COVERGATE: Coverage Exposed

Rich Edelman
Mentor, A Siemens Business
Fremont, CA

***Abstract*- In the hardware verification world a common and popular technique for "checking" is to use coverage. Yet, as common and popular as it is, it is still the realm of the specialist or the Verification IP. This paper will explore and simplify coverage through examples and use models. It will explore functional coverage, line coverage, expression coverage among others. It will explore coverage debug and coverage distribution.**

The examples offer a guide for simple, easy to use coverage models.

I. Introduction

Coverage is a tool common in software design. The concept is easy to think about. If you have not "covered" something then you haven't tested it. For example, a line of software that has never been executed is not covered, and is not tested.

```
10  int *p;
11  int condition1 = 0;
12  if (condition1)
13    value = *p;
```

In the code above, the condition in the 'if' is never true. That means that the assignment statement on line 13 has never been tested. A code coverage tool would generate a report that line 13 is untested. In its simplest form, this is code coverage.

In hardware verification these same coverage concepts apply. But now there are many more things to be covered – not just line execution, but also, expression values, branches, toggles, and functional coverage.

Some coverage is "built-in" to the language. This allows tools to calculate coverage automatically. No intervention is needed. For example, the hardware simulator can know which statements are executable, and can keep track of whether the statement has been executed or not as simulation progresses. At the end of simulation a report of coverage can be produced.

Other kinds of coverage – for example, functional coverage in SystemVerilog is not automatic. Functional coverage is intentional – a verification engineer creates a coverage model and "samples" the coverage and creates a report. For example, a functional coverage model could be created that "covers" the ReadWrite functionality of a bus. The state of the bus must be READ, WRITE and IDLE. If it is each of those values, then we consider it 100% covered. A slightly different coverage model could be created that requires the same ReadWrite state to transition from IDLE->READ and IDLE->WRITE, but never READ->WRITE directly, nor WRITE->READ directly.

This is a simple coverage collector for this RW bus.

```
class coverage_collector extends uvm_subscriber # (transaction);
  `uvm_component_utils (coverage_collector)

  transaction t;

  covergroup cg;
    rw_cp:    coverpoint t.rw;
    addr_cp:  coverpoint t.addr {
      bins regular [] = {[0:99]};
      ignore_bins ignore_addr = {[100:$]};
    }
    data0_cp:  coverpoint t.data [7:0] {
      bins regular [] = {[0:99]};
      ignore_bins ignore_data = {[100:$]};
    }
    data1_cp:  coverpoint t.data [15:8] {
      bins regular [] = {[0:99]};
      ignore_bins ignore_data = {[100:$]};
    }
    data2_cp:  coverpoint t.data [23:16] {
      bins regular [] = {[0:99]};
      ignore_bins ignore_data = {[100:$]};
    }
    data3_cp:  coverpoint t.data [31:24] {
      bins regular [] = {[0:99]};
      ignore_bins ignore_data = {[100:$]};
    }

     addr_X_rw: cross addr_cp, rw_cp;
    data0_X_rw: cross data0_cp, rw_cp;
  endgroup

  virtual function void sample (transaction t);
    this.t = t;
    cg.sample ();
  endfunction

  function new (string name = "coverage_collector",
      uvm_component parent = null);
    super.new (name, parent);
    cg = new ();
  endfunction

  function void write (transaction t);
    sample (t);
    `uvm_info ("COVERAGE", $sformatf ("Coverage=%0d%% (t=%s)",
      cg.get_inst_coverage (), t.convert2string ()), UVM_MEDIUM)
  endfunction
endclass
```

Figure 1 Coverage Collector

This is a simple coverage collector for transitions on the RW signal.

```
class transition_coverage_collector extends uvm_subscriber # (transaction);
  `uvm_component_utils (transition_coverage_collector)

  transaction t;

  covergroup cg;
    rw_cp: coverpoint t.rw {
             bins r_w = (READ  => WRITE);
             bins w_r = (WRITE => READ);

             bins r_i = (READ  => IDLE);
             bins w_i = (WRITE => IDLE);

             bins i_r = (IDLE  => READ);
             bins i_w = (IDLE  => WRITE);
    }
  endgroup

  virtual function void sample (transaction t);
    this.t = t;
    cg.sample ();
  endfunction

  function new (string name = "transition_coverage_collector",
      uvm_component parent = null);
    super.new (name, parent);
    cg = new ();
  endfunction

  function void write (transaction t);
    sample (t);
    `uvm_info ("COVERAGE", $sformatf("Coverage=%0d%% (t=%s)",
      cg.get_inst_coverage (), t.convert2string()), UVM_MEDIUM)
  endfunction
endclass
```
Figure 2 Transition Coverage Collector

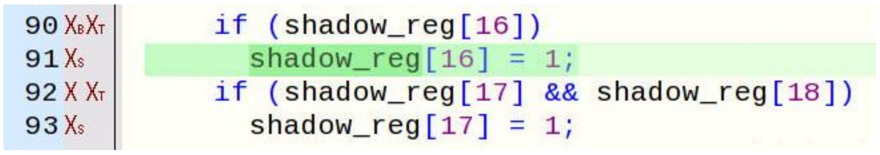
The examples above offer a template for a coverage collector. Create a class with a sample routine. Construct the covergroup in the class constructor. Use the UVM analysis port connections with a monitor.

II. AUTOMATICALLY GENERATED COVERAGE

A. *Statement Coverage*

Statement coverage is simple and easy to understand. A statement is considered covered if it has executed. A statement that is not covered and therefore didn't execute is potentially a source of bugs, since the behavior hasn't been tested.

Statements on lines 91 and 93 are not covered.



```
90 XBXT      if (shadow_reg[16])
91 XS          shadow_reg[16] = 1;
92 X XT      if (shadow_reg[17] && shadow_reg[18])
93 XS          shadow_reg[17] = 1;
```

In this case the 'if' statement executed, but the Boolean value of the condition was never true.

B. *Branch Coverage*

Branch coverage is also easy to understand. Did a branch get taken? In the example above, line 90 was never true, so no branch was taken. Line 92 failed branch coverage, because the expression was never true.

C. *Condition Coverage*

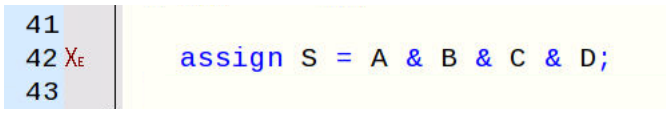
Condition coverage is similar or an extension to branch coverage. But it is a little more complicated. Condition coverage considers an expression. In order for a condition to be covered, each of the inputs to the expression must be covered. For an input to be covered, it must be able to control the output of the expression and the output must go to both 0 and 1 states. For example

```
A & B
```

If the value of B is always zero, then A will not be covered, since it cannot control the output.

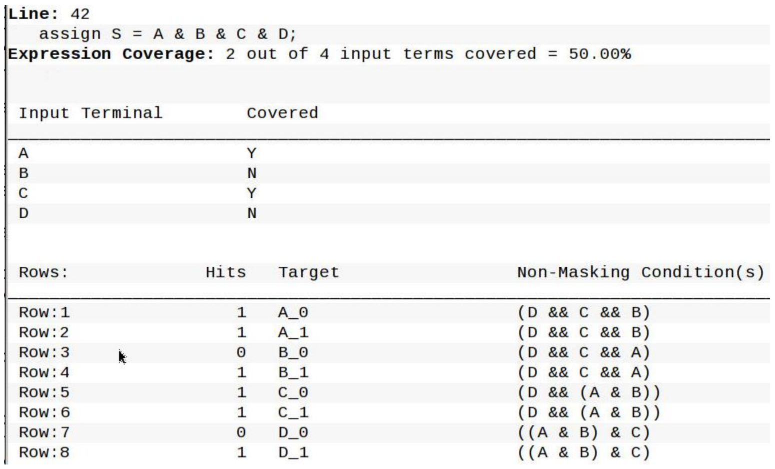
D. *Expression Coverage*

Expression coverage is similar to condition coverage. Expression coverage effectively creates a truth table and counts the numbers of times a row matches the values of the expression inputs.



```
41
42 XE    assign S = A & B & C & D;
43
```

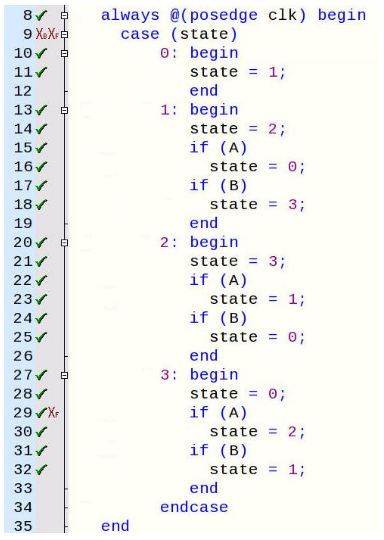
The expression above is a simple one – ANDed signals. In order for one of the inputs to control the output, all the other inputs must be one. Rows 3 and 4 describe how B=0 never occurred when A, C and D were one and how B=1 occurred one time when A, C and D were one. Therefore, B is NOT covered. Similarly for input D. Inputs A and C both could control the output and had the values zero and one. This expression is 50% covered.



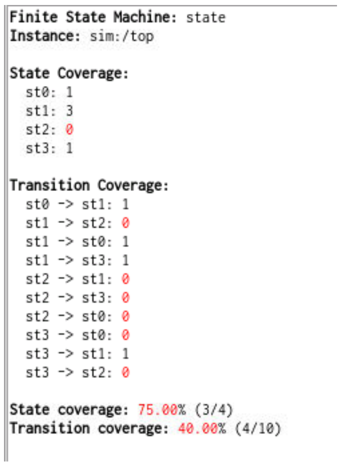
```
Line: 42
   assign S = A & B & C & D;
Expression Coverage: 2 out of 4 input terms covered = 50.00%

 Input Terminal        Covered
------------------------------------------------------------------------
 A                     Y
 B                     N
 C                     Y
 D                     N

 Rows:             Hits   Target                  Non-Masking Condition(s)
------------------------------------------------------------------------
 Row:1                1   A_0                     (D && C && B)
 Row:2                1   A_1                     (D && C && B)
 Row:3                0   B_0                     (D && C && A)
 Row:4                1   B_1                     (D && C && A)
 Row:5                1   C_0                     (D && (A & B))
 Row:6                1   C_1                     (D && (A & B))
 Row:7                0   D_0                     ((A & B) & C)
 Row:8                1   D_1                     ((A & B) & C)
```

E. Finite State Machine Coverage

Finite state machine coverage makes sure that the state variable takes all legal values and that each state transition is taken.



```
always @(posedge clk) begin
  case (state)
      0: begin
         state = 1;
         end
      1: begin
         state = 2;
         if (A)
           state = 0;
         if (B)
           state = 3;
         end
      2: begin
         state = 3;
         if (A)
           state = 1;
         if (B)
           state = 0;
         end
      3: begin
         state = 0;
         if (A)
           state = 2;
         if (B)
           state = 1;
         end
      endcase
end
```



```
Finite State Machine: state
Instance: sim:/top

State Coverage:
  st0: 1
  st1: 3
  st2: 0
  st3: 1

Transition Coverage:
  st0 -> st1: 1
  st1 -> st2: 0
  st1 -> st0: 1
  st1 -> st3: 1
  st2 -> st1: 0
  st2 -> st3: 0
  st2 -> st0: 0
  st3 -> st0: 0
  st3 -> st1: 1
  st3 -> st2: 0

State coverage: 75.00% (3/4)
Transition coverage: 40.00% (4/10)
```

F. Toggle Coverage

Toggle coverage is perhaps the easiest coverage to understand. Did the signal toggle? Did a bit change to a zero and change to a one?

III. SystemVerilog Functional Coverage

Collecting functional coverage is very different from the preceding discussion about statement, condition, branch, expression and toggle coverage. Those coverage metrics can be automatically created by a tool. They are well defined and easy to generate.

Functional coverage, on the other hand, is just as it sounds – it is coverage of some function. The functionality that is being covered is up to the verification team. For example, coverage could be collected on the ‘address’ range of memory access, but not on the data. Furthermore, the address range functional coverage could be binned for certain ranges.

```
241   covergroup cg;
242     rw_cp:     coverpoint t.rw;
243     addr_cp:   coverpoint t.addr {
244       bins regular [] = {[0:99]};
245       ignore_bins ignore_addr = {[100:$]};
246     }
247     data0_cp:   coverpoint t.data [7:0] {
248       bins regular [] = {[0:99]};
249       ignore_bins ignore_data = {[100:$]};
250     }
251     data1_cp:   coverpoint t.data [15:8] {
252       bins regular [] = {[0:99]};
253       ignore_bins ignore_data = {[100:$]};
254     }
255     data2_cp:   coverpoint t.data [23:16] {
256       bins regular [] = {[0:99]};
257       ignore_bins ignore_data = {[100:$]};
258     }
259     data3_cp:   coverpoint t.data [31:24] {
260       bins regular [] = {[0:99]};
261       ignore_bins ignore_data = {[100:$]};
262     }
263
264      addr_X_rw: cross addr_cp, rw_cp;
265     data0_X_rw: cross data0_cp, rw_cp;
266   endgroup
```

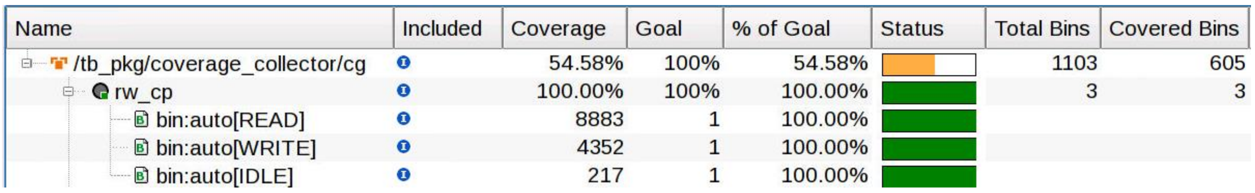
A. *Covergroups*

A covergroup is created to check some functional coverage. The covergroup has a name, in this case ‘cg’. The covergroup contains coverpoints and crosses.

B. *Coverpoints and Coverbins*

A coverpoint comes in two styles. The first style as shown in line 242 above does not define any “bins”. The bins will be created automatically by the tool.

Bins are used as fancy counters. A bin represents a possible value. For the enumeration t.rw (READ, WRITE, IDLE), there are three possible values. Line 242 lets the tool automatically generate a bin for each value. Each time the value is sampled, the bin-counter will be incremented.



Name	Included	Coverage	Goal	% of Goal	Status	Total Bins	Covered Bins
/tb_pkg/coverage_collector/cg	●	54.58%	100%	54.58%	🟧	1103	605
rw_cp	●	100.00%	100%	100.00%	🟩	3	3
bin:auto[READ]	●	8883	1	100.00%	🟩		
bin:auto[WRITE]	●	4352	1	100.00%	🟩		
bin:auto[IDLE]	●	217	1	100.00%	🟩		

The second style of coverpoint defines its own bins. The coverpoint on line 243, covering the address, creates an array of bins named ‘regular[0], regular[1], … regular[99]’.

```
covergroup cg;

  addr_cp:  coverpoint t.addr {
    bins regular [] = {[0:99]};
    ignore_bins ignore_addr = {[100:$]};
  }
```

The syntax above defines 100 bins which will count values from zero to 99. Other values fall into the ‘ignore_addr’ bin, defined as values of 100 or greater. The ignore_addr bin is an ‘ignore_bins’ type – this means that the counts are not considered for functional coverage calculations.

In the covergroup below, a 32 bit data value is covered, one byte at a time. There are 4 coverpoints, each with two bins like the address coverage – an array of bins for values from zero to 99, and a bin for values 100 and greater. So each byte will cover 100 values.

```
covergroup cg;

  data0_cp:  coverpoint t.data [7:0] {
    bins regular [] = {[0:99]};
    ignore_bins ignore_data = {[100:$]};
  }
  data1_cp:  coverpoint t.data [15:8] {
    bins regular [] = {[0:99]};
    ignore_bins ignore_data = {[100:$]};
  }
  data2_cp:  coverpoint t.data [23:16] {
    bins regular [] = {[0:99]};
    ignore_bins ignore_data = {[100:$]};
  }
  data3_cp:  coverpoint t.data [31:24] {
    bins regular [] = {[0:99]};
    ignore_bins ignore_data = {[100:$]};
  }

   addr_X_rw: cross addr_cp, rw_cp;
  data0_X_rw: cross data0_cp, rw_cp;
endgroup
```

C. The sample() function

Covergroups collect coverage by sampling the values of the things they are covering. In the example above, the fields of the ‘transaction t’, namely the t.rw field, the t.addr field and the t.data field.

```
class coverage_collector extends uvm_subscriber # (transaction);
  `uvm_component_utils (coverage_collector)

  transaction t;

  covergroup cg;
    rw_cp:     coverpoint t.rw;
    addr_cp:   coverpoint t.addr {
    data0_cp:  coverpoint t.data [7:0]
    data1_cp:  coverpoint t.data [15:8]
    data2_cp:  coverpoint t.data [23:16]
    data3_cp:  coverpoint t.data [31:24]
  endgroup


  virtual function void sample (transaction t);
    this.t = t;
    cg.sample ();
  endfunction

  function new (string name = "coverage_collector",
      uvm_component parent = null);
```

```
275     super.new (name, parent);
276     cg = new ();
277   endfunction
278
279   function void write (transaction t);
280     sample (t);
281     `uvm_info ("COVERAGE", $sformatf ("Coverage=%0d%% (t=%s)",
282       cg.get_inst_coverage (), t.convert2string ()), UVM_MEDIUM)
283   endfunction
284 endclass
```

The covergroup is defined in a class. It is constructed in the class constructor (line 276). The class above is a “UVM subscriber” and is connected to a monitor in the UVM. Please see the entire example for details.

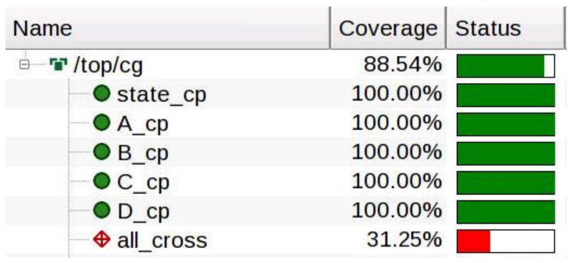
The coverage collector class defines two functions, sample() and write(). When sample() is called, it copies the transaction handle to a class member variable, and then is calls cg.sample(). This is a simple way to have the covergroup be sampled. The write() routine is called from a monitor as ap.write(t), which causes the monitored transaction to be sent to the analysis port write routine, eventually calling the write() routine on line 279. This write routine is how the coverage collector gets the transaction that is to be covered. The write() routine is simple – it calls sample(t) and prints a message. The message is the accumulated coverage of this covergroup.

```
# UVM_INFO tb.sv(281) @ 717190: .a2.cc [COVERAGE] Coverage=55% (t=[] rw=READ, addr=499, data=500)
# UVM_INFO tb.sv(281) @ 717190: .a2.cc [COVERAGE] Coverage=55% (t=[] rw=READ, addr=499, data=500)
# UVM_INFO tb.sv(281) @ 717410: .a2.cc [COVERAGE] Coverage=55% (t=[] rw=READ, addr=699, data=700)
```

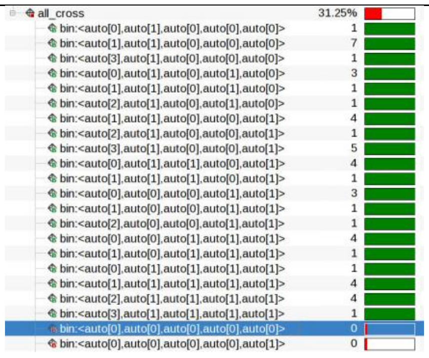
D. Cross

Covergroups also contain crosses. A cross is a bit-wise Cartesian product of two or more coverpoints. For the simple example below, there are 5 variables being sampled – a two bit vector and 4 single bit vectors. This makes 6 bits, or 64 bins. The cross of these 5 variables has 64 bins. A report is generated and it is clear that the condition of all zeroes never occurred.

```
    reg [1:0] state;
    reg A, B, C, D;

    covergroup cg;
      state_cp: coverpoint state;
      A_cp: coverpoint A;
      B_cp: coverpoint B;
      C_cp: coverpoint C;
      D_cp: coverpoint D;

      all_cross: cross state_cp, A_cp, B_cp, C_cp, D_cp;
    endgroup
```



Name	Coverage	Status
/top/cg	88.54%	🟩
state_cp	100.00%	🟩
A_cp	100.00%	🟩
B_cp	100.00%	🟩
C_cp	100.00%	🟩
D_cp	100.00%	🟩
all_cross	31.25%	🟥



all_cross	31.25%	🟥
bin:<auto[0],auto[1],auto[0],auto[0],auto[0]>	1	🟩
bin:<auto[1],auto[1],auto[0],auto[0],auto[0]>	7	🟩
bin:<auto[3],auto[1],auto[0],auto[0],auto[0]>	1	🟩
bin:<auto[0],auto[1],auto[0],auto[1],auto[0]>	3	🟩
bin:<auto[1],auto[1],auto[0],auto[1],auto[0]>	1	🟩
bin:<auto[2],auto[1],auto[0],auto[1],auto[0]>	1	🟩
bin:<auto[1],auto[1],auto[0],auto[0],auto[1]>	4	🟩
bin:<auto[2],auto[1],auto[0],auto[0],auto[1]>	1	🟩
bin:<auto[3],auto[1],auto[0],auto[0],auto[1]>	5	🟩
bin:<auto[0],auto[1],auto[1],auto[0],auto[1]>	4	🟩
bin:<auto[1],auto[1],auto[1],auto[0],auto[1]>	1	🟩
bin:<auto[0],auto[0],auto[0],auto[1],auto[1]>	3	🟩
bin:<auto[1],auto[0],auto[0],auto[1],auto[1]>	1	🟩
bin:<auto[2],auto[0],auto[0],auto[1],auto[1]>	1	🟩
bin:<auto[0],auto[0],auto[1],auto[1],auto[1]>	4	🟩
bin:<auto[1],auto[0],auto[1],auto[1],auto[1]>	1	🟩
bin:<auto[0],auto[1],auto[1],auto[1],auto[1]>	1	🟩
bin:<auto[1],auto[1],auto[1],auto[1],auto[1]>	4	🟩
bin:<auto[2],auto[1],auto[1],auto[1],auto[1]>	4	🟩
bin:<auto[3],auto[1],auto[1],auto[1],auto[1]>	1	🟩
bin:<auto[0],auto[0],auto[0],auto[0],auto[0]>	0	🟥
bin:<auto[0],auto[0],auto[0],auto[0],auto[1]>	0	🟥

The metrics that crosses generate are very valuable, since they count the number of times the bin values were as specified in the cross. They check that all combinations specified actually happen. For example crossing the RW with the address, checks that each address is both READ and WRITTEN. The size of a cross can grow very large if special care is not taken. Crosses have other controls to limit the size, which are beyond the scope of this paper. Please refer to the SystemVerilog LRM. There are many examples.

IV. VERIFICATION IP – HELP IS ON THE WAY

Verification IP is a widely used lever to improve verification productivity in many dimensions. For this paper, the dimension of interest is coverage. Modern verification IP includes coverage models. As the simulation runs, the built-in functional coverage models collect coverage. Each verification model is different, but a useful buying decision for verification IP is the completeness of the coverage model. Ideally the coverage model ensures that each mode and important combinations of modes is covered and is therefore tested. Obtaining a 100% coverage solution is not always possible, but using verification IP allows time to be spent writing tests and constraints to try to achieve that goal, instead of creating coverage models for each protocol in use.

A commercial AXI4 verification IP has a coverage model of more than 5000 lines of SystemVerilog code, including 91 coverpoints, 51 crosses and 2000 lines of supporting expressions, tasks and functions. Writing protocol coverage is not an easy job, and verification IP should be a serious consideration.

V. COVERAGE DEBUG

Debugging coverage problems is not an easy task, since the usual problem with coverage is “something is not covered”. In the debug world, most debug is focused on “why did something happen”. Debugging why something did NOT happen is quite hard. For coverage, most debug is – “Why is my coverage so low?”

A. Built-in Coverage is too low – something is NOT covered

Built-in coverage is the catch all for coverage that is tool generated – not functional coverage. If this coverage is wrong – it means that the hardware is not functioning as expected. This could be due to many reasons.

When a statement in a block is not covered, that means it didn’t execute. If it didn’t execute, then for some reason that block of code wasn’t executed – like an if-then-else that had the wrong value for a condition. Or a clock edge that never triggered, or a task call than never happened.

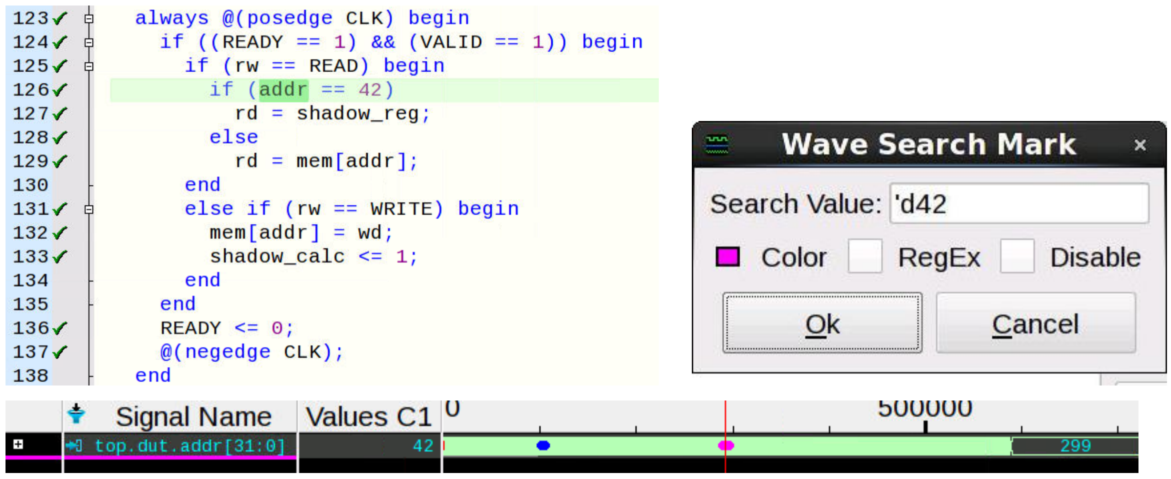
An expression that is not covered means that the signal values don’t allow each input bit to control the output. Same for conditions. Branches are similar to both and to statement coverage. FSM coverage is still similar. The state wasn’t entered or the transition didn’t happen.

These not covered situations can all be debugged using the normal debug techniques. An if-then-else with the wrong expression needs to be debugged by figuring out what values the expression did take and why. Putting the expression in a wave debugger, and finding the drivers for the value is a good place to start. The same can be said for the other types. Something did NOT happen that was expected. This kind of debug can be difficult, but usually it boils down to finding out what DID happen, and why the desired state DID NOT happen.

B. Built-in Coverage is too high – something is covered unexpectedly

Using coverage implies that ‘things’ are being counted or recorded as they happen. A signal toggles, and it is counted. But coverage can also be expected to be ZERO. For example, a section of code may only be executed during exception handling. It is known that the exception will not happen in the test, yet there is coverage on that exception handling code – those lines did execute. This is the case where something may have happened that was unexpected. Usually this is the easy kind of debug. An if-then-else condition might have become true. Finding who caused the value to be true is normally quite easy using driver tracing.

In the example below, the statement at 127 is covered, and that is unexpected. One possible debug is to put the ‘addr’ signal in the wave window and find out when it changes to 42. Then it is an easy matter to find the active driver.



```
123✓   always @(posedge CLK) begin
124✓     if ((READY == 1) && (VALID == 1)) begin
125✓       if (rw == READ) begin
126✓         if (addr == 42)
127✓           rd = shadow_reg;
128✓         else
129✓           rd = mem[addr];
130        end
131✓       else if (rw == WRITE) begin
132✓         mem[addr] = wd;
133✓         shadow_calc <= 1;
134        end
135      end
136✓     READY <= 0;
137✓     @(negedge CLK);
138    end
```

```
Wave Search Mark
Search Value: 'd42
Color   RegEx   Disable
Ok   Cancel
```

```
Signal Name          Values C1   0            500000
top.dut.addr[31:0]   42                              299
```

C. Functional Coverage

Functional coverage is slightly different to debug than built-in coverage. Functional coverage has a trigger. The trigger can be a call to sample(), or can be some event. Coverage is collected when sample() is called, or when the event triggers.

i. Missing Functional Coverage

In the case of missing functional coverage, first check that the sample() got called or the event was triggered. In the code snippets in this paper, sample is used exclusively, since it is easier to control. Additionally putting a breakpoint in the sample routine of the container class makes debugging sample calls easy.

If sample is not getting called, then the debug continues backward to figure out why. Go to all the call sites and check the logic. Normally sample is called when a new “object” is received by a scoreboard or other analysis component. If sample is not called, then coverage will not be collected.

If it is determined that sample is being called, then the next step is to check what data is being sampled. It may be that the wrong transaction handle is being used, or that the transaction handle is being updated or used by other parts of the testbench. (The object wasn’t copied or new’ed properly, so that a later transaction populated a stored handle). It pays to not reuse handles. Create a new handle for each recognized object. This also eliminates the need to the do_copy method.

The coverage class has a sample routine below, and points at its argument (this.t = t). Then it calls the covergroup sample. The coverage class sample was called by the write() routine, which was called as a part of the UVM analysis ports – very easy to debug.

```
    virtual function void sample(transaction t);
      this.t = t;
      cg.sample();
    endfunction

    function void write(transaction t);
      sample(t);
      `uvm_info("COVERAGE", $sformatf("Coverage=%0d%% (t=%s)",
        cg.get_inst_coverage(), t.convert2string()), UVM_MEDIUM)
    endfunction
```

In the monitor code below, a long running process checks the interface clock. When VALID && READY, then a transaction is recognized, constructed and member fields assigned. Then this object handle is passed to any analysis components using the UVM analysis port system – "ap.write()".

```
    task run_phase(uvm_phase phase);
      forever begin
        @(posedge mif.CLK);
        if ((mif.VALID == 1) && (mif.READY == 1)) begin
          t = transaction::type_id::create("t");
          if (mif.rw == READ) begin
            t.rw = mif.rw;
            t.addr = mif.addr;
            @(negedge mif.READY);
            t.data = mif.rd;
          end
          else if (mif.rw == WRITE) begin
            t.rw = mif.rw;
            t.addr = mif.addr;
            t.data = mif.wd;
          end
          else if (mif.rw == IDLE) begin
            t.rw = mif.rw;
            t.addr = 'z;
            t.data = 'z;
          end
          `uvm_info(get_type_name(), $sformatf("Got %s", t.convert2string()), UVM_MEDIUM)
          ap.write(t);
        end
      end
    endtask
```

ii. Too Much Functional Coverage

In the case of too much functional coverage, the problem is reversed. Instead of sample or the trigger not happening, they are happening too much. Debug is essentially the same. Figure out why the sample routine is called too many times or at the wrong time. Simply put a breakpoint in the sample routine and climb the stack to understand why it got called after the breakpoint is hit.

iii. Bad Functional Coverage

Bad functional coverage means that the covergroup, coverpoint or cross was written in a way that did the wrong thing. The most common mistake is that a cross is created that is very large. For example, crossing two 32 bit variables are 2**32 times 2**32 bins. Understanding what crosses have been created is the first step.

Coverpoints can get complicated with expressions and the iff conditional. The best advice is to minimize the number and size of the expressions and make the iff conditional simple.

A common mistake with coverpoints is how the bins are assigned values. Keeping it simple is always advised.

`bins regular = { [0:99]};` makes one bin

`bins regular[10] = {[0:99]);` makes 10 bins

`bins regular[] = { 0:99]};` makes 100 bins

```
covergroup cg;
    rw_cp:    coverpoint t.rw;
    addr_cp:  coverpoint t.addr {
      bins regular[] = { [0:99] };
      ignore_bins  ignore_addr = { [100:$] };
    }
...
```

The bin creation algorithm and syntax are worth studying, but keeping it simple makes things much easier to debug and reason about.

VI. Random Number Distribution

Coverage is normally used to understand which parts of a specification are tested and which are not. Getting good coverage and coverage distribution is important – it means the state space is sampled or tested consistently.

Sometimes the randomization used to generate stimulus is incorrect, or generating undesirable distributions of values. For example a constraint to generate numbers between 0 and 100, with the average being 50, could simply generate pairs of HIGH and LOW values (0, 100). This would result in valid random data, but a very poor distribution.

Many tools are available to analyze distributions including Excel spreadsheets and other charting tools. Below are two quick and easy alternatives: A covergroup for distribution and a simple ASCII binning/charting class.

A. A Covergroup Solution

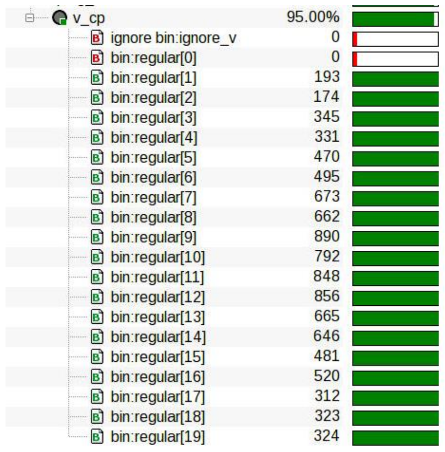
Given the class, constraints and randomization below, a covergoup could be constructed to cover the values. The distribution is specified in the constraint, and the covergroup will try to sample it. The class to be randomized below appears simple, but there are three constraints. If they were each in derived classes in different files, understanding them taken together starts to get harder. Even this simple set of three constraints can be hard to debug. Showing a distribution of values can help understand any constraint writing issues.

```
class R;
  rand int i;
  constraint c_legal_value {
    i > 0;
    i < 100;
  }
  constraint c_by_5 {
    (i%5) == 0;
  }
  constraint c_dist {
    i dist {
       [ 0: 10] := 2,
       [11: 20] := 4,
       [21: 30] := 6,
       [31: 40] := 8,
       [41: 50] := 10,
       [51: 60] := 10,
        [61: 70] := 8,
        [71: 80] := 6,
        [81: 90] := 4,
        [91:100] := 2
      };
    };
  endclass
```

The possible values of ‘i’ are 1 to 99. Further, the values must be divisible by 5. A covergroup as below will generate a simple to view report.

```
    covergroup cg;
      v_cp: coverpoint v {
        bins regular [20] = {[0:99]};
        ignore_bins ignore_v = {[100:$]};
      }
    endgroup
```

The red ‘20’ specifies that there will be 20 bins. Each of the values 0 to 99 will be distributed in the bins. This is a case of clever use of covergroup syntax. The cleverness is limited – and so acceptable. The generated report is readable.



```
v_cp                     95.00%
  ignore bin:ignore_v    0
  bin:regular[0]         0
  bin:regular[1]         193
  bin:regular[2]         174
  bin:regular[3]         345
  bin:regular[4]         331
  bin:regular[5]         470
  bin:regular[6]         495
  bin:regular[7]         673
  bin:regular[8]         662
  bin:regular[9]         890
  bin:regular[10]        792
  bin:regular[11]        848
  bin:regular[12]        856
  bin:regular[13]        665
  bin:regular[14]        646
  bin:regular[15]        481
  bin:regular[16]        520
  bin:regular[17]        312
  bin:regular[18]        323
  bin:regular[19]        324
```

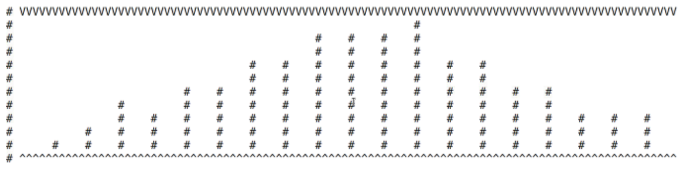
B. A Simple ASCII Graph Solution

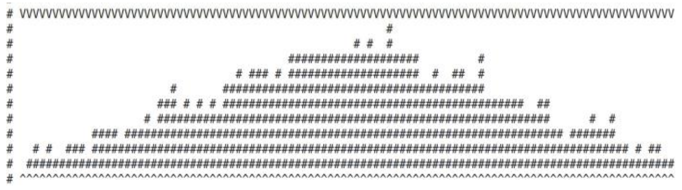
Simply graphing the distribution can improve confidence that things are going right. The complete definition of the ‘DIST’ class below draws a simple ASCII graph. It stores 100 bins of integer values. As a value is found, it is put into the distribution package by calling ‘add(VALUE)’. When a print-out is desired, simply call print(). An ASCII chart is produced. It is rather rough, but gets the job done with little effort. And is quite readable.

```
  class DIST;
    int my_bins[100];

    function void add(int value);
      my_bins[value]++;
    endfunction

    function void print();
      int largest = 0;
      int per_y;
      for (int x = 0; x < 100; x++)
        if (my_bins[x] > largest)
          largest = my_bins[x];
      per_y = largest / 10;
  for (int y = 10; y >= 1; y--) begin
   for (int x = 0; x < 100; x++)
    if (my_bins[x] >= y*per_y)
     $write("#");
    else
     $write(" ");
   $write("\n");
  end

 endfunction
endclass
```

The module below is a simple loop calling randomize 10,000 times, causing the distribution of class R to be tested. At the end of the loop, the distribution is printed as on the top right.

```
module top();
 R    r_h = new();
 DIST d_h = new();

 initial begin
  repeat (10000) begin
   if(!r_h.randomize())
    $fatal(2, "Randomize FAILED");
   d_h.add(r_h.i);
  end
  d_h.print();
 end
endmodule
```





In the TOP graph, it is quite easy to see that the distribution is as designed, with "divisible-by-5" values clearly shown. In the BOTTOM graph, the "divisible_by_5" constraint was removed.

VII. Practical Details

Using coverage is easy, but there are many possible commands and options. The usage in this paper is the simplest. Compile the code. Optimize the code. Turn on coverage instrumentation using the +cover switch. Enable coverage calculation and database creation using the –coverage switch to simulation.

```
vlog ...
vopt +cover -o opt top -debug +designfile
vsim -c opt -coverage -do "coverage save -onexit coverage.ucdb; run -a" -qwavedb=+signal
```

To debug and view reports, just load the UCDB file

```
visualizer design.bin qwave.db -ucdbfile coverage.ucdb
```

VIII. Summary

Using automatically generated coverage, functional coverage and verification IP coverage is a sure way to improve verification productivity and design quality. Different teams, different design styles and different technologies may determine different needs for coverage – perhaps just statement coverage and functional coverage on all the bus interfaces, but there are many good choices, and all verification environments should use some form of coverage and some stage of the design and verification process.

The SystemVerilog LRM chapter on Functional Coverage is about 38 pages of quite dense writing and details. There are many useful examples, features and functionality described there that are not mentioned here. They are the harder to use parts or the more complicated parts. Using the techniques outlined here with the simplest possible approach will yield coverage reports that are easy to write, easy to debug and predictable.

All source code is available from the author.

IX. ACKNOWLEDGEMENTS

The author owes a large gratitude of debt to many people who have suggested ideas and solutions, including Dirk, Dave, Jason and Scott.

X. REFERENCES

[1] SystemVerilog LRM, http://ieeexplore.ieee.org/document/8299595/
[2] SystemVerilog Functional Coverage, Akiva Michelson, Ace Verification, User2User 2007
[3] "What the Heck is FEC?", Ray Salemi, EE Journal, Sept 27, 2011, https://www.eejournal.com/article/20110927-fec/
[4] UVM 1.1d, https://www.accellera.org/images/downloads/standards/uvm/uvm-1.1d.tar.gz