

Randomization Using RandomPkg

User Guide for Release 2013.05

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

<http://www.SynthWorks.com>

Table of Contents

1	RandomPkg Overview.....	3
2	Randomization Using IEEE.math_real.uniform = Yuck!	3
3	Simplifying Randomization	3
4	Manipulating the Seeds	4
5	Basic Randomization	4
6	Large Vector Randomization	5
7	Weighted Randomization	6
8	Usage.....	6
9	Creating a Test	8
10	Random Stability.....	9
11	Other Distributions	10
12	Compiling RandomPkg and Friends.....	11
13	About RandomPkg	11
14	Future Work	12
15	About the Author - Jim Lewis	12
16	Revision Guide	13
16.1	2013.05 May 2013	13
16.2	2013.04 April 2013.....	13
16.3	2.2 June 2012.....	13
16.4	2.1 July 2011	13

1 RandomPkg Overview

RandomPkg provides types subprograms and protected type methods to simplify randomization. As such, it is at the heart of SynthWorks' constrained and coverage driven random methodology.

This documentation provides details on the STANDARD revision 2.2 of RandomPkg. As such this documentation supercedes the slides presented in the webinar in March of 2009. These packages are updated from time to time and are freely available at <http://www.synthworks.com/downloads>.

2 Randomization Using IEEE.math_real.uniform = Yuck!

A basic form of randomization can be accomplished by using the procedure uniform the IEEE math_real package. However, this always results in randomization being a multi-step process: call uniform to randomize a value, scale the value, and then use the value.

```
RandomGenProc : process
  variable RandomVal : real ;           -- Random value
  variable DataSent : integer ;
  variable seed1 : positive := 7 ;     -- initialize seeds
  variable seed2 : positive := 1 ;
begin
  for i in 0 to 255*6 loop
    uniform(seed1, seed2, RandomVal) ; -- randomize 0.0 to 1.0
    DataSent := integer(trunc(RandomVal*256.0)) ; -- scale to 0 to 255
    do_transaction(..., DataSent, ...) ;
    . . .
```

Optimally we would like to be able to call a function to do this so we can do this all in one step. Unfortunately we cannot write a normal VHDL function since we need to read and update the seed as well as return a randomized value.

3 Simplifying Randomization

RandomPkg uses a protected type, named RandomPType, to encapsulate the seed. Within a protected type, impure functions can read and update the seed as well as return a randomized value. To randomize values using the protected type, declare a variable of type RandomPType, initialize the seed, and randomize values.

```
RandomGenProc : process
  variable RV : RandomPType ;          -- protected type from RandomPkg
begin
  RV.InitSeed (RV'instance_name) ;    -- Generate initial seeds
  for i in 0 to 255*6 loop
    do_transaction(..., RV.RandInt(0, 255), ...) ; -- random value between 0 and 255
```

Note the calls to protected type methods (subprograms) include the protected type variable (RV) within the call (such as RV.RandInt(0, 255)).

4 Manipulating the Seeds

With protected types internal objects are private and accessible only through methods. The internal representation of the seed has a valid initial value, however, to ensure that each process' randomization is independent of each other, it is important to give each seed to a different initial value. As a result for a test that uses more than one randomization variable, initialize each seed once - if there is only one randomization variable, there is no need to initialize the seed.

The method `InitSeed` converts its argument value to `RandomSeedType` (the internal representation of the seed) and stores the value within the protected type. `InitSeed` is overloaded to accept either string or integer values. The preferred way to give each seed a unique value is pass the string value, `RV'instance_name`.

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

The methods `GetSeed` and `SetSeed` are used to read and restore a seed value. The declarations for these are shown below.

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

The function `to_string` and procedures `write` and `read` are used to write and read values of type `RandomSeedType`. The declarations for these subprograms are shown below. Note these are in `RandomBasePkg.vhd` and are separate from the protected type.

```
function to_string(A : RandomSeedType) return string ;
procedure write(L: inout line ; A : RandomSeedType ) ;
procedure read (L: inout line ; A : out RandomSeedType ; good : out boolean ) ;
procedure read (L: inout line ; A : out RandomSeedType ) ;
```

For a long test, it may be advantageous to read the seed periodically and print it out. If a failure or other interesting condition is generated, the seed may be restored to a value that was recorded near the failure with the intent of generating the error quickly to assist with debug.

5 Basic Randomization

The basic randomization generates an integer value that is either within some range or within a set of values. The set of values and exclude values are all of type `integer_vector` (defined in VHDL-2008). The examples below show the basic randomization overloading. When a value of `integer_vector` is specified, the extra set of parentheses denote that it is an aggregate value.

```
RandomGenProc : process
    variable RV          : RandomPType ;           -- protected type from RandomPkg
```

```

    variable DataInt : integer ;
begin
    RV.InitSeed (RV'instance_name) ;           -- Generate initial seeds

    -- Generate a value in range 0 to 255
    DataInt := RV.RandInt(0, 255) ;
    . . .
    -- Generate a value in range 1 to 9 except exclude values 2,4,6,8
    DataInt := RV.RandInt(1, 9, (2,4,6,8)) ;
    . . .
    -- Generate a value in set 1,3,5,7,9
    DataInt := RV.RandInt( (1,3,7,9) ) ; -- note two sets of parens required
    . . .
    -- Generate a value in set 1,3,5,7,9 except exclude values 3,7
    DataInt := RV.RandInt((1,3,7,9), (3,7) ) ;

```

These same functions are available for types `std_logic_vector`(`RandSlv`), unsigned (`RandUnsigned`) and signed (`RandSigned`). Note that parameter values are still specified as integers and there is an additional value used to specify the size of the value to generate. For example, the following call to `RandSlv` defines the array size to be 8 bits.

```

    . . .
    variable DataSlv : std_logic_vector(7 downto 0) ;
begin
    . . .
    -- Generate a value in range 0 to 255
    DataSlv := RV.RandSlv(0, 255, 8) ;

```

For randomizing within a range, there is also a `RandReal` function. Like procedure `Uniform`, it never generates its end values.

The overloading for the `RandInt` functions is as follows.

```

impure function RandInt (Min, Max : integer) return integer ;
impure function RandInt (Min, Max: integer; Exclude: integer_vector)
    return integer ;
impure function RandInt ( A : integer_vector ) return integer ;
impure function RandInt ( A : integer_vector; Exclude: integer_vector)
    return integer ;

```

6 Large Vector Randomization

Integer randomization is only valid in the range of -2^{31} to $+2^{31} - 1$. Large vector randomization uses multiple randomizations to create a value. The following overloading is available for `RandUnsigned`, `RandSlv`, and `RandSigned`.

```

impure function RandUnsigned (Size : natural) return unsigned ;
impure function RandUnsigned (Max : unsigned) return unsigned ;
impure function RandUnsigned (Min, Max : unsigned) return unsigned ;

```

The size parameter specifies the number of bits in the vector. The Max parameter allows randomization between 0 and Max. The Min and Max parameters allow randomizing a range of values.

7 Weighted Randomization

A weighted distribution randomly generates each of set of values a specified percentage of the time. RandomPType provides a weighted distribution that specifies a value and its weight (DistValInt) and one that only specifies weights (DistInt).

DistValInt is called with an array of value pairs. The first item in the pair is the value and the second is the weight. The frequency that each value will occur is $\text{weight}/(\text{sum of weights})$. As a result, in the following call to DistValInt, the likelihood of a 1 to occur is 7/10 times or 70%. The likelihood of 3 is 20% and 5 is 10%.

```
variable RV : RandomPType ;
. . .
DataInt := RV.DistValInt( ((1, 7), (3, 2), (5, 1)) ) ;
```

DistInt is a simplified version of DistValInt. The input to DistInt is an integer_vector of weights. The return value is the index of the selected weight. For a literal value, it will return a value from 0 to N-1 where N is the number of weights specified. As a result, the following call to DistInt the likelihood of a 0 is 70%, 1 is 20% and 2 is 10%.

```
variable RV : RandomPType ;
. . .
DataInt := RV.DistValInt( ((7, 2, 1)) ) ;
```

8 Usage

Each randomization result is produced by a function and that result can be used directly in an expression. Hence, we can randomize a delay that is between 3 and 10 clocks.

```
wait for RV.RandInt(3, 10) * tperiod_Clk - tpd ;
wait until Clk = '1' ;
```

The values can also be used directly inside a case statement. The following example uses DistInt to generate the first case target 70% of the time, the second 20%, and the third 10%.

```
variable RV : RandomPType ;
. . .
StimGen : while TestActive loop      -- Repeat until done
  case RV.DistInt( (7, 2, 1) ) is
    when 0 =>  -- Normal Handling      -- 70%
      . . .
    when 1 =>  -- Error Case 1        -- 20%
      . . .
    when 2 =>  -- Error Case 2        -- 10%
      . . .
    when others =>
```

```

        report "DistInt" severity failure ; -- Signal bug in DistInt
    end case ;
end loop ;

```

The following code segment generates the transactions for writing to DMA_WORD_COUNT, DMA_ADDR_HI, and DMA_ADDR_LO in a random order that is different every time this code segment is run. The sequence finishes with a write to DMA_CTRL. When DistInt is called with a weight of 0, the corresponding value does not get generated. Hence by initializing all of the weights to 1 and then setting it to 0 when it is selected, each case target only occurs once. The "for loop" loops three times to allow each transaction to be selected.

```

variable RV : RandomPType ;
. . .
Wt0 := 1; Wt1 := 1; Wt2 := 1; -- Initial Weights
for i in 1 to 3 loop          -- Loop 1x per transaction
    case RV.DistInt( (Wt0, Wt1, Wt2) ) is -- Select transaction
        when 0 =>                -- Transaction 0
            CpuWrite(CpuRec, DMA_WORD_COUNT, DmaWcIn);
            Wt0 := 0 ;           -- remove from randomization

        when 1 =>                -- Transaction 1
            CpuWrite(CpuRec, DMA_ADDR_HI, DmaAddrHiIn);
            Wt1 := 0 ;           -- remove from randomization

        when 2 =>                -- Transaction 2
            CpuWrite(CpuRec, DMA_ADDR_LO, DmaAddrLoIn);
            Wt2 := 0 ;           -- remove from randomization

        when others =>          report "DistInt" severity failure ;

    end case ;
end loop ;

CpuWrite(CpuRec, DMA_CTRL, START_DMA or DmaCycle);

```

The following code segment uses an exclude list to keep from repeating the last value. Note when passing an integer value to an integer_vector parameter, an aggregate using named association "(0=> LastDataInt)" is used to denote a single element array. Note that during the first execution of this process, LastDataInt has the value integer'left (a very small number), which is outside the range 0 to 255, and as a result, has no impact on the randomization.

```

RandomGenProc : process
    variable RV : RandomPType ;
    variable DataInt, LastDataInt : integer ;
begin
    . . .
    DataInt := RV.RandInt(0, 255, (0 => LastDataInt)) ;

    LastDataInt := DataInt;
    . . .
end process;

```

The following code segment uses an exclude list to keep from repeating the four previous values.

```
RandProc : process
  variable RV : RandomPtype ;
  variable DataInt : integer ;
  variable Prev4DataInt : integer_vector(3 downto 0) := (others => integer'low) ;
begin
  . . .
  DataInt := RV.RandInt(0, 100, Prev4DataInt) ;

  Prev4DataInt := Prev4DataInt(2 downto 0) & DataInt ;
  . . .
```

9 Creating a Test

Creating tests is all about methodology. SynthWorks' methodology marries randomization subprograms (from RandomPkg) and functional coverage subprograms (from CoveragePkg - also freely available at <http://www.synthworks.com/downloads>) with VHDL programming constructs. Each test sequence is derived by randomly selecting either branches of code or values for operations. Randomization constraints are created using normal sequential coding techniques (such as nesting of case, if, loop, and assignment statements). This approach is simple yet powerful. Since all of the code is sequential, randomized sequences are readily mixed with directed and algorithmic sequences.

A simple demonstration of randomizing is the following test which uses heuristics (guesses) at length of bursts of data and delays between bursts of data to randomization traffic being sent to a FIFO.

```
variable RV : RandomPType ;
. . .
TxStimGen : while TestActive loop
  -- Burst between 1 and 10 values
  BurstLen := RV.RandInt(Min => 1, Max => 10);
  for i in 1 to BurstLen loop
    DataSent := DataSent + 1 ;
    WriteToFifo(DataSent) ;
  end loop ;
  -- Delay between bursts: (BurstLen <=3: 1-6, >3: 3-10)
  if BurstLen <= 3 then
    BurstDelay := RV.RandInt(1, 6) ; -- small burst, small delay
  else
    BurstDelay := RV.RandInt(3, 10) ; -- bigger burst, bigger delay
  end if ;
  wait for BurstDelay * tperiod_Clk - tpd ;
  wait until Clk = '1' ;
end loop TxStimGen ;
```

Functional coverage counts which test cases have been generated and give engineers an indication of when testing is done. This is essential when using randomization to create a test as otherwise there is no way to know what the test actually did. Functional coverage can be implemented using subprogram calls (either custom or from the CoveragePkg) or VHDL code. Functional coverage is stored in signals and can be used to change the randomization (either directly as a constraint or indirectly as something that contributes to changing a constraint) to generate missing coverage items.

With a FIFO, we need to see lots of write attempts while full and read attempts while empty. One thing we can do to improve the previous test is to increase or decrease the burst length and delay based on the number of write attempts while full or read attempts while empty we have seen. To explore how to generate the coverage, see the CoveragePkg documentation.

For a design for which has numerous conditions we need to generate, we can do coverage on the input stimulus and then randomly select one of the uncovered conditions as the next transaction to be generated.

Solutions for the two previous coverage driven randomization problems are provided in SynthWorks' VHDL Testbenches and Verification class.

10 Random Stability

A protected type is always used with a variable object. If the object is declared in a process, it is a regular variable. If the object is declared in an architecture, then it is declared as a shared variable.

All of the examples in this document show RandomPType being defined in a process as a regular variable. This is done to ensure random stability. Random stability is the ability to re-run a test and get exactly the same sequence. Random stability is required for verification since if we find a failure and then fix it, if the same sequence is not generated, we will not know the fix actually worked.

Random stability is lost when a randomization variable is declared as a shared variable in an architecture and shared among multiple processes. When a randomization variable is shared, the seed is shared. Each randomization reads and updates the seed. If the processes accessing the shared variable run during the same delta cycle, then the randomization of the test depends on the order of which RandomPType is accessed. This order can change anytime the design is optimized - which will happen after fixing bugs. As a result, the test is unstable.

To ensure stability, create a separate variable for randomization in each process.

11 Other Distributions

By default, all randomizations use a uniform distribution. In addition to uniform distributions, RandomPType also provides distributions for FavorSmall, FavorBig, normal, and poisson. The following is the overloading for these functions.

```
-- Generate values, each with an equal probability
impure function Uniform (Min, Max : in real) return real ;
impure function Uniform (Min, Max : integer) return integer ;
impure function Uniform (Min, Max : integer ; Exclude: integer_vector) return
integer ;

-- Generate more small numbers than big
impure function FavorSmall (Min, Max : real) return real ;
impure function FavorSmall (Min, Max : integer) return integer ;
impure function FavorSmall (Min, Max: integer; Exclude: integer_vector) return
integer ;

-- Generate more big numbers than small
impure function FavorBig (Min, Max : real) return real ;
impure function FavorBig (Min, Max : integer) return integer ;
impure function FavorBig (Min, Max : integer ; Exclude: integer_vector) return
integer ;

-- Generate normal = gaussian distribution
impure function Normal (Mean, StdDeviation : real) return real ;
impure function Normal (Mean, StdDeviation, Min, Max : real) return real ;
impure function Normal (
    Mean          : real ;
    StdDeviation  : real ;
    Min           : integer ;
    Max           : integer ;
    Exclude       : integer_vector := NULL_INTV
) return integer ;

-- Generate poisson distribution
impure function Poisson (Mean : real) return real ;
impure function Poisson (Mean, Min, Max : real) return real ;
impure function Poisson (
    Mean          : real ;
    Min           : integer ;
    Max           : integer ;
    Exclude       : integer_vector := NULL_INTV
) return integer ;
```

The package also provides experimental mechanisms for changing the distributions used with functions `RandInt`, `RandSlv`, `RandUnsigned`, and `RandSigned`.

12 Compiling RandomPkg and Friends

Turn on the VHDL-2008 compile switch. Compile the files, `SortListPkg_int.vhd`, `RandomBasePkg.vhd`, and `RandomPkg.vhd`. We typically put these into a named library such as `SynthWorks` or `OSVVM`.

To take the packages for a test run, compile the program, `Demo_Rand.vhd`, into the same library as the packages and run it for 1 ns in your simulator.

Your programs need to reference `RandomPkg`. If your programs use IO for the seed (`to_string`, `write`, `read`), then you will also need to include `RandomBasePkg`.

```
library OSVVM ;
use OSVVM.RandomPkg.all ;
```

13 About RandomPkg

`RandomPkg` 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 `RandomPkg` and OSVVM by purchasing your VHDL training from SynthWorks.

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

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

If you find any innovative usage for the package, let us know - we can set you up to

14 Future Work

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

Things not documented in this document, such as type RandomParmType and method SetRandomParm, are experimental and may be removed in a future revision of the package (to reduce the overhead to basic randomization). Note that the current version of this package gives direct access to this capability via methods FavorSmall, FavorBig, normal, and poisson.

In addition to the RandomPkg, we also are freely distributing our coverage package, CoveragePkg. See <http://www.SynthWorks.com/downloads>. Over time we will also be releasing other packages that we currently distribute with our classes (such as scoreboards and memory modeling) and hope to convince simulation vendors to distribute our libraries with their tools.

15 About the Author - Jim Lewis

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

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

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

16 Revision Guide

16.1 2013.05

May 2013

Added big vector randomization.

16.2 2013.04

April 2013

Changed DistInt return value. The return value is now determined by the range of the input array. For literal values, this produces the same value as it did previously. Also added better error checking for weight values.

Added better min, max error handling in Uniform, FavorBig, FavorSmall, Normal, Poisson.

16.3 2.2

June 2012

Removed '_' in the name of subprograms FavorBig and FavorSmall to make more consistent with other subprogram names.

16.4 2.1

July 2011

Bug fix to convenience functions for slv, unsigned, and signed.