BRINGING CONTINUOUS DOMAIN INTO SYSTEMVERILOG COVERGROUPS

Prabal K Bhattacharya
Cadence Design Systems
2655 Seely Avenue,
San Jose, USA
1-408-894-2508
prabal@cadence.com

Swapnajit Chakraborti
Cadence Design Systems
57 A&B, NSEZ,
Noida, India
+91-1203984000
swapnaj@cadence.com

Scott Little
Intel Corporation
2111 NE 25th Avenue,
Hillsboro, USA
1-503-696-8080
scott.little@intel.com

Donald O'Riordan
Cadence Design Systems
2655 Seely Avenue,
San Jose, USA
1-408-428-5794
riordan@cadence.com

Vaibhav Bhutani
Cadence Design Systems
57 A&B, NSEZ,
Noida, India
+91-1203984000
vbhutani@cadence.com

ABSTRACT

This paper proposes a set of requirements for specifying functional coverage on an analog or mixed-signal block. We explain how the real number data type can be introduced in the SystemVerilog coverpoint specification and how it can enable a complete coverage specification for a mixed-signal verification environment. In discussing the requirements, we explore the challenges in partitioning the infinite continuum of real numbers during bin creation, precisely representing and comparing real numbers, naming automatically created bins, scoring real valued covergroups and managing duplicate values within a coverpoint. We also illustrate how existing functional coverage language aspects such as scalar bins, vector bins, ignore bins, and cross coverage can be extended to real data types. Finally we show a design example where we illustrate how support for the real data type in SystemVerilog covergroups helps realize the verification goals for a complete mixed-signal system.

1. INTRODUCTION

Functional coverage is the process of determining how much functionality of the design has been exercised by the verification environment. It is a well established component of a modern day verification environment for measuring the completeness of a given verification test suite.

Two notable characteristics of functional coverage are that it is user-specified and based on the design intent. Regardless of the actual design code, the functional coverage goals and specification can be developed so long as it accurately captures the high-level design intent. The SystemVerilog functional coverage feature facilitates such user specification of design intent by providing language constructs that can conveniently express functional coverage models. When a standard SystemVerilog simulator executes such a coverage specification, it enables the user to perform coverage analysis in the same breath as design verification and paves the way to develop high quality tests in less time. Such coverage driven verification has become fairly well understood and is an established practice in the digital functional verification world.

Let us now turn our attention to the world of mixed-signal verification. A complete mixed-signal SOC verification suite usually involves a complex integration of blocks created in different languages and at different levels of abstraction. Such integration choices are influenced by both performance and accuracy concerns. The verification engineer must continue to change the representation of the key blocks in his SOC system from a digital or functional representation to a more accurate transistor level representation and vice versa. As the complexity of the analog grows, it becomes more challenging to exhaust the complete state space of the analog system(s) in the SOC comprising of operating modes, process variation, corners analysis and so forth using simple circuit simulation techniques. This leads to the thought of applying traditional digital verification techniques such as constrained random stimulus generation and functional coverage to the continuous domain. Furthermore, from a coverage model reuse point of view, it is important that coverage specifications written with a discrete event driven system in mind continue to work when the representation of that system is changed to one of continuous values.

The combination of these two requirements has led to a multitude of analyses in the EDA verification tool space relating to the extension of functional coverage to the analog or mixed-signal parts of the design. With an overwhelming majority of the verification systems being written using SystemVerilog, it becomes a requirement to provide a formal definition of *mixed-signal* functional coverage in the language. Since mixed-signal design behavior can only be effectively expressed using a floating point representation, it is mandatory that the SystemVerilog standard be extended to support functional coverage using real numbers. This paper describes such an extension.

2. CURRENT STATE OF COVERAGE POINT SUPPORT IN SYSTEMVERILOG

SystemVerilog functional coverage, as described in Chapter 19 of IEEE P1800 SystemVerilog Language Reference Manual [1], enables design and verification engineers to create functional coverage models using the covergroup construct. A typical coverage model contains coverpoints and cross declarations with user-defined or implicit bins for sampling and scoring. Primarily, two types of bins can be specified with any coverpoint declaration, namely, scalar bins and vector bins. Scalar bins score all the values/ranges specified by the bin definition within a single bin. In contrast, a vector bin declaration creates individual bins for each of the integral values specified by the bin definition. The current syntax of the covergroup construct allows only integral expressions for both coverpoints and bin values and hence users are restricted to integral domain coverage models. Besides its coverage model specification, the current language standard provides various built-in methods for runtime coverage collection and control for users. The IEEE P1800 SystemVerilog Language Reference Manual standard provides a mechanism to specify a sampling event with each covergroup declaration for triggering coverage collection. A covergroup declaration is considered to be a type declaration and hence the current standard mandates its instantiation for enabling coverage collection. In general, design and verification engineers analyze both type-based and instance-based coverage data generated by covergroup declarations and their instances during simulation. The IEEE P1800 SystemVerilog Language Reference Manual provides well-defined artifacts to generate such reports enabling post-simulation 'hole-analysis' of coverage models. Overall, the current SystemVerilog covergroup syntax addresses several requirements for enabling Coverage-Driven-Verification flows which are essential for constrained-random based simulation.

3. PROPOSAL TO INTRODUCE REAL VALUED COVERAGE POINTS IN SYSTEMVERILOG

3.1 BASIC PREMISE

A covergroup can contain one or more coverage points. As part of providing support for functional coverage in an analog or mixed-signal design block, we propose that a coverage point can be an integral ***or real*** variable or an integral ***or real*** expression. Each coverage point includes a set of bins associated with its sampled values or its value transitions. The bins can be explicitly defined by the user or automatically created by SystemVerilog.

Given that the set of real numbers represents a continuum, then in order to associate a set of bins to the real-valued coverage point, we propose the following basic requirement that such coverage points must satisfy.

A coverpoint of type real must be a range or a finite set of real numbers. In other words, the following coverpoint declaration is illegal when `var` is real:

```
covergroup g1 @(posedge clk);
  c: coverpoint var; // not legal if var is real
endgroup
```

Another way to state the above is that automatic bin declaration (either implicitly, or explicitly by specifying a value for the **auto_bin_max** coverage option) will not be allowed for any real valued coverpoint without a range or finite value specification. Along the same lines, **cross** coverage for such real valued coverpoint will also not be allowed.

```
coverpoint a; // Not legal if a is of type real
coverpoint b; // Not legal if b is of type real
cross a, b; // Not legal if a or b is of type
real
```

3.2 SCALAR BINS WITH REAL VALUED COVERPOINTS

A scalar bin with a real numbered coverpoint is legal and can contain either a singleton value or a range. In case of an absolute value specification, the bin score gets incremented when the value of the coverpoint 'exactly' matches (see section 4.1.2) with the absolute value. In case of a range specification, the bin score gets incremented when the value of the coverpoint satisfies the range expression as outlined in Section 5. The following is an example of a scalar bin with real valued coverpoints.

```
coverpoint c  {
    bins b = {0, [0.5:0.8], 1.0};
}
```

It will be an error to declare a integer valued coverpoint to have bin specification involving real numbers.

3.3 RANGE SPECIFICATION FOR REAL VALUED COVERPOINTS

The range of a bin for real valued coverpoints shall be of the following types

- Open
- Closed
- Half Open/Half Closed

A range is called Open, when it is of the form `(a:b)` where for any `r` completely contained in the range, `a < r < b`

A range is called Closed, when it is of the form `[a:b]` where for any r completely contained in the range, `a ≤ r ≤ b`

A range is called Half Open or Half Closed, when it is of the form `[a:b)` or `(a:b]` where for any `r` completely contained in the range, `a ≤ r < b` or `a < r ≤ b` respectively.

3.4 BIN CREATION AND RANGE_PRECISION

When a coverage point is declared for a real variable or expression, the size of its vector bin will be determined with the help of a new per instance covergroup option called `range_precision`. This option shall have the following properties:

- The syntax for the `range_precision` option will be of the form `option.range_precision=value` where `value` can be any positive real number. The **range_precision** option will have no default value and not specifying this option will result in an error.
- Setting the `range_precision` option on the entire covergroup will apply to *all* real valued coverpoints in the covergroup.
- For integral type coverpoints, specifying `range_precision` will have no effect.
- For scalar bins, the `range_precision` option is not needed.

The `range_precision` value will be used to divide the specified range and create automatic bins. The process of division will start from the left bound of each range and will add the range precision value to the left bound to create the right bound of the new bin. The range of the new bin thus created will be open on the right bound. This is explained with a relevant example in the next section.

3.4.1 BIN CREATION FOR VECTOR BINS

If a vector bin has range `⟨a:b⟩` and its `range_precision` is set to `r`, where `a`, `b` and `r` are SystemVerilog real numbers, the new bins will be created as follows:

`⟨a : a+r)`, `[a+r : a+2r)`, …, `[a+m*r : b⟩` where `m=n-1`, `n=number of bins created`

where ⟨ and ⟩ are left and right bounds and can be either open, i.e. "(" or ")" or closed, i.e. "[" or "]".

Consider the following example of a user-defined vector bin **b1** with no explicit size. Here **a** is one SystemVerilog real type variable.

```
coverpoint a {
    option.range_precision=0.1;
    bins b1[] = {[3.5:3.8]}
}
```

In the above case, after applying the bin creation formula as described earlier using **range_precision**, **n**=3 bins will be created as shown below:

```
b1[3.5:3.6)  → [3.5:3.6)
b1[3.6:3.7)  → [3.6:3.7)
b1[3.7:3.8]  → [3.7:3.8]
```

If the size of a vector bin is explicitly specified, the **range_precision** option will be used to divide any range expression into sub-ranges and then the possible bin values will be uniformly distributed among the specified bins. Consider the following example:

```
coverpoint a {
  option.range_precision=0.1;
  bins a1[10] = {0.0, [3.5:10.5], 127.0, 255.0};
}
```

For the above declaration, after applying the **range_precision** option, we get 73 sub ranges. Since the bin size is 10, bins a1[0] to a1[8] will get 7 values each and bin a1[9] will get 10 values. The mappings of the bins to values will look like:

```
a1[0] → { 0.0, [3.5:4.1) }
a1[1] → { [4.1:4.8) }
…
a1[9]  → { [9.7:10.5), 127.0, 255.0 }
```

The algorithm used here for distributing the values to various bins is similar to the behavior of sized vector bins for integral coverpoints as described in Section 19.5 of the P1800 SystemVerilog Language Reference Manual.

As mentioned in section 3.1, automatic vector bins are not supported with real coverpoints. It will be illegal to specify **auto_bin_max** for real valued coverpoints with an explicit user-defined bin declaration, namely scalar, vector (fixed size, un-sized) bins, etc.

Once the vector bins are created based upon the **range_precision** option, the bin definitions need to be reported to users as well. The naming of the vector bins for real type based coverpoints and related challenges are described in detail in section 4.1.1.

Although the necessity of transition bin support for real based coverpoints is appreciated, it is currently kept outside the realm of this paper. This activity may be carried out as an extension of this work.

3.4.2 END POINT DEFINITION FOR REAL VALUED COVERPOINTS

As per the current IEEE P1800 SystemVerilog Language Reference Manual, the user can specify '**$**' as an endpoint of bin ranges. This happens to be a common practice among coverage model developers during initial phases of model creation. The same syntax will be applied for real type coverpoints. The value of '**$**' – when appearing in the left bound of a range – will be **-DBL_MAX**. It will be **+DBL_MAX** when appearing in the right bound of a range. The values of **-DBL_MAX** and **+DBL_MAX** may have some hardware dependency based upon the OS and architecture. For integral coverpoints, the values of the endpoints are decided by the size of the variable or expression specified with coverpoints. In case of real based coverpoint ranges, the value of endpoints are fixed i.e. **-DBL_MAX** and **+DBL_MAX**.

The presence of '**$**' in the range expression of a vector bin may lead to an error condition when the range value exceeds the maximum value that can be represented in a floating point number system. Such a condition may arise when for a range [a:b], abs(a-b) > **DBL_MAX**. In such cases, an implementation is free to choose its desired behavior. The use of '**$**' is allowed in the case of scalar bins and will not cause any overflow related issue as in vector bins.

When a bin range inside a vector bin involving a real valued coverpoint contains '**$**', the construction of bin ranges will continue to follow the general rule laid out in section 3.4.1. For a vector bin with fixed size, the range will be divided into equal parts by the specified size. Because of the issue related to range value exceeding the maximum real value **(abs(a-b) > DBL_MAX)**, the results may not match user expectation and will be tool dependent. For a vector bin with unspecified size, the range will be divided into equal parts by the value of the **range_precision** option.

4. CHALLENGES ASSOCIATED WITH FLOATING POINT SPECIFICATION

As discussed in section 3.4.1, one of the challenges of allowing coverpoints for real data types is the visualization of the vector bin names. For integral coverpoints, the names of vector bins include the value. The nomenclature of real value based vector bins is also done in similar manner. As the representation of the real value and its formatting has significant impact on the displayed value, those two factors are very important for visualization. This is described in the following section.

4.1.1 Naming of vector bins

In case of a vector bin, the name of a bin is determined from the value of the range. Consider the following example of a user-defined vector bin where singleton real values are used:

```
bins b1[] = {2.3, 2.499};
```

For the above bin declaration, a user would expect two bins, namely, **b1[2.3]** and **b1[2.499]** to be created, sampled and reported by relevant tools. It is a known fact that current IEEE 754 standard [2] for representing floating point numbers (henceforth we have used "real" and "floating point" numbers interchangeably) does not ensure that exact value specified by the user will be stored always. The problem is that floating point numbers are represented by binary fractions and only those floating point numbers which are multiples of power of 2 can be represented exactly within a given precision. For all other numbers, approximate values are stored and thus in most cases we will find round-off errors in the representation. For example, storing the value 0.1 requires an infinite number of binary digits to represent it exactly. The point to note here is that the amount of such round-off errors will depend on number of precision digits available for representation which, again, may be architecture/OS/compiler dependent. It is possible that these round-off errors may impact the digits within the precision of interest to the user. In that case, the value specified by the user may not match exactly the value reported. Now creating the bin names of vector bins using the real values will require the formatting of that value and it is obvious that formatting precision will have a role to play regarding how the bin names are reported. The implementation may choose to use a default standard double format precision for creating the vector bin name. However, the user may also require some capability to control the precision of digits while formatting the bin names. Providing such capability will also ensure that users are aware of such issues although they may not arise frequently. In the future, it may be necessary to introduce a new covergroup level option to handle such user configurability.

Another area which is prone to round-off errors is when the range for a particular vector bin is computed by applying the **range_precision** value to the left and right bounds [section 3.4.1]. It is possible that floating point round-off errors arising as a result of arithmetic operations in such cases may impact the digits of precision within a user's interest. For example, if bin range is specified as [62.2:64] and range_precision = 0.62, the first sub-range as per user expectation will be [62.2:62.2+0.62) or [62.2:62.82). But due to round-off error involved in floating point addition 62.2+0.62, the right bound of computed sub-range will not be exactly 62.82 as expected by user. This will cause mismatch of sampling as well as of the bin names with user expectation.

Apart from the above two scenarios, another possibility is that the user specifies a floating point number which has more digits than the available precision. In that case also, round-off errors may lead to unpredictable behavior that does not match with user expectations. For example, a user specifies a range value as "2.999999999999999999999999999999999999999999999999999999". Now this will be treated as "3" and sampling and reporting will be based on 3. Although this is somewhat of a corner case, it is mentioned to clarify the potential mismatch with user expectations. In such cases, users should be warned accordingly. Similar issue exists for the following declaration as well.

```
bins b1[] =
  {0.100000000000000001:
   0.100000000000000003}
```

with the **range_precision** option set to **0.000000000000000001**

With the precision limit set to 16 bits, all the sub-ranges would now evaluate to 0.1 and therefore the bin names for the ranges would not be unique anymore.

With all the examples shown above, we raise the question if it makes sense for the user to be able to set a floating point precision such that round-off errors like the above can be avoided in some cases if not all. This idea involves the introduction of a new *global option* that would control the precision of printing a floating point value which in turn would influence how the names of bin ranges would be represented.

Users should also be sensitized towards issues related to round-off errors occurring because of floating point arithmetic operations involving real bin values. For example, if a user specifies a bin declaration as: "**bins b1[] = {0.9};**" for a real variable "**a**" and expects that when "**a**" takes a value 0.3*3 then there will be a match, this expectation will be incorrect. The reason is that 0.9 is not exactly equal to 0.3*3 due to round-off errors involved in arithmetic operation (*). This issue is discussed in detail in the next section.

4.1.2 Applying a 'fudge factor' or 'tolerance'

During simulation, the real type bin values are compared with the coverpoint values to determine a match. There is a well-known matching algorithm proposed by Knuth [3] for matching "double precision" numbers which can be used for SystemVerilog real based coverpoint matching as well. This matching algorithm works on the principle of matching two double precision numbers using a tolerance (*epsilon*) value rather than doing an exact match. SystemVerilog real value matching during sampling of covergroups will work based on this same principle. A new covergroup option will be added for enabling user configuration of such tolerance in the future. It should be noted that real value comparison based on such tolerances will help address the issue of mismatch due to round-off errors during floating point operations. Hence, the two values "0.3*3" and "0.9" will now match as expected by the user.

5. SCORING BINS FOR REAL VALUED COVERPOINTS

The process of scoring integral bins is described in 19.5 of the IEEE P1800 SystemVerilog Language Reference Manual. The score or count for a given bin is initialized to zero and incremented every time the coverage point matches one of the values contained in the set defined for a given bin. Evaluation of the coverage point expression takes place when the covergroup is sampled. The sampling process is independent of whether the coverpoint is of real or integral type. The example below illustrates the sampling of a real valued coverpoint.

```
c: coverpoint c {
  option.range_precision=0.1;
  bins b[] = {0,[3:5],8};
}
```

If the coverage event for the given covergroup generates n events at which a value of **c** is sampled, n will remain the same even if **c** is of type real. In absence of a clocking event, the behavior of the built-in sample() method will also remain the same if **c** is of type real.

Given a coverpoint of integral values, each value in the bin can be enumerated. This is not the case for real valued coverpoints. A bin for a real valued coverpoint may contain a range of real numbers which cannot be explicitly enumerated. As a result the rules for scoring a real valued coverpoint are slightly different, but are generally what the user would expect. The following example illustrates the scoring rules for real valued coverpoints.

```
c: coverpoint c {
  option.range_precision=0.1;
  bins b[] = {0,[0.5:0.8),8}
}
```

Based on the range specification and the `range_precision` option, the following bin mappings will be created:

```
b[0] -> 0.0
b[0.5:0.6)
b[0.6:0.7)
b[0.7:0.8)
b[8] -> 8.0
```

For the five bins created in this example, using the given range precision 0.1, we will examine how various sample points would be scored. A number of example sample points and their scoring are shown in the table below.

Value	Condition satisfied	Bin scored
0.001	NONE	NONE
0	c == 0	c.b[0]
0.2	NONE	NONE
0.55	$0.5 \leq c < 0.6$	c.b[0.5:0.6)
0.6	$0.6 \leq c < 0.7$	c.b[0.6:0.7)
0.79	$0.7 \leq c < 0.8$	c.b[0.7:0.8)
8.000001	NONE	NONE

The above table emphasizes the point that given a range specification, scoring a real valued number works much like an integral expression. As noted above, when a range specification is not present, a real numbered coverpoint object can generate values that would not satisfy numerical equality due to the floating point precision characteristics of real numbers. Until a tolerance-based matching scheme (as described in 4.1.2) is introduced, we propose that this aspect of real numbers must be dealt with by the user. It is therefore expected that users will normally create range specifications suitable for capturing the ranges of interest. We also expect that scalar real valued bins will be the exception as they may feel 'unreliable' to the user, behave differently across implementations, etc

5.1 Ignoring Bins

The following summarizes the current behavior of bin ignore process per the IEEE P1800 SystemVerilog Language Reference Manual (Sec. 19.5.4).

"All values or transitions associated with ignore bins are excluded from coverage. For state bins, each ignored value is removed from the set of values associated with any coverage bin. ... The removal of ignored values shall occur after the distribution of values to the specified bins. ...

The above may result in a bin that is associated with no values or sequences. Such empty bins are excluded from coverage (see 19.11)."

For vector bins with real numbered coverpoints, the process of ignoring bins shall start with the creation of bin ranges according to the `range_precision` option (unless the vector bin is explicitly sized). Once the bin ranges are created, the ranges that fall completely under the ignore_bins range shall be removed since such ranges would be rendered empty after the ignored values are removed from the state of values associated with that bin range. The ranges whose values are not completely covered within in the range of ignored values shall be retained, but inside that range the ignored values shall not be scored.

The following examples illustrate the above rule. We consider three separate coverpoint definitions below and look at how ignore bins work for real numbers when the ignored value is a specific number, a range, or a mix thereof.

```
option.range_precision = 0.1;
coverpoint a {
  bins b1[] = {[2.4:2.8]};
  ignore_bins ig = {2.5};
}
coverpoint b {
  bins b1[] = {[2.4:2.8]};
  ignore_bins ig = {[2.5:2.6]};
}
coverpoint c {
  bins b1 = {[2.4:2.8]};
  ignore_bins ig = {[2.59:2.63],2.79};
}
```

For each of the coverpoints the bins of `b1` are divided as shown below.

```
(1) b1[2.4:2.5)
(2) b1[2.5:2.6)
(3) b1[2.6:2.7)
(4) b1[2.7:2.8]
```

For coverpoint `a` , the value of 2.5 must be ignored when scoring. This is done by modifying (2) to be `b1(2.5:2.6)`. For coverpoint `b` an entire range `[2.5:2.6]` is ignored. This requires the complete removal of (2) and the modification of (3) to `b1(2.6:2.7)`. Coverpoint `c` ignores a range as well as a point. This results in changes to several bins as well as a splitting of the range in (4). The changes to (2) are `b1[2.5:2.59)` and the change to (3) is `b1(2.63:2.7)`. The changes to (4) result in the split range of `{[2.7:2.79),(2.79:2.8]}`.

5.2 Duplicate values across bins

For a vector bin, once the bin ranges are constructed (either from the explicitly specified size or from the `range_precision` option), if duplicate values or ranges of values exist across bin ranges then a point matching both bins will be scored multiple times. This example illustrates how duplicate bins are handled.

```
real a;
option.range_precision = 0.2;
coverpoint a {
  bins b1[] = {[2.4:2.8], [2.5:3.0]};
}
```

In this case the bins are:

```
b1[2.4:2.6)
b1[2.6:2.8]
b1[2.5:2.7)
b1[2.7:2.9)
b1[2.9:3.0]
```

A value of 2.65 would be therefore scored in both the bins `b1[2.6:2.8]` and `b1[2.5:2.7)`. Duplicates are also allowed for fixed size vector bins.

6. PUTTING IT ALL TOGETHER

To illustrate the utility of the real valued coverpoints, we use real valued covergroups to help quantify the quality of random test generation for a voltage detector circuit. This voltage detector produces both warning and error detection events when the input voltage crosses a trip point. The trip points for the warning and error events can be independently configured to four different values.

The coverpoints are setup to capture when the input signal is in various regions of operation. Ideally, the ranges would be half-open, but this representation allows for cases where the implementation does not yet support half-open intervals.

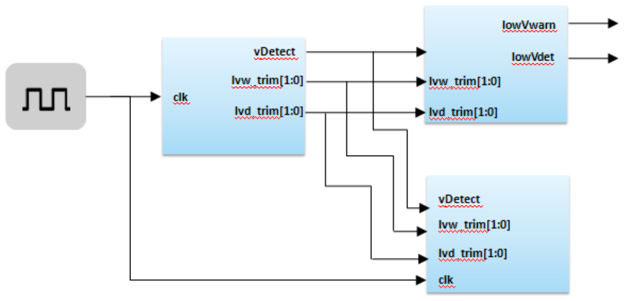


Figure 1 Block diagram of voltage detector with testbench

6.1 Top Level Testbench

The source code for the top level testbench is shown below and a block diagram of the testbench is shown in Figure 1. The testbench contains a driver, the voltage detector, and the coverage collection module. The driver generates random stimulus for the voltage detector and the coverage collection module observes and captures what is generated.

Coverpoints are created to capture values of `vDetect` based on the possible trip points. These coverpoints are then crossed with the trim values. This coverage will help us understand if the voltage is moving through the important ranges while the design is configured for the different trim settings. Functional checks would be expected to complete a proper testbench but are omitted here for brevity.

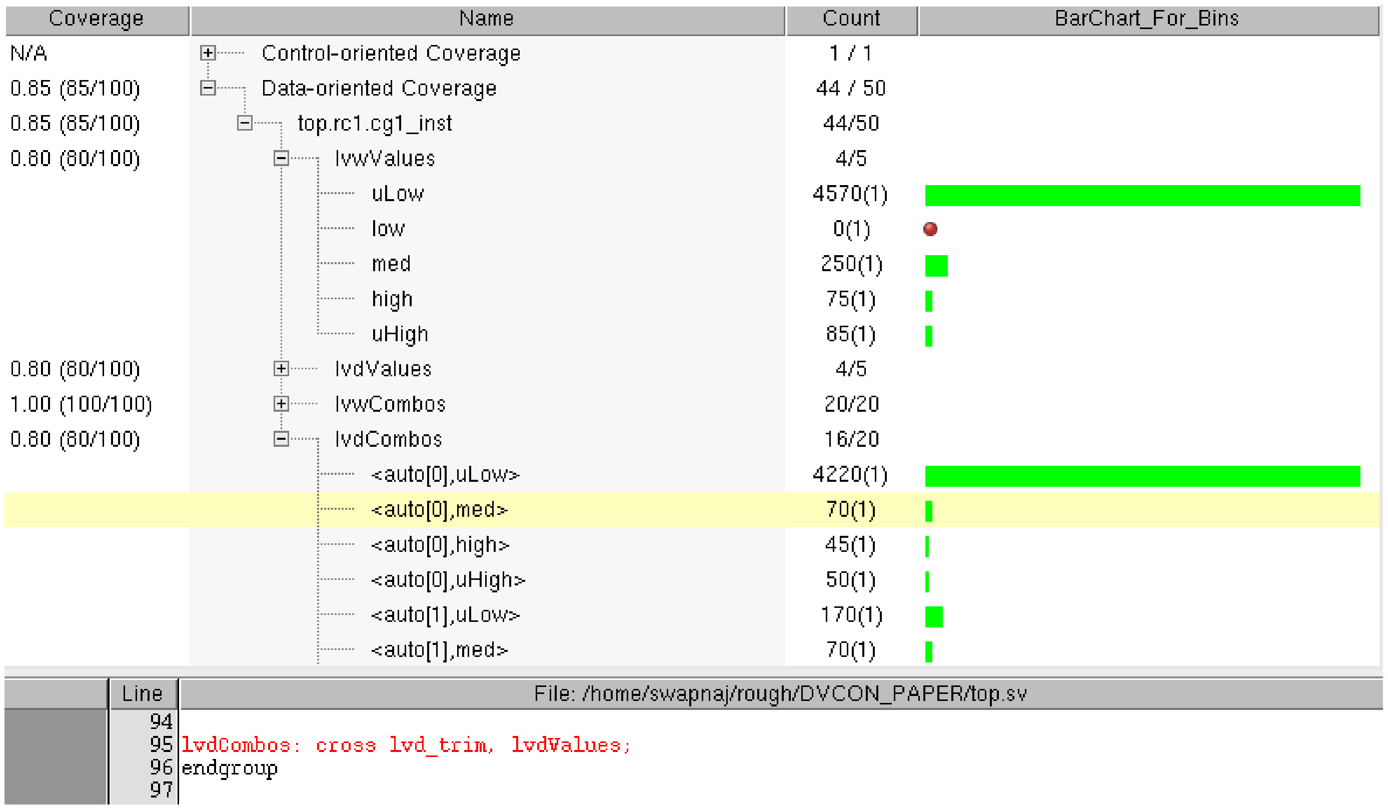
```
`timescale 1ns / 10ps

module top();

  logic clk;
  always #1 clk = ~clk;

  initial
  begin
     clk = 0;
     #10000 $finish;
  end

  wire [1:0] lvw_trim;
  wire [1:0] lvd_trim;

  tbDriver dr1(.*);
  voltageDetector voltD(.lowVwarn(lowVwarn),
                        .lowVdet (lowVdet ),
                        .vDetect (vDetect ),
                        .lvw_trim(lvw_trim),
                        .lvd_trim(lvd_trim));
  realCov rc1(.*);
endmodule

module tbDriver (
  output var real       vDetect,
  output logic [1:0] lvw_trim,
  output logic [1:0] lvd_trim,
  input logic        clk
);
  class c;
    rand real v_detect;
    rand integer lvw_trim_val;
    rand integer lvd_trim_val;
       constraint  c1  {  v_detect  inside  {
[0.0:3.3] } ; }
       constraint  c2  {  lvw_trim_val  inside  {
[0:3] } ; }
       constraint  c3  {  lvd_trim_val  inside  {
[0:3] } ; }
       constraint   c4   {   lvd_trim_val   <=
lvw_trim_val; }
       function void self_print();
         $display("v_detect=%f, lvw_trim_val=%d,
lvd_trim_val=%d\n",   v_detect,   lvw_trim_val,
lvd_trim_val);
    endfunction
  endclass : c

  c c_inst;

  initial begin
    c_inst = new();
       for(int i = 0; i < 2000; i++) begin
         assert (c_inst.randomize());
         vDetect = c_inst.v_detect;
         lvw_trim = c_inst.lvw_trim_val;
         lvd_trim = c_inst.lvd_trim_val;
         #10 c_inst.self_print();
    end
  end

endmodule

module realCov (
  input logic       clk,
  input real        vDetect,
  input logic [1:0] lvw_trim,
  input logic [1:0] lvd_trim
);

covergroup lvdLvwCombos @(posedge clk);

option.per_instance = 1;

//Capture the various ranges for LVW
lvwValues: coverpoint vDetect {
  bins uLow  = {[$:1.97]};
  bins low   = {[1.98:1.99]};
  bins med   = {[2.0:2.69]};
  bins high  = {[2.7:2.99]};
  bins uHigh = {[3.0:$]};
}

//Capture the various ranges for LVD
lvdValues: coverpoint vDetect {
  bins uLow  = {[$:1.87]};
  bins low   = {[1.88:1.89]};
  bins med   = {[1.9:2.59]};
  bins high  = {[2.6:2.89]};
  bins uHigh = {[2.9:$]};
}

lvwCombos: cross lvw_trim, lvwValues;

lvdCombos: cross lvd_trim, lvdValues;
endgroup

lvdLvwCombos cg1 ;

initial
begin
   cg1 = new;
   cg1.set_inst_name("cg1_inst");
end

endmodule
```

6.2 Voltage Detector Module

```
`include "constants.vams"
`include "disciplines.vams"
`timescale 1ns / 10ps

module voltageDetector (
  lowVwarn,
  lowVdet,
  vDetect,
  lvw_trim,
  lvd_trim
);

  output      lowVwarn, lowVdet;
  logic       lowVwarn, lowVdet;
  reg         lowVwarn, lowVdet;

  input       vDetect;
  input [1:0] lvw_trim;
  input [1:0] lvd_trim;

  wreal       vDetect;
  logic [1:0] lvw_trim;
  logic [1:0] lvd_trim;

  real lvw_val, lvd_val;

  initial begin
    lowVwarn <= 1'b0;
    lowVdet  <= 1'b1;
  end

  always @(lvw_trim) begin
    case(lvw_trim)
      2'b00 : lvw_val = 1.98;
      2'b01 : lvw_val = 2.0;
      2'b10 : lvw_val = 2.7;
      2'b11 : lvw_val = 3.0;
    endcase
  end

  always @(lvd_trim) begin
    case(lvd_trim)
      2'b00 : lvd_val = 1.88;
      2'b01 : lvd_val = 1.9;
      2'b10 : lvd_val = 2.6;
      2'b11 : lvd_val = 2.9;
    endcase
  end

  always @(vDetect < lvw_val)
    lowVwarn <= 1'b1;

  always @(vDetect < lvd_val) begin
    $display("vDetect  inside  voltagedetector  =
%g\n", vDetect);
    lowVdet  <= 1'b1;
  end

endmodule
```

6.3 Coverage Results

A simulation is run and coverage data is collected using the Cadence Incisive Enterprise Simulator [4]. We would expect one hundred percent coverage for such a simple coverage model. However, there are two bins (see Figure 2) that do not show any coverage. Upon on closer analysis we discover that these ranges are very narrow and are more difficult to hit. If we increase the number of iterations we eventually hit these missing points although it is likely easier to write a small number of directed tests to hit these narrow bins.



Coverage	Name	Count	BarChart_For_Bins
N/A	Control-oriented Coverage	1 / 1	
0.85 (85/100)	Data-oriented Coverage	44 / 50	
0.85 (85/100)	top.rc1.cg1_inst	44/50	
0.80 (80/100)	lvwValues	4/5	
	uLow	4570(1)	🟩
	low	0(1)	●
	med	250(1)	🟩
	high	75(1)	🟩
	uHigh	85(1)	🟩
0.80 (80/100)	lvdValues	4/5	
1.00 (100/100)	lvwCombos	20/20	
0.80 (80/100)	lvdCombos	16/20	
	<auto[0],uLow>	4220(1)	🟩
	<auto[0],med>	70(1)	🟩
	<auto[0],high>	45(1)	🟩
	<auto[0],uHigh>	50(1)	🟩
	<auto[1],uLow>	170(1)	🟩
	<auto[1],med>	70(1)	🟩

Line File: /home/swapnaj/rough/DVCON_PAPER/top.sv

```

lvdCombos: cross lvd_trim, lvdValues;
endgroup

```

Figure 2 Coverage Chart showing some Empty Bins

7. REFERENCES

[1] IEEE P1800 SystemVerilog Language Reference Manual 2009

[2] IEEE 754 Standard Floating Point Numbers

[3] The Art of Computer Programming, volume 2: Seminumerical Algorithms, Donald E. Knuth, 3rd edition, 1998.

[4] Cadence Incisive Enterprise Simulator, http://www.cadence.com/products/fv/enterprise_simulator