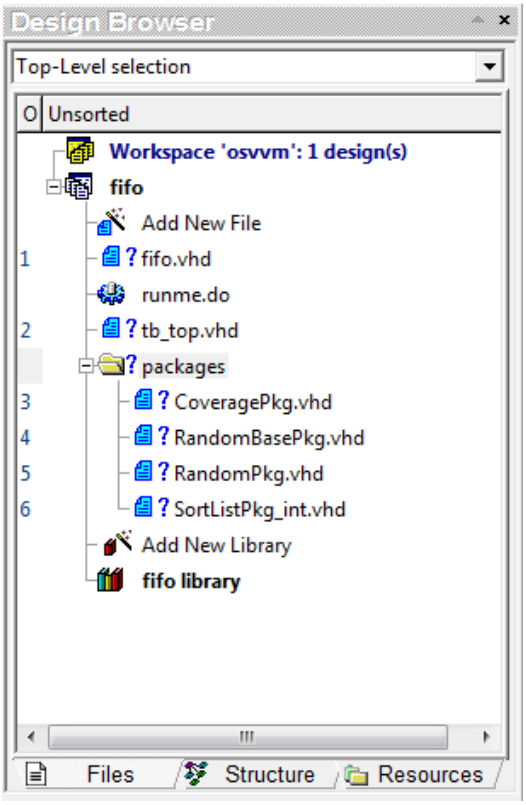
   Right Names: OSVVM, fifo_1

2. Once the design and workspace have been created you need **to copy** the items from the following two folders in the OSVVM root folder **and place them** to the *src* folder of your Active-HDL design:
   1) The entire p*ackages* folder with the included files
   2) The FIFO source files located in *OSVVM\example\FIFO\src* folder

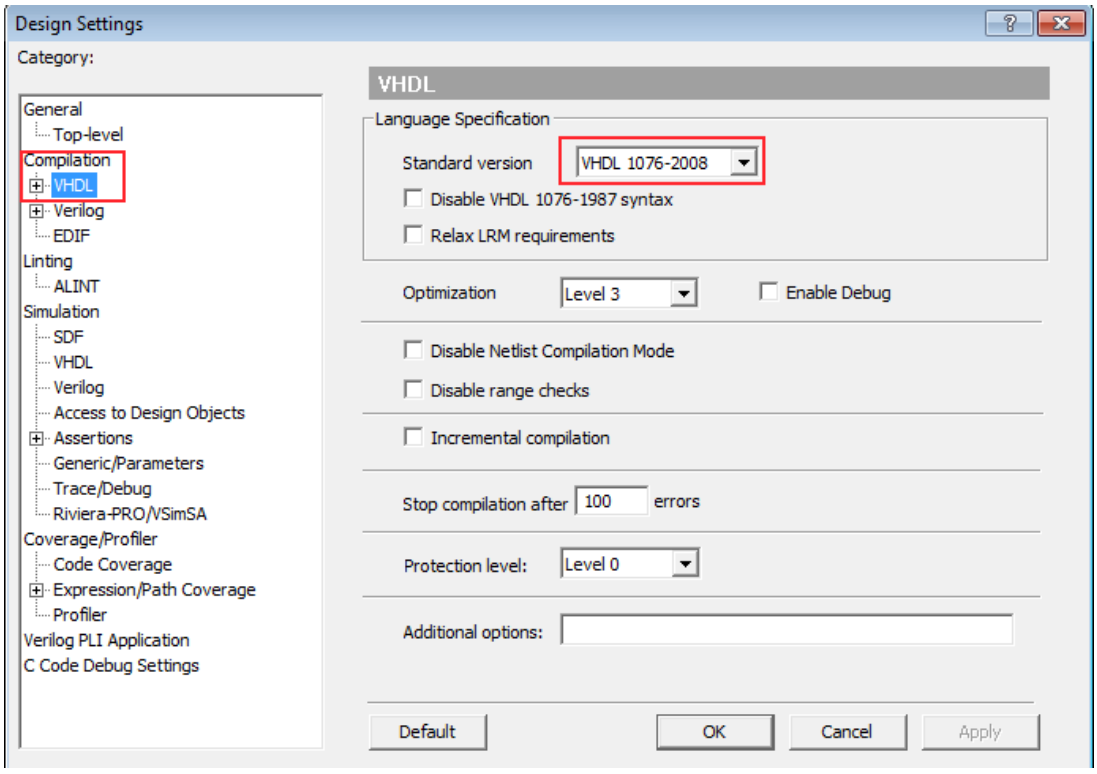
3. After the above files were copied to the Active-HDL's *src* folder, you need **to add** the source files to your Active-HDL design. The best way to do that is to right click on the design name in Active-HDL's Design Browser window and select "Refresh Contents" from the context menu. Select "All file" option at the top. This should checkmark the packages folder and all other vhd and do files in the *Source folder view*:



4. Press OK button to get all your source files added to the design.



5. Change your VHDL compilation settings by going to *Design* | *Settings* | *Compilation* | *VHDL*. Set *Standard version* to VHDL 1076-2008 as shown below:



6. After that you can right click on your design name and select *Compile All with File Reorder* option. This will get all source files compiled.
7. Initialize your simulation using tb_top as your top level entity.
8. Switch to the *Structure tab* of the *Design Browser* and add the signals from the DUT instance to the *Waveform*. In the Structure tab browse to DUT, right click on it and select add to waveform.
9. Click on *Run (Alt+F5)* button to run the simulation.
10. Once the simulation is finished you can observe the following signals in the *Waveform* viewer:
    - wr_clk, wr_en, wr_data – for write transactions
    - rd_clk, rd_en, rd_data – for read transactions
11. In the *Console* window you may observe the messages about the cover points with the time stamps as well as the final report at the end of the simulation.

Running the example in Riviera-PRO

Follow the steps below to compile and simulate the FIFO example in Aldec's Riviera-PRO version 2011.02 or later.

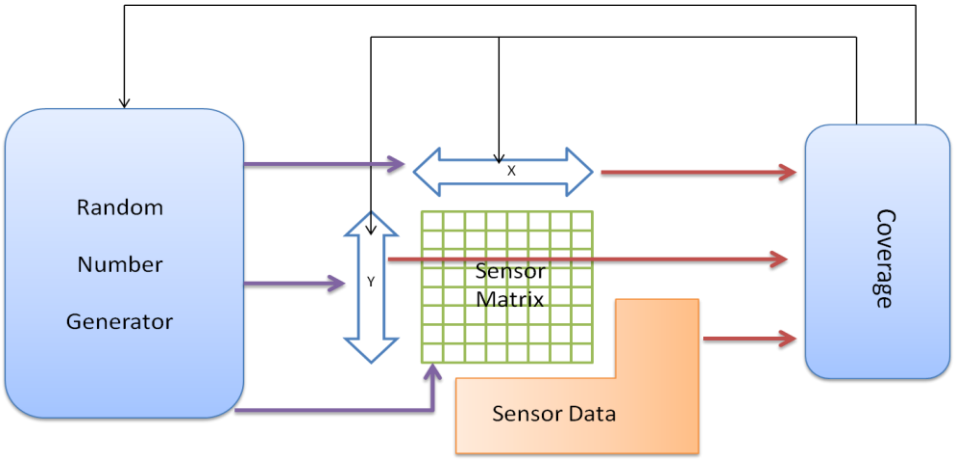
1. You can either create a copy of the original design or run directly in OSVVM/example/FIFO folder.
2. Launch Riviera-PRO and set the current directory (*File* | *Change Directory*) to where you stored the example: <YourPath>/example/FIFO
3. In the *File Browser* window of Riviera-PRO expand the src folder so that you can see the files inside, right click on the runme.do file and select *Execute* from the context menu. Observe the compilation and simulation messages in the console.
4. The runme.do script logs all of the design signals so they will be available at any point of the simulation. Switch to the *Debug* perspective of Riviera-PRO (by using the Debug perspective button at the bottom of your Riviera-PRO main window). In Debug perspective you will see the *Hierarchy Viewer* and the *Object Viewer* on the right.
5. Once the DUT signals are added to the waveform you can observe the signals such as
    - wr_clk, wr_en, wr_data – for write transactions
    - rd_clk, rd_en, rd_data – for read transactions
6. In the *Console* window you may observe the messages about the cover points with the time stamp as well as the final report at the end of the simulation.

Matrix Example Description

Introduction

This design has been created for demonstration of the advanced features of the functional coverage package. The design represents the abstract model of 8x8 matrixes of sensors. Each sensor has an 8 bit data register. The data for the registers is generated randomly. The testbench is checking for two types of coverage:

1. Picking up a sensor node randomly until all sensors are picked. The intelligent coverage feature has been used to accomplish that.
2. Data coverage is collected across all sensor registers to make sure all data ranges were hit by the random generator.



Data Generation

8-bit binary data is generated randomly with Normal distribution; operator of the simulator can modify mean value, standard deviation and increment of standard deviation of the distribution using three generic parameters:

- DataMean
- DataSDstart
- DataSDinc

The SensorData is randomly generated using RandomPkg's Normal (Mean, StdDeviation, Min, Max: real) function:

```
Rsens.Normal(DataMean, DataSD, 1, 255);
```

Above function returns real values between 1 and 255 with given mean and standard deviation. Normal is a Gaussian distribution of the data.

This randomly generated data is written inside 8x8 matrix indices. To speed up coverage goal achievement, test produces first batch of sensor data using DataMean and DataSDstart values as distribution parameters. If coverage is not achieved, current DataSD (standard deviation) value is incremented by DataSDinc to flatten Normal distribution curve and improve coverage. Setting DataSDinc to 0.0 turns this feature off and increases number of iterations needed to achieve coverage. The test prints number of iterations needed to fully cover all data values at the end of simulation.

Functional Coverage

Sensor data is collected by randomly selecting sensor indices in the 8x8 matrix for coverage purpose and all indices are also checked against coverage. If generic parameter Intelligent is set to true, index coverage results are used to control selection of the next index pair. If the generic is set to false, indices are generated with uniform distribution, which more than doubles the number of iterations needed to achieve index coverage.

```
(X, Y) := XYCov.RandCovPoint;
```

XYCov is shared variable used to handle coverage package procedures. RandCovPoint randomizes only uncovered indices if Intelligent is set to true. If Intelligent is set to false then X and Y values are selected randomly every time irrespective of coverage using functions:

```
X := Rxy.RandInt(0,7);
```

```
Y := Rxy.RandInt(0,7);
```

Irrespective of Intelligent generic above, the values read from the matrix of sensors are checked for the data coverage:

```
DCov.ICover(sensors.get(X, Y));  -- collect coverage for data
```

DCov is a shared variable and ICover is a procedure which is used to cover different bins.

The indices coverage is also being collected:

```
XYCov.ICover((X, Y));
```

The indices coverage is being used as the loop iterator merely to demonstrate the intelligent coverage feature.

Reporting

```
XYCov.WriteCovDb ("quicktest_XYcovdb.txt", OpenKind => WRITE_MODE );
```

WriteCovDB is used to generate database file for later use.

Compilation and simulation script contains three simulation commands (two of them commented out) with different values of generics set for given simulation run. The middle command gives decent results with intelligent features turned on. The other two produce long, un-optimized simulation and fast, maximally optimized simulation.

Contacting Aldec

To contact Aldec with any questions about OS-VVM please visit *www.aldec.com/support*, log in (or register) and open a new support case with your question.

Resources

OS-VVM page on the Aldec website:
http://www.aldec.com/en/solutions/functional_verification/os_vvm

About Aldec, Inc.

Established in 1984, Aldec Inc. is an industry leader in Electronic Design Verification and offers a patented technology suite including: RTL Design, RTL Simulators, Hardware-Assisted Verification, Design Rule Checking, IP Cores, DO-254 Functional Verification and Military/Aerospace solutions. Continuous innovation, superior product quality and total commitment to customer service comprise the foundation of Aldec's corporate mission. For more information, visit *www.aldec.com*.

About SynthWorks

SynthWorks provide trainings in leading edge VHDL verification techniques, including transaction based testing, bus functional modeling, self-checking, data structures (linked-lists, scoreboards, memories), directed, algorithmic, constrained random, and coverage driven random testing, and functional coverage. For more information, visit *www.synthworks.com*