Средства VHDL для функциональной верификации цифровых систем
Методология OS-VVM

Николай Авдеев, Пётр Бибило (г. Минск, Беларусь)

С помощью методологии OS-VVM демонстрируются возможности псевдослучайной генерации тестов и функционального покрытия при верификации цифровых систем.

Предлагаемая статья продолжает тематику функциональной верификации исходных VHDL-описаний цифровых систем [1]. Функциональная верификация в данном случае интерпретируется как проверка соответствия VHDL-описания проекта цифровой системы заданным требованиям. Для такой верификации предложена [2] методология OS-VVM (Open Source VHDL Verification Methodology), ориентированная на создание сложных тестирующих программ, а именно, позволяющая реализовать настраиваемую генерацию псевдослучайных тестов и функциональное покрытие.

В отличие от проблемно-ориентированных или прямых тестов [3, с. 463], настраиваемая генерация псевдослучайных тестов позволяет обнаруживать случайные ошибки в проектах. Функциональное покрытие (functional coverage) предназначено для измерения того, какая часть функций проекта была проверена во время выполнения моделирования [4, с. 323]. В методологии OS-VVM функциональное покрытие осуществляется сбором значений переменных и сигналов VHDL-проекта при выполнении моделирования. Методология OS-VVM основана на VHDL-пакетах *CoveragePkg* и *RandomPkg* стандарта VHDL 2008 [5].

Ниже на примерах иллюстрируется применение средств VHDL-пакетов *CoveragePkg* и *RandomPkg*. Предполагается, что читатели знакомы с основами языка VHDL и со статьёй [1], в которой описан тип *protected* (защищённый), широко используемый в указанных пакетах VHDL.

Традиционный подход

Рассмотрим генерацию псевдослучайных тестовых векторов на примере тестирования простейшего VHDL-проекта цифровой системы – умножителя *mult*.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mult is
port (a, b : in
  std_logic_vector (4 downto 1);
     d : out
  std_logic_vector (8 downto 1));
end mult;
architecture func of mult is
signal e : integer range 0 to 225;
begin
p0 : process(a, b)
  variable a_int, b_int :
   integer range 0 to 15;
 begin
  a_int := to_integer(unsigned(a));
  b_int := to_integer(unsigned(b));
  e <= a_int * b_int;
end process;
d <= std_logic_vector(to_unsigned(e,8));
end func;
```

Этот VHDL-проект задаёт описание компонента, предназначенного для перемножения целых положительных чисел *a* и *b*, заданных 4-разрядными двоичными векторами. Выходные сигналы устройства – это 8-разрядные двоичные векторы, представляющие числа из диапазона [0, 225], который обусловлен тем, что числа *a*, *b* принимают значения из диапазона [0, 15].

Умножитель описан на алгоритмическом уровне: с помощью функции *to_integer* пакета *numeric_std* осуществляется переход к численным значениям входных векторов, затем полученные числа перемножаются, после чего осуществляется преобразование произведения (числа *e*) в выходной вектор *d*.

Целью функциональной верификации блока умножителя является генерация и подача на его входы всех возможных пар псевдослучайных значений для *a* и *b*. При этом будет осуществляться проверка, все ли значения *a* и *b* перебраны и все ли различные пары *a* и *b* поданы на вход умножителя. Таких пар будет 256, поскольку *a* и *b* принимают значения из диапазона [0, 15]. Процесс генерации пар чисел *a* и *b* должен выполняться до достижения заданной цели.

По сути, такая цель верификации (тестирования) называется *функциональным покрытием*, которое осуществляется подсчётом значений переменных проекта по заранее определённым корзинам (bins) – диапазонам значений, имеющих специальное назначение в проекте. Для корзин, использующих только одну переменную, создаётся структура данных, которую называют точкой покрытия (coverage point). В нашем примере точка покрытия для числа *a* включает 16 корзин.

По корзинам перекрёстного покрытия (cross coverage) распределяются пары (тройки и т.д.) значений двух либо нескольких переменных. Для корзин, использующих две и более переменных, создаётся структура данных, называемая точкой перекрёстного покрытия. В нашем примере точка перекрёстного покрытия пары чисел *a* и *b* включает 256 корзин.

Напишем соответствующую тестирующую программу (см. листинг 1), используя традиционные средства VHDL – генератор псевдослучайных чисел, базирующийся на процедуре *uniform* пакета *math_real* VHDL-библиотеки IEEE.

Важное место в тестирующей программе *testbench* занимает подготовка псевдослучайных тестовых воздействий, подаваемых на вход компонента *mult*, т.е. тестовых векторов *a*, *b*. В строках 21–24 листинга 1 декларируются переменные для задания начальных значений генераторов чисел *a*, *b*. В процессе *RandomGenProc1* выполняются две процедуры (строки 30, 33) *ieee.math_real.uniform* (т.е. процедуры *uniform* из пакета *math_real* библиотеки IEEE) для генерации двух псевдослучайных вещественных чисел *RndValA*, *RndValB* из диапазона (0.0, 1.0) (не

включая граничные значения 0.0 и 1.0). Далее будем говорить о числах *a*, *b*.

Для генерации числа *a* процедуре *uniform* требуется задать начальные значения, например *SeedA1*, *SeedA2*; результатом выполнения процедуры будет псевдослучайное число *RndValA* и новые значения *SeedA1*, *SeedA2*. В строке 31 переменная *RndValA* умножается на 16.0, и с помощью функции *ieee.math_real.trunc* производится округление результата вниз до ближайшего целого (отбрасывание дробной части). Затем выполняется преобразование вещественного числа в целое число *RndA*, чтобы в итоге получить целые числа из диапазона [0, 15]. В строке 32 значение *RndA* с помощью функции *to_unsigned* пакета *numeric_std* преобразуется в вектор с требуемым числом разрядов (4), который назначается сигналу *a*, подаваемому на вход тестируемого компонента *mult*.

Аналогичным образом, в строках 33–35 всё повторяется для получения второго псевдослучайного числа *b*. Такие пары псевдослучайных чисел *a*, *b* генерируются в цикле *while* до тех пор, пока не будут получены все пары *a*, *b*. Поскольку неизвестно заранее, сколько циклов генерации псевдослучайных чисел потребуется для перебора всех возможных пар, в строке 43 задано большое число (1 000 000) итераций цикла по переменной *i*, чтобы цикл *MainCovLoop* не стал бесконечным. При выполнении условия *i* = 1 000 000 происходит выход из цикла *MainCovLoop* по команде *exit*.

Поясним операторы, предназначенные для проверки того, что все возможные пары *a*, *b* сгенерированы и подаются на вход *mult*. Для этого необходимо собирать статистику поданных значений чисел *a*, *b*. В строке 13 декларируются переменные *CovA* и *CovB*, представляющие собой массивы из 16 нулевых элементов. Каждый элемент массивов представляет собой корзину для одного определённого значения числа *a* или *b* соответственно. Значение элемента массива *CovA* будет соответствовать числу выпадений значения числа *a*, равного номеру элемента. Например, число выпадений значения *a* = 5 будет храниться в элементе 5 массива *CovA*, и если значение 5 выпадет 12 раз, то *CovA(5)* = 12. Переменные *CovA*, *CovB* имеют тип *integer_vector*, который введён стандартом VHDL 2008.

Перекрёстное покрытие. В строке 14 декларируется переменная *CovCrossAB*, предназначенная для хране-

Листинг 1. Тестирующая программа, использующая традиционные средства VHDL

```
1 library ieee; use ieee.std_logic_1164.all;
2 use ieee.math_real.all; use ieee.numeric_std.all;
3 entity testbench is
4 end;
5 architecture tb1 of testbench is
6  component mult
7   port (a, b : in std_logic_vector (4 downto 1);
8      d  : out std_logic_vector (8 downto 1));
9  end component;
10 signal a, b : std_logic_vector (4 downto 1);
11 signal d  : std_logic_vector (8 downto 1);
12 -- создание массивов (корзин) для a, b и пар (a,b)
13 shared variable CovA, CovB :
        integer_vector (0 to 15) := (others => 0);
14 shared variable CovCrossAB :
        integer_vector (0 to 255) := (others => 0);
15 begin
16 DUV : mult port map (a => a, b => b, d => d);
17 RandomGenProc1 : process
18  variable RndValA, RndValB : real;
19  variable RndA, RndB    : integer;
20 -- Начальные значения генератора
21 variable SeedA1          : positive := 7;
22 variable SeedA2          : positive := 1;
23 variable SeedB1          : positive := 4;
24 variable SeedB2          : positive := 2;
25 variable i, a_i, b_i       : natural := 0;
26 variable CovACovered, CovBCovered : boolean := false;
27 variable CovCrossABCovered    : boolean := false;
28 begin
29 MainCovLoop : while not CovCrossABCovered loop
30  uniform(SeedA1, SeedA2, RndValA);   -- генерация a
31  RndA := integer(trunc(RndValA*16.0));
32  a <= std_logic_vector(to_unsigned(RndA, 4));
33  uniform(SeedB1, SeedB2, RndValB);   -- генерация b
34  RndB := integer(trunc(RndValB*16.0));
35  b <= std_logic_vector(to_unsigned(RndB, 4));
36  wait for 10 ns;
37  a_i := to_integer(unsigned(a));
38  b_i := to_integer(unsigned(b));
39  CovA(a_i) := CovA(a_i) + 1 ;
40  CovB(b_i) := CovB(b_i) + 1 ;
41  CovCrossAB(b_i*16+a_i) := CovCrossAB(b_i*16+a_i) + 1;
42  i := i+1;
43  if i = 1000000 then exit; end if;
44  CovACovered := true;
45  for j in 0 to 15 loop     -- проверка покрытия a
46   if CovA(j)=0 then CovACovered := false; end if;
47  end loop;
48  CovBCovered := true;
49  for j in 0 to 15 loop     -- проверка покрытия b
50   if CovB(j)=0 then CovBCovered := false; end if;
51  end loop;
52  CovCrossABCovered := true;
53  for j in 0 to 255 loop  -- проверка покрытия пар a,b
54   if CovCrossAB(j)=0 then CovCrossABCovered := false;
55   end if;
56  end loop;
57 end loop;
58 wait for 10 ns; wait;
59 end process;
60 end architecture tb1;
```

ния числа выпадений значений пар *a*, *b* и представляющая собой массив целых чисел из 256 элементов (для чисел *a* и *b* имеется по 16 значений, таким образом, перекрёстных корзин или пар значений чисел *a*, *b* будет 16×16=256). В таблице 1 показана принятая нумерация перекрёстных корзин для значений *a*, *b* в массиве *CovCrossAB*. Например, число выпадений пары *a*=2, *b*=14 будет храниться в элементе 226 массива *CovCrossAB*.

В строках 39, 40 происходит сбор покрытия значений чисел *a* и *b*, представляющий собой увеличение на единицу значения элемента массива *CovA* и *CovB*, при этом номер элемента массива равен значению числа *a* и *b* соответственно. В строке 41 осуществляется сбор пар значений чисел *a* и *b*, попадающих в соответствующие корзины – элементы массива *CovCrossAB*.

В строках 26, 27 декларируются переменные *CovACovered*, *CovBCovered*, *CovCrossABCovered*, которые являются флагами, обозначающими полное покрытие всех соответствующих корзин *CovA*, *CovB*, *CovCrossAB*. В строках 44–56 выполняются три цикла для проверки полноты покрытия точек *CovA*, *CovB*, *CovCrossAB*. В строке 29 выполняется проверка условия работы цикла *while*, который исполняется до тех пор, пока значение флага *CovCrossABCovered* не станет истинным (*true*), т.е. до покрытия всех пар чисел *a*, *b*.

Анализ временной диаграммы, полученной после выполнения тестирующей программы (листинг 1), показал, что при использовании стандартного генератора случайных чисел с равномерным распределением вероятностей появления чисел, для получения всех 256 пар чисел *a*, *b* требуется 1615 циклов генерации. При этом покрытие корзин для числа *a* достигается за первые 103 цикла, а покрытие корзин для числа *b* – за первые 68 циклов.

Рассмотренный выше пример является иллюстративным. На практике приходится решать задачи функционального покрытия проектов большой размерности. Тестирующие программы усложняются, если у тестируемых компонентов не два, а большее число входных портов. В таких случаях для получения тестовых воздействий требуется генерация не пар, а наборов чисел. Усложняется также VHDL-код для проверки покрытия перекрёстных корзин, особенно, если приходится учитывать разрешённые и запрещённые диапазоны.

Таблица 1. Нумерация перекрёстных корзин в массиве *CovCrossAB*

b \ a	0	1	2	…	14	15
0	0	1	2	…	14	15
1	16	17	18	…	30	31
2	32	33	34	…	46	47
…	…	…	…	…	…	…
14	224	225	226	…	238	239
15	240	241	242	…	254	255

Таким образом, применение традиционных средств VHDL для генерации случайных чисел и функционального покрытия не позволяет унифицировать написание тестирующих программ, т.к. для каждого проекта необходимо определять и использовать новые типы данных и писать сложный код, что требует большого опыта программирования на VHDL. Поэтому целесообразно использовать уже разработанные средства, предлагаемые методологией OS-VVM.

Написание программы с использованием пакетов CoveragePkg, RandomPkg

Методология OS-VVM базируется на использовании средств VHDL-пакетов *RandomPkg*, *CoveragePkg*, предназначенных для эффективного решения следующих задач функциональной верификации:

- генерация псевдослучайных чисел, подчиняющихся различным законам распределения вероятностей их появления, например, нормальному закону с заданными параметрами;
- генерация псевдослучайных чисел, представленных различными типами данных – *real*, *integer*, *unsigned*, *signed*, *std_logic_vector*;
- исключение некоторых значений (из определённых диапазонов) при генерации псевдослучайных чисел;
- организация перебора всех значений каждого из генерируемых чисел;
- организация перебора всех пар (троек и т.д.) значений генерируемых чисел.

Рассмотрим программу (см. листинг 2), которая написана с использованием средств пакетов *RandomPkg*, *CoveragePkg*. Эта программа является более компактной, позволяет решать те же задачи тестирования, что и программа из листинга 1, и выдаёт те же результаты моделирования.

В листинге 2 жирным шрифтом выделены строки, позволяющие осуществлять генерацию псевдослучайных чисел *a*, *b* средствами пакета *RandomPkg*. Для каждого генератора чисел *a*, *b* в строке 13 декларируется своя переменная *RndA*, *RndB* защищённого типа *RandomPType*. Затем требуется задать начальные значения для каждого генератора, чтобы они выдавали разные последовательности псевдослучайных чисел. Для этого в строках 28, 29 вызывается метод *InitSeed* с такими же начальными значениями аргументов, как в листинге 1, после чего можно использовать генераторы. В строках 31, 32 вызывается метод *RandSlv*(0, 15, 4), который возвращает 4-разрядный вектор типа *std_logic_vector*, псевдослучайное значение которого лежит в диапазоне [0, 15]. Таким образом, использование защищённого типа *RandomPType* и его методов существенно упрощает написание кода и работу с генератором псевдослучайных чисел по сравнению со стандартным подходом (листинг 1), т.к. не требуются дополнительные переменные *SeedA1*, *SeedA2*, *SeedB1*, *SeedB2*.

Теперь рассмотрим средства пакета *CoveragePkg*, позволяющие проводить функциональное покрытие. Основным объектом при покрытии являются корзины. В пакете декларируется защищённый тип *CovBinBaseType* и множество методов, позволяющих выполнять различные операции над корзинами. Как уже говорилось выше, под корзиной понимается «место», где хранится число попаданий значений переменной в определённый диапазон при выполнении моделирования. Строго говоря, в пакете *CoveragePkg* для корзины определён тип данных *CovBinBaseType*, а именно, тип *record* (запись). Поля этой записи задают:

- *BinVal* – массив, включающий минимальное и максимальное значения интервала собираемых корзиной значений;
- *Count* – текущее (при моделировании) число попаданий в корзину значений из заданного диапазона;
- *AtLeast* – цель покрытия задаёт число попаданий, при котором корзина будет считаться покрытой;
- *Weight* – вес – это целое (не равное нулю) положительное число, по которому вычисляется вероятность выбора корзины при использовании метода *RandCovPoint*;
- *Action* – действие (корзина может быть запрещённой, игнорируемой или рабочей); для рабочей корзины считается число попаданий.

Для сбора покрытия значений чисел *a*, *b* и пар <*a*, *b*> в листинге 2 используются общие переменные *CovA*, *CovB*, *CovCrossAB* защищённого типа *CovP-*

Type, которые декларируются в строке 14. В строках 24–26 для каждой переменной создаются корзины с помощью функции *GenBin*(0, 15), которые посредством вызова метода *AddBins* добавляются в соответствующую переменную. Параметры 0, 15 функции *GenBin* задают интервал собираемых значений. Всего создаётся 16 корзин – для каждого значения числа (от 0 до 15). По умолчанию, значение поля *AtLeast* принимается равным 1. Это означает, что в результате моделирования хотя бы одно значение должно попасть в каждую корзину, чтобы точка покрытия считалась проверенной (покрытой). Для создания перекрёстных корзин в строке 26 используется метод *AddCross*. Следует отметить, что методы *AddBins*, *AddCross* и функция *GenBin* являются перегруженными и позволяют создавать корзины с различными значениями параметров корзин (*BinVal*, *AtLeast*, *Weight*, *Action*).

В строках 34–36 листинга 2 для сбора значений чисел используется перегруженный метод *ICover*, который увеличивает текущее число попаданий *Count* значения числа в соответствующей корзине. Если, например, при моделировании число *a* = 5 повторится 17 раз, то число попаданий в корзину, собирающей числа 5, будет равно *Count* = 17. Заметим, что в листинге 1 элементы массива целых чисел *CovA* эквивалентны полю *Count* в записи *CovBinBaseType*.

В строках 39–41 с помощью метода *IsCovered* определяются флаги *CovACovered*, *CovBCovered*, *CovCrossABCovered*, которые, как и в предыдущем примере, обозначают, покрыты или нет соответствующие точки покрытия. Метод *IsCovered* возвращает значение *true*, если достигнуто требуемое покрытие всех корзин. Все корзины считаются покрытыми, если для каждой из них число попавших в нее чисел (*Count*) не менее значения цели (*AtLeast*). По умолчанию, для корзин целью является 1, т.е. считается, что корзина заполнена, если в неё попало хотя бы одно число.

В строке 43 с помощью метода *WriteBin* выводятся на консоль отчёты о заполнении корзин соответствующих точек покрытия. Метод *WriteBin* является перегруженным и может выводить отчёт в файл, что можно использовать для статистической обработки данных. Также есть методы *WriteCovDb* и *ReadCovDb*, позволяющие сохранять в файл и читать из файла базу данных точки покрытия. Это даёт возможность собирать суммарное покрытие по разным сеансам моделирования. Фрагмент отчёта о заполнении точки покрытия *CovA* для программы листинга 2, выдаваемый методом *WriteBin*, приведён ниже:

Листинг 2. Использование пакетов *RandomPkg*, *CoveragePkg*

```
1 library ieee; use ieee.std_logic_1164.all;
2 use ieee.numeric_std.all;
3 use work.RandomPkg.all; use work.CoveragePkg.all;
4 entity testbench is
5 end;
6 architecture tb2 of testbench is
7  component mult
8   port (a, b : in std_logic_vector (4 downto 1);
9      d  : out std_logic_vector (8 downto 1));
10 end component;
11 signal a, b : std_logic_vector (4 downto 1);
12 signal d  : std_logic_vector (8 downto 1);
13 shared variable RndA, RndB : RandomPType;
14 shared variable CovA, CovB, CovCrossAB : CovPType;
15 begin
16 DUV : mult port map (a => a, b => b, d => d);
17 RandomGenProc1 : process
18  variable i, a_i, b_i    : natural := 0;
19  variable CovACovered    : boolean := false;
20  variable CovBCovered    : boolean := false;
21  variable CovCrossABCovered : boolean := false;
22 begin
23  -- создание корзин для a, b, (a, b)
24  CovA.AddBins(GenBin(0, 15));
25  CovB.AddBins(GenBin(0, 15));
26  CovCrossAB.AddCross(GenBin(0, 15), GenBin(0, 15));
27  -- инициализация начальных значений генераторов
28  RndA.InitSeed(IV => (7, 1));
29  RndB.InitSeed(IV => (4, 2));
30  MainCovLoop : while not CovCrossABCovered loop
31   a <= RndA.RandSlv(0, 15, 4);
32   b <= RndB.RandSlv(0, 15, 4);
33   wait for 10 ns;
34   a_i := to_integer(unsigned(a)); CovA.ICover(a_i);
35   b_i := to_integer(unsigned(b)); CovB.ICover(b_i);
36   CovCrossAB.ICover((a_i, b_i));
37   i := i+1;
38   if i = 1000000 then exit; end if;
39   CovACovered := CovA.IsCovered;
40   CovBCovered := CovB.IsCovered;
41   CovCrossABCovered := CovCrossAB.IsCovered;
42  end loop;
43  CovA.WriteBin; CovB.WriteBin; CovCrossAB.WriteBin;
44  wait for 10 ns; wait;
45 end process;
46 end architecture tb2;
```

```
WriteBin:
# %% Bin:(0)  Count = 96
# %% Bin:(1)  Count = 110
. . . пропущены строки
# %% Bin:(14)  Count = 95
# %% Bin:(15)  Count = 92
```

В пакете *RandomPkg* для псевдослучайного генератора используется стандартная функция *ieee.math_real.uniform* и в листингах 1 и 2 используются одни и те же начальные значения генераторов, поэтому результаты моделирования обеих программ одинаковы, т.е. покрытие точек *CovA*, *CovB* и *CovCrossAB* достигается за 103, 68 и 1615 циклов генерации чисел соответственно. Недостатком программ в листингах 1 и 2 является избыточность циклов генерации чисел, т.е. для покрытия 256 корзин достаточно 256 циклов

Листинг 3. Программа, выполняющая интеллектуальное покрытие

```
1 library ieee; use ieee.std_logic_1164.all;
2 use ieee.numeric_std.all; use work.CoveragePkg.all;
3 entity testbench is
4 end;
5 architecture tb3 of testbench is
6  component mult
7   port (a, b : in std_logic_vector (4 downto 1);
8      d  : out std_logic_vector (8 downto 1));
9  end component;
10 signal a, b : std_logic_vector (4 downto 1);
11 signal d  : std_logic_vector (8 downto 1);
12 shared variable CovA, CovB, CovCrossAB : CovPType;
13 begin
14 DUV : mult port map (a => a, b => b, d => d);
15 RandomGenProc1 : process
16  variable RndA, RndB    : integer;
17  variable i, a_i, b_i    : natural := 0;
18  variable CovACovered    : boolean := false;
19  variable CovBCovered    : boolean := false;
20  variable CovCrossABCovered : boolean := false;
21 begin
22  -- инициализация генератора псевдослучайных чисел
23  CovCrossAB.InitSeed(CovCrossAB'instance_name);
24  -- создание корзин для a, b, (a, b)
25  CovA.AddBins(GenBin(0, 15));
26  CovB.AddBins(GenBin(0, 15));
27  CovCrossAB.AddCross(GenBin(0, 15), GenBin(0, 15));
28  MainCovLoop : while not CovCrossABCovered loop
29   (RndA, RndB) := CovCrossAB.RandCovPoint;
30   a <= std_logic_vector(to_unsigned(RndA, 4));
31   b <= std_logic_vector(to_unsigned(RndB, 4));
32   wait for 10 ns;
33   a_i := to_integer(unsigned(a)); CovA.ICover(a_i);
34   b_i := to_integer(unsigned(b)); CovB.ICover(b_i);
35   CovCrossAB.ICover((a_i, b_i));
36   i := i+1;
37   CovACovered := CovA.IsCovered;
38   CovBCovered := CovB.IsCovered;
39   CovCrossABCovered := CovCrossAB.IsCovered;
40  end loop;
41  CovA.WriteBin; CovB.WriteBin; CovCrossAB.WriteBin;
42  wait for 10 ns; wait;
43 end process;
44 end architecture tb3;
```

(при последовательном переборе), а в приведённых примерах циклов требуется в 6,3 раза больше. Например, пара $a = 10, b = 8$ была сгенерирована 14 раз, а пара $a = 5, b = 1$ – только один раз. В следующем примере будет показано, как решить данную задачу средствами пакета *CoveragePkg*.

Интеллектуальное покрытие

В листинге 3 приведена программа тестирования, выполняющая интеллектуальное покрытие (*intelligent coverage*) [2]. Её выполнение показывает, что для перебора всех пар случайных чисел *a*, *b* требуется ровно 256 итераций цикла генерации. Это достигается использованием метода *RandCovPoint* (строка 29), который возвращает псевдослучайно выбранное значение типа *integer_vector* (в данном примере массив из пары чисел) из псевдослучайно выбранной корзины. При этом для псевдослучайного генератора значений используются только те корзины, которые не достигли заданного процента от цели покрытия *AtLeast* (по умолчанию, 100%). Покрытие всех значений числа *a* достигается за первые 52 цикла, покрытие всех значений числа *b* – за первые 38 циклов.

Отметим, что инициализация начального значения псевдослучайного генератора в точке покрытия *CovCrossAB* выполняется с помощью метода *InitSeed* (строка 23 в листинге 3). Для этого метода аргументом является строка

```
:testbench(tb3): CovCrossAB
```

Эта строка возвращается атрибутом *'instance_name* для переменной CovCrossAB, который указывает путь в дереве иерархии проекта к этой переменной. Такой способ настройки генераторов псевдослучайных чисел позволяет задавать уникальные начальные значения для каждого генератора.

Заметим, что пакет *RandomPkg* и декларируемый в нём защищённый тип *RandomPType* в листинге 3 явно не используются. Но в защищённом типе *CovPType* пакета *CoveragePkg* имеется внутренняя переменная *RV* защищённого типа *RandomPType*, и доступ к ней осуществляется вызовом соответствующих методов.

Заключение

Приведённые примеры только иллюстрируют возможности методологии OS-VVM, не исчерпывая всех возможностей пакетов *CoveragePkg* и *RandomPkg*. Не были рассмотрены, например, способы задания требуемых распределений генерируемых псевдослучайных чисел, что может быть выполнено с помощью пакета *RandomPkg*, который поддерживает генерацию случайных чисел в различных форматах, с различными распределениями и ограничениями.

Литература

1. *Авдеев Н.А., Бибило П.Н.* Средства VHDL для функциональной верификации цифровых систем // Современная электроника. – 2013. – № 3.
2. http://osvvm.org/about-os-vvm.
3. *Хаханов В.И., Хаханова И.В., Литвинова Е.И., Гузь О.А.* Проектирование и верификация цифровых систем на кристаллах. Verilog & SystemVerilog. – Харьков : ХНУРЭ, 2010.
4. *Spear C., Tumbush G.* SystemVerilog for Verification. A Guide to Learning the Testbench Language Features, Springer, 2012.
5. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-2008.