

# Constrained Random Verification with VHDL

By Jim Lewis

SynthWorks VHDL Training

[jim@synthworks.com](mailto:jim@synthworks.com)

<http://www.SynthWorks.com>

SynthWorks

---

## Constrained Random Verification with VHDL

SynthWorks

Copyright © 1999 - 2012 by SynthWorks Design Inc.

Reproduction of this entire document in whole for individual usage is permitted. All other rights reserved. In particular, no material from this guide may be reproduced and used in a group presentation, tutorial, or classroom without the express written permission of SynthWorks Design Inc.

SynthWorks is a service mark belonging to SynthWorks Design Inc.

This webinar is derived from the class, VHDL Testbenches and Verification

### Contact Information

Jim Lewis, President  
SynthWorks Design Inc  
11898 SW 128th Avenue  
Tigard, Oregon 97223  
503-590-4787  
[jim@SynthWorks.com](mailto:jim@SynthWorks.com)

[www.SynthWorks.com](http://www.SynthWorks.com)

# Constrained Random Verification

## In VHDL?

- While VHDL does not have built-in randomization constructs, most are easy to generate once we have a function.
- The foundation of this approach is the use of a protected type (VHDL-2002)
- Protected types are currently implemented by common VHDL simulators

## Topics

- Randomization with `ieee.math_real.uniform` (yuck)
- Data Structures for Randomization
- Setting the Seed Value
- Randomization with Uniform Distribution
- Randomization with Weighted Distribution
- Testing Using Randomization
- Functional Coverage
- Random Stability
- Future Work in VHDL Standards on Randomization

3

# Constrained Random Verification

## What is Constrained Random (CR) Verification?

- CR tests use randomization and programming constructs to create a set of values, operations, and/or sequences that are valid for a given environment

## When / Where to use it?

- CR is well suited to environments that have a diverse set of operations, sequences, and/or interactions that are difficult to cover completely

## When / Where not to use it?

- Tests with a finite set of operations - read / write all registers in a design
- Tests that can be algorithmically or numerically generated can often hit all interesting values quicker and/or more completely.

## Why use it?

- Where it works well, constrained random tests are faster to write, and hence, faster to verify your design.

## Randomization with Math\_Real

- In the package, IEEE.math\_real, there is a procedure named uniform.

```
procedure uniform(variable seed1, seed2 : inout positive; variable X : out real);
```

- The output, X, is a pseudo-random number with a uniform distribution in the open interval of (0.0 to 1.0)
- There are two seeds that are inout and must be in the following range:

```
1 <= seed1 <= 2147483562
1 <= seed2 <= 2147483398
```

5

## Randomization with Math\_Real (yuck)

- To use uniform as a basis for randomization, we must

```
RandomGenProc : process
  variable RandomVal : real ;
  variable DataSent : integer ;

  -- Declare seeds and initialize
  -- Uniform uses seeds as state information,
  -- so initialize only once
  variable DataSent_seed1 : positive := 7 ;
  variable DataSent_seed2 : positive := 1 ;
  . . .
begin
  . . .
  for i in 1 to 1000 loop

    -- Generate a value between 0.0 and 1.0 (non-inclusive)
    uniform(DataSent_seed1, DataSent_seed2, RandomVal) ;

    -- Convert to integer in range of 0 to 255
    DataSent := integer(trunc(RandomVal*256.0)) ;
```

Too much work to be an effective methodology by itself

6

## Data Structures for Randomization

- Use Procedures?
  - No. Makes randomization a two step process: get value, use value
- Use Functions?
  - Do not allow seed to be inout.
- Protected type = container type
  - Contains private variables.
  - Contains methods = procedures and functions
    - Procedures and impure functions can access private variables
  - Has a declaration and a body similar to a package
- For randomization, use protected type with
  - Seeds as private variables
  - Procedures to set the seeds
  - Impure functions to randomize values

7

## Data Structures for Randomization

```

type RandomPType is protected
  -- Initialize Seed
  procedure InitSeed (SeedIn : String ) ;
  -- Generate a value in range & range with exclude
  impure function RandInt (Min, Max : integer) return integer ;
  impure function RandInt (Min, Max : integer;
    Exclude: integer_vector ) return integer ;
  -- Generate a value in a set & a set with exclude
  impure function RandInt (A : integer_vector) return integer ;
  impure function RandInt (A : integer_vector ;
    Exclude: integer_vector ) return integer ;
  -- Distribution with just weights & weights with exclude
  impure function DistInt ( A : integer_vector ) return integer;
  impure function DistInt ( A : integer_vector ;
    Exclude: integer_vector ) return integer ;
  -- Distribution with weight and value & with exclude
  impure function DistIntVal ( A : DistType ) return integer ;
  impure function DistIntVal ( A : DistType ) return integer ;
    Exclude: integer_vector ) return integer ;
end protected RandomPType ;

```

8

## Data Structures for Randomization

- Protected type body for RandomPType (conceptual model)

```

type RandomPType is protected body
  -- private variable stores seed value
  variable RandomSeed : integer_vector(1 to 2) := (1, 7) ;

  -- InitSeed: Initialize Seed
  procedure InitSeed (S : string) is
  begin
    RandomSeed := GenRandomSeed(S) ;
  end procedure InitSeed ;

  -- RandInt: generate random integers in a range
  --   Calls uniform using seeds and
  --   Returns a value scaled in the specified range
  --   Impure to allow reading of seed values
  impure function RandInt (Min, Max : Integer) return integer is
    variable RandomVal, ValRange : real ;
  begin
    uniform(RandomSeed(1), RandomSeed(2), RandomVal) ;
    ValRange := real(Max - Min + 1) ;
    return integer(trunc(RandomVal*ValRange)) + Min ;
  end function RandInt ;
end protected body RandomPType ;

```

9

## Randomization with RandomPType

- To use RandomPType for randomization, we

```

RandomGenProc : process
  variable DataSent : integer ;

  -- Declare a variable of RandomPType - one per process
  variable RV : RandomPType ;

begin
  -- Initialize the Seed - unique value for each process
  RV.InitSeed( RV'instance_name ) ;

  for i in 1 to 1000 loop

    -- Do a transaction with value in range 0 to 255
    do_transaction(. . . , RV.RandInt(0, 255), . . . ) ;
  end loop
end process RandomGenProc ;

```

In call to InitSeed and RandInt, note the usage of RV

```

RV.InitSeed( RV'instance_name ) ;
RV.RandInt(0, 255) ;

```

## Initializing the Seed

- Method InitSeed translates its parameter into a legal seed value

```
procedure InitSeed (S : string ) ;
procedure InitSeed (I : integer ) ;
```

- Recommended: Use a string based on the process name

```
RV.InitSeed( S => RV'instance_name ) ; -- or 'path_name
```

- instance\_name and instance names include instance labels and give different instances of the same design different seeds.

- To give all instances the same seed, use the process name

- Integer Based

```
RV.InitSeed( I => 10 ) ;
```

11

## Randomization with a Uniform Distribution

- Randomize a value in the inclusive range, 0 to 15

```
DataInt := RV.RandInt( Min => 0, Max => 15 ) ;
```

- Randomize a value in the range (0 to 15) excluding values 3, 7, or 11

```
DataInt := RV.RandInt( 0, 15, ( 3, 7, 11 ) ) ;
```

- Note sets are integer\_vector and require parentheses

- Randomize a value within the set (1, 2, 3, 5, 7, 11)

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

- Randomize a value in the set (1, 2, 3, 5, 7, 11) excluding values (5, 11)

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

- There is also RandSlv, RandUnsigned, and RandSigned

```
DataSlv8 := RV.RandSlv( 0, 15, 8 ) ; -- 8 = length of array
```

## Randomization with Uniform Distribution

- Randomizing an enumerated type with uniform distribution

```

variable RV : RandomPType ;
type StateType is (IDLE, ONE, TWO, THREE) ;
variable RanState1, RanState2 : StateType ;
. . .

RanState1 := StateType'val( RV.RandInt(0, 3) ) ;

RanState2 := StateType'val(
    RV.RandInt(0, StateType'pos(StateType'right)) ) ;

```

- StateType'val returns the StateType value for a position number
- StateType'right returns the right most value of StateType (THREE)
- StateType'pos returns position number (range from 0 to N-1)

13

## Randomization with Weighted Distributions

- DistValInt: specifies value and weight
  - Input = unconstrained array of records with value and weight

```

variable RV : RandomPType ;
. . . -- ((val1, wt1), (val2, wt2), ...
RandVal := RV.DistValInt( ((1, 7), (3, 2), (5, 1)) ) ;

```

- % generated = weight / (Sum of weights)
- Generates 1:  $7 / (7 + 2 + 1) = 70\%$  of the time
- Generates 3:  $2 / (7 + 2 + 1) = 20\%$
- Generates 5:  $1 / (7 + 2 + 1) = 10\%$
- DistInt: specifies weight
  - Values range from 0 to N-1 (where N = # weights)
  - Input = integer\_vector that specifies the weight.

```

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

```

- Generates 0: 70%, 1: 20%, 2: 10%
- Both functions also support exclude vectors

14

## Testing Using Randomization

- Randomization functions return a value that can be used in expressions
- Generating Random Delay of 3 to 10 Clocks

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

- Randomly selecting one of 3 sequences (uniform distribution)

```
variable RV : RandomPType ;
. . .
case RV.RandInt(1, 3) is
  when 1 =>
    . . .
  when 2 =>
    . . .
  when 3 =>
    . . .
  when others =>   report "RandInt" severity failure ;
end case ;
```

15

## Testing Using Randomization

- Randomly select a sequence with a weighted distribution

```
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 ;
  end case ;
end loop ;
```

## Testing Using Randomization

- Randomizing order of 3 transactions

```

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);

```

17

## Testing Using Randomization

- Excluding the last value

```

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

```

- Excluding the four previously generated 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 ;
  . . .

```

18

## Testing Using Randomization

- FIFO Test: Create bursts of values with idle times between

```

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) ;
  else
    BurstDelay := RV.RandInt(3, 10) ;
  end if ;
  wait for BurstDelay * tperiod_Clk - tpd ;
  wait until Clk = '1' ;
end loop TxStimGen ;

```

19

## Functional Coverage

- Functional coverage is code that measures execution of a test plan
  - It tracks requirements, features, and boundary conditions.
  - Verifies that a constrained random generates all interesting conditions

- Define interesting conditions

```

ReadEmpty <= Empty and FifoRd when rising_edge(Clk);
WriteFull <= Full and FifoWr when rising_edge(Clk);

```

- Use VHDL-2008 external names to access a design signal.
- Count the Conditions = Functional Coverage
  - Allows use of coverage to algorithmically modify parameters of the test

```

ReadEmptyCov <= ReadEmptyCov + 1 when
    rising_edge(ReadEmpty) and nReset = '1' ;
WriteFullCov <= WriteFullCov + 1 when
    rising_edge(WriteFull) and nReset = '1') ;

```

- Alternately, toggle coverage can be used to track conditions.
  - ReadEmptyCov = ToggleCoverage(ReadEmpty)/2

## Functional Coverage

- For more powerful functional coverage support, see CoveragePkg
  - Available as open source at: <http://www.synthworks.com/downloads>
- CoveragePkg
  - Simplifies modeling and collecting high fidelity functional coverage
  - Implements both point and cross coverage
  - Contains methods for interacting with the coverage data structure
- Implements "Intelligent Coverage" = Coverage driven randomization
  - Randomizes across coverage holes
    - Minimizes redundant stimulus and reduces sim cycles
  - Replaces CR in the first level stimulus shaping.
    - Use CR and other methods to refine the stimulus.
    - Less work so reduces development time
  - Balances the randomization solution without a solver
- See OS-VVM webinar at: <http://www.synthworks.com/downloads>

21

## Random Stability

- Random stability is the ability to re-run a test and get the same stimulus
- Stability is effected by number of randomization variables
- ~~● One randomization variable (shared) per architecture or design
 
  - Randomization from different processes share the same seed.
  - If process execution order changes, order of randomization changes and randomized values (the stimulus) will change
  - Process execution order can change with compile or optimization
  - Fix a bug, recompile, and test may not produce the same stimulus
  - Test is unstable.~~
- One randomization variable per process
  - Each process has its own seed.
  - If order of randomization in process stays the same, test is stable.
- One randomization variable per randomization (as done in FifoTest)
  - Each item has its own seed.
  - The test is stable

## VHDL Randomization Summary

- Constrained Random = random values limited to valid range
  - Here we create the constraints procedurally (in code)
  - As such, it is easy to mix algorithmic code with constrained random
- Techniques can implemented in any VHDL environment
- Techniques
  - Randomize values with either uniform or weighted distributions
  - Randomize test scenarios with DistInt and Case Distribute pattern
  - Randomize ordering of sequences with DistInt
  - Randomize ad-hoc similar to BurstLen and BurstDelay in FIFO Test driving the design in a direction
  - Randomize based on observed coverage.
- Requires functional coverage to verify what was tested.
- Available as open source at: <http://www.synthworks.com/downloads>

---

## SynthWorks VHDL Training

SynthWorks is committed to supporting open source VHDL packages  
Support us by buying your training from us.

Comprehensive VHDL Introduction 4 Days

[http://www.synthworks.com/comprehensive\\_vhdl\\_introduction.htm](http://www.synthworks.com/comprehensive_vhdl_introduction.htm)

A design and verification engineer's introduction to VHDL syntax, RTL coding, and testbenches. Students get VHDL hardware experience with our FPGA based lab board.

VHDL Testbenches and Verification 5 days - OS-VVM Boot Camp

[http://www.synthworks.com/vhdl\\_testbench\\_verification.htm](http://www.synthworks.com/vhdl_testbench_verification.htm)

Learn the latest 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.

VHDL Coding for Synthesis 4 Days

[http://www.synthworks.com/vhdl\\_rtl\\_synthesis.htm](http://www.synthworks.com/vhdl_rtl_synthesis.htm)

Learn VHDL RTL (FPGA and ASIC) coding styles, methodologies, design techniques, problem solving techniques, and advanced language constructs to produce better, faster, and smaller logic.

For additional courses see: <http://www.synthworks.com>