A Practical Look @ SystemVerilog Coverage – Tips, Tricks, and Gotchas

Doug Smith
Doulos
16165 Monterey Road, Suite 109
Morgan Hill, CA USA
+1-888-GO DOULOS
doug.smith@doulos.com

John Aynsley
Doulos
Church Hatch, 22 Market Place
Ringwood, Hampshire UK
+44 1425 471223
john.aynsley@doulos.com

ABSTRACT

Functional verification of today's large and complex designs is a major challenge and bottleneck. As a result, various tools, techniques, and languages have been developed to automate as much as possible to maximize productivity. For example, automatic testbench generation of random stimulus offers a significant aid in finding obscure and hard-to-find bugs. With random testing, however, there is often no obvious relationship between a given simulation run and the desired test activity unless a functional coverage relationship is defined.

Once this relationship is defined, the number of times each scenario occurs is recorded as functional coverage, providing a quantitative metric of what has been tested on a device. In SystemVerilog, functional coverage is defined in terms of cover properties and functional covergroups. A rich set of language constructs is provided for defining functional scenarios and the crossing or intersection of those scenarios. SystemVerilog also offers a coverage API for accessing coverage results at simulation runtime.

Unfortunately, not all coverage-related language features are ideal or even straightforward. For instance, a rather useful feature omitted in the IEEE-1800 standard is the ability to query coverage results from specific coverage bins. Nonetheless, with a little ingenuity these shortcomings can be worked-around, which this paper describes. Tips and tricks are presented like how to direct stimulus generation using coverage results, or how to coordinate cover properties with covergroups to take advantage of the cover property's temporal syntax when matching functional behavior. Likewise, several gotchas to avoid are considered. Armed with the appropriate toolset, SystemVerilog coverage can provide an effective tool for accomplishing coverage driven verification.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids – hardware description languages, simulation, and verification.

General Terms

Languages, Verification.

Keywords

Coverage, SystemVerilog, SVA, SystemVerilog Assertions, covergroup, coverpoint, cover property, default, bins, wildcard, cross.

1. INTRODUCTION

Functional coverage comes in two flavors in SystemVerilog. One type of coverage comes from a cover property, which uses the same temporal syntax used by SystemVerilog assertions (SVA). Since cover properties uses the same properties as asserts, the same work creating the properties can be reused in both checking and coverage gathering. Cover properties are typically used for protocol coverage since the temporal syntax is ideal for describing sequences of events over time as needed for bus interfaces. Unfortunately, cover properties can only be placed in structural code (i.e., modules, programs, or interfaces) and cannot be used in class-based objects. Likewise, their coverage information is not easily accessible in SystemVerilog for use in a testbench (for example, for steering stimulus generation).

For example, Figure 1 shows a sample `cover property`. The simulator keeps track of how many times the sequence occurs and it can be viewed it in the simulation waveform or coverage report.

```
cover property ( @(posedge clk)
  $rose(req) |=> ((req && ack)[*0:$] ##1 !req) );
```

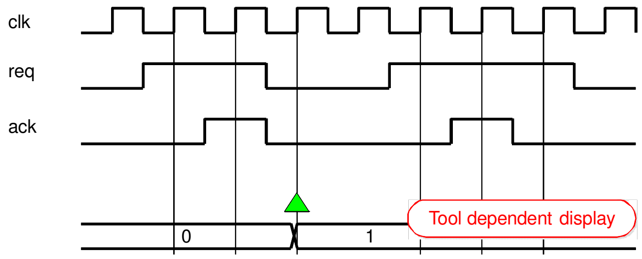
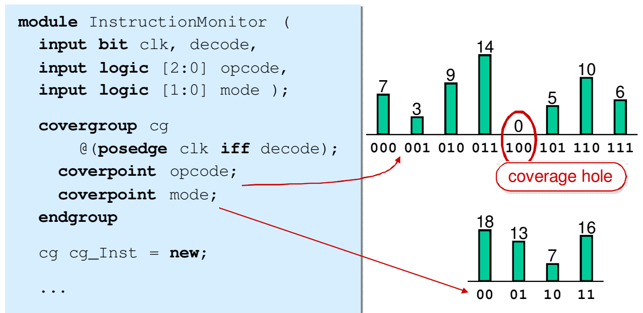


Figure 1: Property coverage using a cover property.

The second type of functional coverage is sample-based coverage provided by a covergroup. Covergroups record the number of occurrences of various values specified as coverpoints. These coverpoints can be hierarchically referenced by your testcase or testbench so that you can query whether certain values or scenarios have occurred. They also provide a means for creating cross coverage. Unlike cover properties, covergroups may be used in both class-based objects or structural code.

For example, Figure 2 illustrates a covergroup. When defining covergroups, the covergroup is given a name (e.g., `cg`) and optionally provide a sampling event, which in this case is the positive edge of `clk` qualified by the `decode` signal. In other words, when a valid instruction occurs (`decode` asserted), the values on the `opcode` and `mode` signals are sampled.



```
module InstructionMonitor (
  input bit clk, decode,
  input logic [2:0] opcode,
  input logic [1:0] mode );

  covergroup cg
      @(posedge clk iff decode);
    coverpoint opcode;
    coverpoint mode;
  endgroup

  cg cg_Inst = new;

  ...

endmodule: InstructionMonitor
```

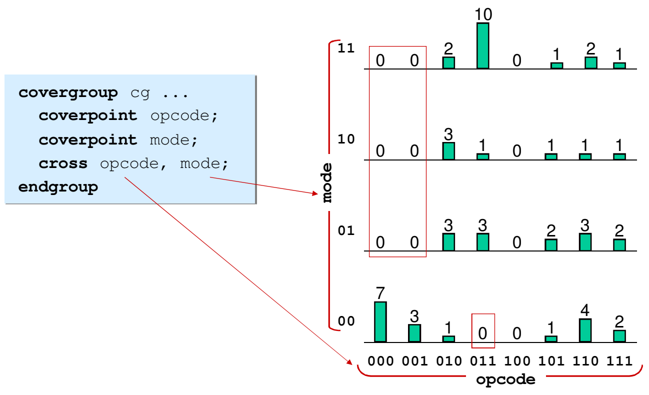
Figure 2: Sample based coverage using a covergroup.

Since `opcode` has $2^3 = 8$ possible values, 8 bins are created to keep track of the number of times each value occurs. For the `mode` input, there are $2^2 = 4$ possible values so 4 bins will be created.

Defining the covergroup alone does not start the coverage collection. A covergroup needs to be instantiated using the `new` operator and given an instance name. Inside a class, an instance name is not required and the `new` operator is called on the covergroup inside the class constructor.

Where coverage becomes really interesting is when the individual coverpoints or bins are crossed together. Crossing terms simply creates a matrix that shows when the different values cross or simultaneously occur together.

For example, in Figure 3 all opcodes values are being crossed with all possible values of mode. In other words, the matrix shows all the simultaneous occurrences of the different opcodes in all 4 modes. The zeros in the matrix reveal coverage holes; i.e., values that have either not been testing, generated, or possibly values that are invalid or undefined. In any verification effort, coverage holes must be identified and either filled by writing more stimulus, or justifiably ignored in cases such as when the scenarios are unreachable by design.



```
covergroup cg ...
  coverpoint opcode;
  coverpoint mode;
  cross opcode, mode;
endgroup
```

Figure 3: Cross-coverage example.

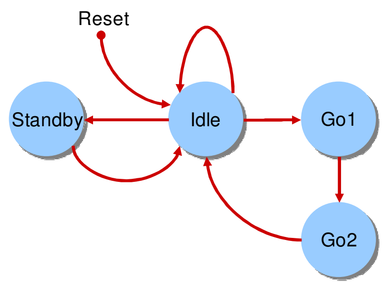
2. COVERAGE TIPS

SystemVerilog offers a wide range of options and syntax for defining coverage in any environment. For a full and detailed description of what is supported, refer to the IEEE 1800 language reference manual ([2],[3]).

Writing coverage in SystemVerilog is easy to do; however, there are a few tips that might help make your coverage even easier and more productive. Here are few worth considering.

2.1 *Tip #1*: Use of shorthand notation

SystemVerilog defines many concise ways of defining coverage. Figure 4 shows an example of a state machine and several shorthand notations available. Transitional coverage can be defined using the => operator, which keeps a record of the transitions from one state to the next.



```
enum { Idle, Standby, Go1, Go2 } states;
covergroup cg_FSM @(posedge Clock);
  coverpoint State {
    bins valid_states[] =
                  { Idle, Standby, Go1, Go2 };
    bins valid_trans =
                ( Idle => Go1 => Go2 => Idle ),
                ( Idle => Standby => Idle );
    bins reset_trans =
                ( Go1, Go2, Standby => Idle );
    bins idle_range = ( Idle[*5:7] => Go1 );
    bins go1_repeat = ( Go1 [-> 5] );
    wildcard bins idle_trans =( 2'bx1 => Idle );
  }
endgroup
```

Figure 4: FSM transition coverage.

Notice the syntax for the `reset_trans` bin: (Go1, Go2, Standby => Idle). This is really saying, *record all of the transitions to the reset state* or:

```
Go1 => Idle, Go2 => Idle, Standby => Idle
```

Explicitly writing all the transitions is not required. Likewise, the `idle_range` bin uses the sequence repeat operator [* *range*], which translates into one of the following sequences:

```
( Idle => Idle => Idle => Idle => Idle )

( Idle => Idle => Idle => Idle => Idle => Idle )

( Idle => Idle => Idle => Idle => Idle => Idle
=> Idle )
```

SystemVerilog also defines a non-consecutive operator, [->N:M], used in coverage bin `go1_repeat`. Here, coverage will be collected everytime there are 5 non-consecutive occurrences of the Go1 state.

Not only can ranges be defined in covergroups, including using the open range operator `$`, which specifies either the minimum or maximum value for a coverpoint, but the `wildcard bins` operator helps to easily define a range of values. With `wildcard`, any X, Z, or ? will be treated as a wildcard. In the `idle_trans` bin example above, the expression

```
( 2'bx1 => Idle )
```

translates into

```
Standby => Idle      // Standby = 2'b01

Go2 => Idle          // Go2 = 2'b11
```

Using these shorthand notations makes coverage much easier to write.

2.2 *Tip #2*: Covergroup arguments add flexibility

Covergroups can be defined with arguments using the same syntax as functions and tasks. Doing so makes a covergroup more flexible and reusable. For example,

```
covergroup cg (ref int v, input string comment);
  coverpoint v;

  option.per_instance = 1;
  option.weight = 5;
  option.goal = 90;
  option.comment = comment;
endgroup

int a, b;

cg cg_inst1 = new(a, "cg_inst1 - variable a");
cg cg_inst2 = new(b, "cg_inst2 - variable b");
```

Here, two arguments have been added to the covergroup `cg`. An argument `v` is added so a signal or variable to cover can be passed into the covergroup and used with the coverpoint. Notice, the argument is passed by reference using the `ref` keyword so the covergroup can see the variable's value as it changes. Likewise, other arguments like strings can be passed to be used in the covergroup's options.

Once a covergroup has arguments, multiple instances can be created where the variable to cover is pass into the covergroup during the call to `new`, creating specific covergroups for each variable. This allows reuse of the same covergroup definition.

Arguments on covergroups also facilitate reuse in another way. Many times, coverage needs collected on values outside the module where the covergroup is define such as probing down inside the design hierarchy from a testbench. SystemVerilog allows for coverpoints to use hierarchical path names with coverpoints as follows:

```
covergroup cg;
  coverpoint testbench.covunit.a;
  coverpoint $root.test.count;

  // ILLEGAL - reference to coverpoint
  // coverpoint testbench.covunit.cg_inst.cp_a;
endgroup
```

However, hard-coding pathnames inside of anything makes it less reusable. Instead, covergroup arguments can be used to provide a way to pass references without hard-coding the references into the covergroup. For example, the following shows how to make a generic covergroup and then pass the specific pathnames into it (provided the reference points to an equivalent type as the covergroup argument):

```
covergroup cg (ref logic [7:0] a, ref int b);
  coverpoint a;
  coverpoint b;
endgroup

cg cg_inst = new(testbench.covunit.a,
                 $root.test.count);
```

2.3 *Tip #3*: Utilize coverage options

Covergroups have many options that can be customized. For every covergroup, there are *type* options and *per instance* options, represented by a corresponding internal structure.

Type options apply to the entire covergroup type and can only be set when the covergroup is declared or by using the scope resolution operator (::). They are specified using the `type_option` covergroup member. There are 4 type options available:

Table 1: Covergroup type options.

Type Option	Description
weight	Weight of coverage in the coverage calculation
goal	Percentage of coverage to reach
strobe	Samples the coverage values once everything is stable (i.e., postponed simulation region)
comment	String comment

The *weight* and *goal* options are probably the most noteworthy. To remove a covergroup from the coverage calculation, simply set the covergroup's weight to 0 and it will no longer effect the result. The goal is important because it determines whether the coverage report shows a complete coverage (green) or still missing values (amber or red). The following example shows how to use type options.

```
covergroup cg @(posedge clk);
  type_option.weight = 5;    // % of calculation
  type_option.goal = 90;     // % of goal
  type_option.strobe = 1;    // Postponed region
  cp_a: coverpoint a {
      type_option.comment = comment;
  };
  coverpoint b;
endgroup

// Requires constant expressions
cg::type_option.goal = 100;
cg::cp_a::type_option.weight = 80;
```

One of the nice things about coverage in general is that it is cumulative. For example, if a covergroup is created in a class object, every instance of that covergroup adds to the overall cumulative coverage even if the class objects later becomes garage collected along with the covergroup.

However, often *per instance* coverage is required as well. Take for example, a system-on-chip. Many sub-systems will connect to the system interconnect along with the memory and the processor. Each device connects to the system interconnect using the same module that implements the bus protocol, and this module can contain a covergroup for the bus protocol. While cumulative coverage is collected across all the occurrences of the covergroup, the coverage result will not reflect which interconnects are performing the communication nor the type of communication (e.g., burst reads, simple writes, etc.). For coverage results per instance, the *per instance* option can be used. *Per instance* coverage keeps track of coverage for each interface instance so it can be seen if all the bus interfaces have been adequately exercised and tested.

To enable *per instance* coverage, the `per_instance` option is used with the `option` structure inside a covergroup. The following illustrates several of the *per instance* options available:

```
covergroup cg @(posedge clk);
  option.per_instance = 1;   // Turns on options
  option.weight = 5;         // % of calculation
  option.goal = 90;          // % of at_least
  option.at_least = 10;      // Number to see
  option.comment = comment;

  coverpoint a { option.auto_bin_max = 128; };
  coverpoint b { option.weight = 50; };
endgroup
```

There are 9 different options, but of particular importance is the `at_least` option. The `at_least` option specifies the number of times a value must occur in order to reach the coverage goal. For example, say an engineer would like to see at least 10 occurrences of a coverpoint value. If only 8 occurrences are seen, then the coverage for that coverpoint will only be 80%. Of course, the goal of any verification effort is to reach 100% coverage so setting the `goal`, `weight`, and `at_least` options are important in achieving any verification goal.

3. COVERAGE TRICKS

Defining coverage is typically straightforward in SystemVerilog, but there are some limitations and shortcomings in the language. However, that does not mean that these shortcomings cannot be worked around. In this section, several tricks will be presented to hopefully help you get the most out of SystemVerilog coverage and workaround some of its irksome limitations.

3.1 *Trick #1*: Combine cover properties with covergroups

Cover properties and covergroups essentially have different uses. A cover property looks for a match of a temporal sequence or event while a covergroup creates bins of the different values as they occur. However, sometimes it is very useful to use the powerful SVA temporal property syntax to observe behavior and cross that coverage with other values or events that occur. For example, a cover property can easily describe a read or write transaction across a bus interface, and it would be useful to cross all read and write transactions to different address spaces.

Likewise, in a class-based testbench environment where transactions are randomly generated, it might be useful to feedback the cover property coverage to help steer the stimulus generation, or at least use the cover property temporal syntax in a monitor to match when read and write transactions occur across an interface so they can be sent off to a scoreboard instead of implementing a bus protocol state machine in the monitor.

Unfortunately, cover properties cannot be used in an object because they create a thread of execution and an object is nothing more than a chunk memory. Even so, cover properties can be combined with covergroups to enable them to record protocol coverage, eliminate the need for state machines in monitors, and cross the protocol coverage with other interesting coverpoints.

For example, consider a class-based testbench environment with a coverage collector as shown in Figure 5:

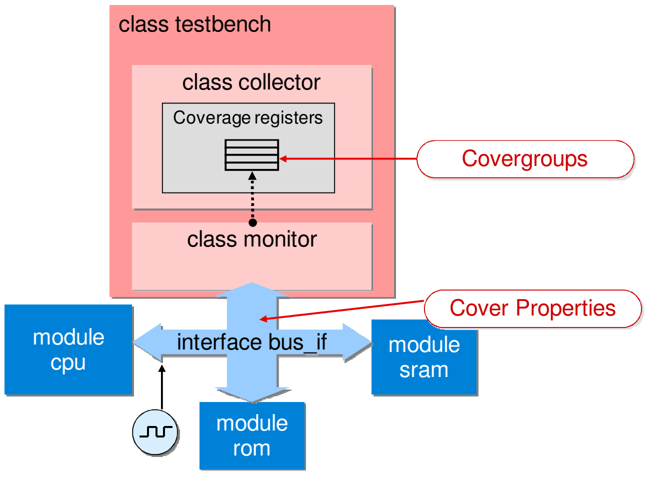


Figure 5: Class-based testbench with coverage collector.

A cover property that monitors for APB read and write transactions across the system interface might look like this:

```
interface apb_if;

   ...

   sequence apb_trans;
     t_apb_a  addr;
```

```
      t_apb_d  data;

      @(posedge PCLK)
       (
        (( PSEL && PWRITE ), addr = PADDR,
                             data = PWDATA )
          ##1
        ( PENABLE, cover_write( addr, data ))
       ) or
       (
        (( PSEL && !PWRITE ), addr = PADDR )
          ##1
        ( PENABLE, data = PRDATA,
                   cover_read ( addr, data ))
       );
    endsequence: apb_trans

    cover property ( apb_trans );

endinterface
```

This sequence implements the APB protocol, captures the address and data, and passes the information into the corresponding `cover_write()` or `cover_read()` tasks. These tasks also live within the interface, and could be implemented as follows:

```
typedef struct packed  {
  t_dir              dir;
  t_apb_a    addr;
  t_apb_d    data;
} apb_s;

apb_s  t;
bit  cov_trig = 0;
...

task cover_write( t_apb_a  addr, t_apb_d  data);
   t = { WRITE, addr, data };
   cov_trig = ~cov_trig;
 endfunction

task cover_read( t_apb_a  addr, t_apb_d  data);
   t = { READ, addr, data };
   cov_trig = ~cov_trig;
 endfunction
```

Here, the `cov_trig` variable will be used to signal the class-based monitor when there is a new transaction to grab through its virtual interface. A named event could also be used, but not all simulators provide good support for an event through a virtual interface.

In the monitor, there is no need for an APB state machine since the protocol is being monitored by the cover property. Instead, the monitor simply waits for the coverage trigger to toggle like this:

```
class monitor ...;
  task run ();
    forever begin
      apb_trans  tr;
      @( bus_if.cov_trig )
      tr = new( bus_if.t.dir,
                bus_if.t.addr,
                bus_if.t.data );
      cov_collector.write( tr );
    end
  endtask: run
  ...
endclass
```

Using this approach allows you to have the best of both worlds. The temporal syntax can be used to create the FSM to watch the bus protocol and simplify the monitor development, and then covergroups can be used in the class-based environment to record the information. Once the information is in the covergroup, cross coverage can be created, or the coverage information used for test stimulus feedback.

3.2 *Trick #2*: Create coverpoints to query bin coverage

Built-in to all covergroups, coverpoints, and crosses is a function called `get_coverage()`, which returns a real number of the current percentage of coverage. For example,

```
initial
   repeat (100) @(posedge clk) begin
      cg_inst.sample();      // Sample coverage

      if ( cg_inst.get_coverage() > 90.0 )
         cg_inst.stop();
   end
```

In this example, the `sample()` method is being used to manually sample the coverage values. The coverage percentage is then used to determine if the goal of 90.0% has been met and if so then turn off the coverage collecting.

Not only is `get_coverage()` useful for controlling coverage collection, but it can also be used to steer random stimulus. However, it is important to understand how the coverage is computed. A coverage bin is considered covered if it has reached its goal of 100%, and a coverpoint is considered covered if all its bins have reached 100%; otherwise, it is considered uncovered or 0%. In other words, say for example that a bin has been hit 4 out of 5 times, and the coverage goal is set to 5 occurrences. A simulator report will show that the bin is covered 4/5=80%. Unfortunately, querying the coverage on the coverpoint would result in 0% covered, not 80%, because the bin has not reached its coverage goal. If a coverpoint has 2 bins and one of them has reached its goal, then the coverpoint will be considered 50% covered, 3 bins with one bin covered—33%, and so on. So there is no way to query the true bin coverage, but at least whether a bin has reached its coverage goal or not can be determined.

The `get_coverage()` function works on covergroups, coverpoints, and crosses, but not on individual coverage bins. For example, given the following covergroup,

```
covergroup cg;
  coverpoint i {
    bins zero     = { 0 };
    bins tiny     = { [1:100] };
    bins hunds[3] =
            { 200,300,400,500,600,700,800,900 };
  }
endgroup
```

the querying of coverage on the `zero` bin would be illegal:

```
// ILLEGAL - not allowed!
// cov = cg_inst.i.zero.get_coverage();
```

In other words, SystemVerilog does not provide fine-grain details to the actual values covered in a coverpoint. Fortunately, there is a partial workaround. Each value of interest can be turned into a unique coverpoint so that the `get_coverage()` function can be called on each value. This syntax is somewhat cumbersome and tedious, but it accomplishes the goal. For example, defining coverage on individual opcodes for a processor design might look something like this:

```
covergroup instr_cg;
  op_nop : coverpoint instr_word[15:12] {
             bins op = { nop_op };
           }

  op_load : coverpoint instr_word[15:12] {
             bins op = { load_op };
           }

  op_store : coverpoint instr_word[15:12] {
             bins op = { store_op };
           }

  op_move : coverpoint instr_word[15:12] {
             bins op = { move_op };
           }
  ...
endgroup
```

Now with the individual coverpoints defined for each value, the coverage can be queried without any problems:

```
cov = cg_inst.op_nop.get_coverage();
```

Again, remember that the only coverage returned from these coverpoints is either 0% or 100%—i.e., either the goal has been reached or not. Per instance coverage could also be specified with the `at_least` option set for each opcode value.

3.3 *Trick #3*: Direct stimulus with coverage

Often times, engineers want to feedback coverage information directly into their constrained random stimulus generation. For example, this coverage could be used in a `randcase` to steer the direction of the random stimulus like this:

```
// Bias randomness to hit uncovered coverpoints
randcase
  (101 - cg_inst.a.get_coverage): ...;
  (101 - cg_inst.b.get_coverage): ...;
  (101 - cg_inst.c.get_coverage): ...;
  ...
endcase
```

Here, the current percentage of coverage is being used to determine the weighting in the `randcase` statement. If the coverpoint has reached its goal, then 100% will be returned so the percentage is subtracted from 101 to bias the `randcase` to choose the cases that have not been seen (without excluding it altogether, which would happen if subtracted from 100).

Another way to steer stimulus is using the SystemVerilog distribution weighting called `dist`. With the `dist` constraint, you can specify the probability that a particular value will occur. The `dist` weighting can be defined as an expression; however, not all simulators support expressions in a distribution. Alternatively, variables can be used for each value's weight. For example,

```
int           weight_nop     = 1,
              weight_load    = 1,
              weight_store   = 1,
              weight_add     = 1,
              ...;
```

```
constraint bias_opcodes {
      opcode dist {
             nop_op          := weight_nop,
             load_op         := weight_load,
             store_op        := weight_store,
             add_op          := weight_add,
             ...
      };
}
```

In this example, all the values have an equal weighting of 1 at simulation startup. As simulation progresses, these weights must be updated to affect the randomization of the opcode stimulus. Before `randomize()` is called, a method called `pre_randomized()` is invoked, which is an ideal place to update these weights that will be used in the `dist` constraint when `randomize()` is called. For example,

```
function real calc_weight( opcode_t op );
   real cov;
   case ( op )        // Grab coverage
      nop_op:
         cov = covunit.cg.op_nop.get_coverage;
      load_op:
         cov = covunit.cg.op_load.get_coverage;
      store_op:
         cov = covunit.cg.op_store.get_coverage;
      ...
   endcase

   calc_weight = (100 - cov) * 0.5;

endfunction : calc_weight


function void pre_randomize();
      // Set dist weighting
      weight_nop     += calc_weight(nop_op);
      weight_load    += calc_weight(load_op);
      weight_store   += calc_weight(store_op);
      weight_add     += calc_weight(add_op);
      ...
endfunction
```

The `calc_weight()` function is called for each opcode and the coverage updated by grabbing the current coverage and subtracting it from 100. Recall from section 3.2, the value that `get_coverage()` will return will only be 100% or 0%—i.e., covered or not covered. Therefore, if the opcode is not covered, then 100*0.5 = 5 will be added; otherwise, nothing will be added to the weighting (i.e., 0 * 0.5). Using this formula, opcodes that have been seen a lot will not increase their weighting; whereas, unseen opcodes will have a very high probability of being selected next.

One word of caution—the point of randomization is to find hard-to-find corner cases due to all the randomization. When constraining randomization like this, the stimulus is no longer *truly random*, which means that while it will quickly fill coverage matrixes, it may not do a very good job uncovering those hard-to-find bugs.

3.4 *Trick #4*: Covering cover properties

When a cover property matches a behavior, the simulator keeps track of the number of times that that behavior is attempted or matched. Unlike covergroups, which can be queried for coverage, there is no direct way to access this coverage information from within SystemVerilog; instead, it is included in a simulator's coverage report. Likewise, there are no language constructs that allow weighting of the cover property in a coverage report. While there are no direct ways in SystemVerilog to access this coverage information, there are still a few indirect ways to acquire it.

3.4.1 Solution 1: SystemVerilog only approach

The easiest way to figure out the number of matches from a cover property is to simply keep track using a counter. Cover properties allow for a statement to execute when the property is matched where a coverage counter can be incremented. For example, coverage could be store in an associative array as follows:

```
int coverage[string];        // Coverage array

c1: cover property ( a |=> b ) coverage["c1"]++;
c2: cover property ( c |=> d ) coverage["c2"]++;
```

This coverage records the number of successful attempts of the property; however, coverage can also be kept for the number of matches of either the precondition or the condition by using a function:

```
function void cov( string s );
  coverage[s]++;
endfunction

c1: cover property (a |=> ( b, cov("c1") );
c2: cover property (( c, cov("c2") ) |=> d );
```

When `c1` successfully matches the condition of `b` true, or `c2` matches the precondition of `c` true, then the coverage array is incremented by the function call to `cov()`. This coverage is then readily available within the testbench environment through the `coverage` associative array.

Of course, creating a counter does not add the cover property to the overall coverage calculation. For that, the associative array can be changed into a true-false array using a `bit` data type, and a covergroup. Since the elements will be set by different properties at different deltas, the covergroup has its `type_option.strobe` set so that coverage is only collected at the end of the time slot. After coverage is sampled, then the coverage array needs to be cleared in preparation for the next clock cycle, which can be accomplished on the opposite edge of the clock using a default assignment pattern:

```
bit coverage[string];

c1: cover property ( a |=> b ) coverage["c1"]=1;
c2: cover property ( c |=> d ) coverage["c2"]=1;

covergroup cg @(posedge clk);
  type_option.strobe = 1; // Sample end of cycle

  coverpoint coverage["c1"] {
     bins match = { 1 };
  }
  coverpoint coverage["c2"] {
     bins match = { 1 };
  }
endgroup
cg cg1 = new;

// Clear the coverage after it is sampled
always @(negedge clk)
  coverage = '{default:0};
```

In addition, this coverage can be weighted and crossed:

```
covergroup cg @(posedge clk);
  type_option.strobe = 1;
  option.per_instance = 1;

  cp_c1: coverpoint coverage["c1"] {
      bins match = { 1 };
      option.weight = 1;
  }
  cp_c2: coverpoint coverage["c2"] {
      bins match = { 1 };
      option.weight = 0.5;
  }

  cross cp_c1, cp_c2;
endgroup
```

This covergroup creates a cross matrix for every time that property `c1` occurs at the same time as `c2`, and it only records the matches. Removing the bins in the coverpoints would also record all the non-matches, creating a 2x2 coverage matrix.

3.4.2 Solution 2: DPI and VPI coverage extensions

The previous solution required keeping track of matches manually in a variable. However, the simulator already does this and the information is accessible through the VPI assertion extensions. The VPI function, `vpi_get()`, has been extended so that the number of times a property is attempted, succeeds, or fails can be obtained.

Unfortunately, most simulators do not support these assertion extensions; however, if you are fortunate enough to use such a simulator,[1] then the coverage can easily be obtained by calling the `vpi_get()` function from a DPI function call (DPI is easier because it does not require registering the user-defined functions). For example, a simple C function called `coverage()` can be defined that queries and returns a `cover property`'s coverage:

```
#include <vpi_user.h>
#include <sv_vpi_user.h>
#include <svdpi.h>

// DPI function
int coverage ( const char *pathname ) {
  vpiHandle          a_handle;
  s_vpi_error_info   error;
  PLI_INT32          obj;
  PLI_INT32          count;

  // Get a handle to the property
  if ( a_handle = vpi_handle_by_name(
                    (PLI_BYTE8 *) pathname,
                     NULL )) {

    // Check that the handle is a property
    if ((obj = vpi_get( vpiType, a_handle )) &&
       ((obj == vpiCover)||(obj == vpiAssert))){

       // Retrieve the coverage
       count = vpi_get( vpiAssertSuccessCovered,
                        a_handle );

       if ( vpi_chk_error(&error) > 0 )
            vpi_printf( "%s\n", error.message);
       else
```

[1] The following solution works with Cadence Incisive simulator 9.2-p27.

```
            return count;    // Return coverage
      } else
         vpi_printf( "%s is not a property!\n",
                      pathname);
    } else
       vpi_printf( "ERROR! Cannot find %s\n",
                    pathname );
   return -1;
 }
```

On the SystemVerilog side, the function can be imported through the DPI as:

```
import "DPI-C" context function int coverage(
input string pathname );
```

and then called inside the testbench code as:

```
c1: cover property ( @(posedge clk) a |=> b );

final
   $display( "c1 matched = %d times",
              coverage( "tb.c1" ));
```

where `tb.c1` is the hierarchical pathname to the `c1` cover property. (Notice, the `context` keyword is required here since the DPI function is accessing VPI).

3.4.3 Solution 3: DPI and VPI assertion callbacks

While many simulators have not implemented the VPI assertion extensions, most have at least implemented the *assertion callbacks*. A callback can be created for several reasons such as an assertion starting, succeeding, or failing, and these callbacks can be used for asserts or cover properties.

In the first solution above (3.4.1), a function had to be added to every property in order to record the match in a coverage array. Using callbacks and DPI, all the properties in a design can automatically be detected and monitored, and every match recorded and stored in the DPI code. For example, the following structure could be used in C to store the information (a C++ hash would also work nicely):

```
typedef struct {
      PLI_BYTE8      *pathname;
      int            count;
} cover_t;
```

and then used to create an array of structures for all the properties:

```
static cover_t *cover_info[MAX_PROPERTIES];
```

In order to enable the `cover property` callbacks, the following function could be used traverse through the design and enable the callbacks on all cover properties found:

```
void enable_property_coverage () {
   vpiHandle         iter;
   vpiHandle         c_handle;
   s_vpi_error_info  err;
   PLI_BYTE8         *fullname;

   if ((iter = vpi_iterate(vpiAssertion, 0)) !=
                                           NULL){

      while ((c_handle=vpi_scan(iter)) != NULL){

#ifndef VCS
        // Enable callbacks only on cover props
        if (vpi_get(vpiType, c_handle) ==
                                      vpiCover) {
#endif

        // Allocate the memory for the coverage
        cover_info[coverid] = (cover_t *)
                        malloc(sizeof(cover_t));

        // Store information in coverage array
        // First, grab the name. Do a string
        // copy to make it portable across
        // simulators.
        fullname = vpi_get_str( vpiFullName,
                                c_handle );
        cover_info[coverid]->pathname =
                    malloc(strlen(fullname)+1);
        strcpy(cover_info[coverid]->pathname,
               fullname);

        // Initialize the coverage count
        cover_info[coverid]->count = 0;

        // Register the callback
        if ( vpi_register_assertion_cb (
               c_handle, cbAssertionSuccess,
               foundmatch, (PLI_BYTE8 *)
               cover_info[coverid++]) == NULL){

           if ( vpi_chk_error( &err ) > 0 )
              vpi_printf( "%s\n",err.message );
           else
        vpi_printf( "ERROR! Cannot register
callback for cover property %s\n", vpi_get_str(
vpiFullName, c_handle) );

      } else
           vpi_printf("VPI: Adding coverage
callback for cover property %s\n", vpi_get_str(
vpiFullName, c_handle ));

#ifndef VCS
          }
#endif
        }
     }
}
```

The `vpi_register_assertion_cb()` function is called to register a `cbAssertionSuccess` callback; i.e., every time the cover property succeeds, the specified function, `foundmatch()`, is invoked. The `foundmatch()` function can simply increment the coverage count:

```
static PLI_INT32 foundmatch(
    PLI_INT32 reason, p_vpi_time ct,
    vpiHandle assert, p_vpi_attempt_info info,
    PLI_BYTE8* user_data ) {

    cover_t *cover = (cover_t *) user_data;

    // Increment the coverage counter
    cover->count++;

    return 0;
}
```

Lastly, a function is needed to provide access to this coverage information so a function named `coverage()` is used:

```
int coverage ( const char *pathname ) {
   vpiHandle         cover_handle;
   s_vpi_error_info  err;
   cover_t           *cover;
```

```
    int               i;

    // Get a handle to the cover property
    if ( cover_handle = vpi_handle_by_name(
                 (PLI_BYTE8 *) pathname, NULL )){

       if ( vpi_chk_error( &err ) > 0 ) {
           vpi_printf( "%s\n", err.message );
           return -1;
       }
    }

    // Find the coverage in the array
    for ( i = 0; i < MAX_PROPERTIES; i++ ) {
       if ( strcmp( cover_info[i]->pathname,
                    pathname) == 0 )

          // Return the coverage
          return cover_info[i]->count;
    }

    vpi_printf( "ERROR! Could not find coverage
 for %s\n", pathname );
    return -1;
 }
```

In SystemVerilog, the DPI functions is imported, remembering to specify the `context` keyword since VPI is being used:

```
import "DPI-C" context function void
enable_property_coverage();
import "DPI-C" context function int coverage(
input string pathname );
```

To turn on the cover property coverage, call the `enable_property_coverage()` function at time zero in an initial block:

```
initial
   enable_property_coverage();
```

and then the coverage is collected and stored, and queried in a testbench by calling the `coverage()` function:

```
final
   $display("c1 matched = %d times",
             coverage( "tb.c1" ));
```

where `tb.c1` is the pathname to the `cover property` labeled `c1`. Also note, this method can be used to keep track of assertion coverage as well to see how well test cases are stimulating the assertion checks.

Callbacks provide the ability to keep track of the coverage information in any way desired. To add this coverage into the overall coverage calculation, an exported SystemVerilog function could be called from the callback routine that samples the values into a covergroup similar to the solution provided in section 3.4.1.

4. COVERAGE GOTCHAS

While SystemVerilog coverage has a few things to be desired (hence, the coverage tricks in the previous section), there are a few features to avoid or at least use with caution.

4.1 Gotcha #1: Avoid illegal_bins

The `illegal_bins` keyword can be used to remove unused or illegal values from the overall coverage calculation. For example,

```
logic [2:0] opcode;

covergroup cg @(posedge clk iff decode);
  coverpoint opcode {
    bins move_op[] = { 3'b000, 3'b001 };
    bins alu_op = {[3'b010:3'b011],
                   [3'b101:3'b110]};
    bins jump_op = {3'b111};
    illegal_bins unused_op = {3'b100};
  }
```

The `illegal_bins` directive also throws errors, which begs the question, "Should a passive covergroup actively throw error messages?" and "If the covergroup is relied on for checking, what happens when coverage is turned off?"

A better option is to use the `ignore_bins` keyword. `ignore_bins` will remove the values from the coverage calculation without throwing the error. If a check is really needed for an illegal value, then write an assertion!

4.2 Gotcha #2: Avoid using default

The keyword `default` is used as a catch-all for all other possible values that have not already been thrown into a bin. In the following example, the `others[] = default` will create a bin for every value not yet specified:

```
bit [15:0] i;
covergroup cg_Short @(posedge Clock);
  coverpoint i {
    bins zero     = { 0 };
    bins tiny     = { [1:100] };
    bins hunds[3] = { 200, 300, 400, 500, 600,
                      700, 800, 900 };
    bins huge     = { [1000:$] };
    ignore_bins ignore = { [501:599] };

    bins others[] = default;
  }
endgroup
```

At first glance, `default` would appear quite useful. However, there are two possible issues. First, what if the coverpoint has a very large number of values? Some simulators croak on the above example:

```
# ** Fatal: The number of singleton values
exceeded the system limit of 2147483647 for
unconstrained array bin 'other' in Coverpoint
'a' of Covergroup instance '\/covunit/cg_i'.
```

The solution to this is to not use the open range with `default`. Instead, use the following:

```
bins others = default;
```

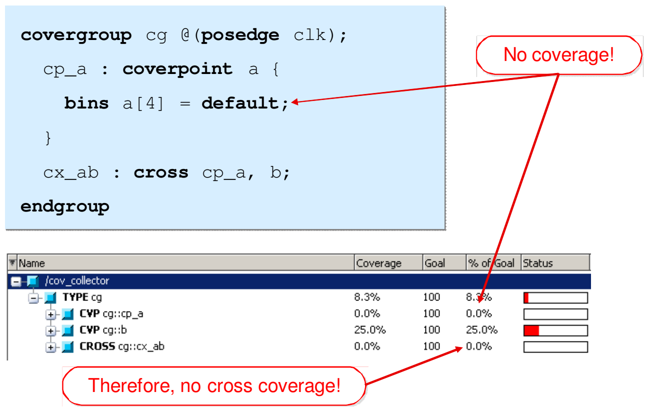
which buckets all other values into one bin called "others".

Secondly, `default` pulls values out of the coverage calculation. For example, suppose you wanted a shorthand way of taking all possible values and dividing them into several bins. Then you wanted to cross those values with another coverpoint. The obvious way to do this would be to use the `default` statement:

```
covergroup cg @(posedge clk);
  cp_a : coverpoint a {
    bins a[4] = default;
  }
```

```
   cx_ab : cross cp_a, b;
 endgroup
```

However, the problem with this example is that the coverpoint `cp_a` will have no coverage collected for it because it is using the `default` keyword. If the coverpoint has no coverage, then neither will the cross (see Figure 6).



```
covergroup cg @(posedge clk);
  cp_a : coverpoint a {
    bins a[4] = default;
  }
  cx_ab : cross cp_a, b;
endgroup
```

Name	Coverage	Goal	% of Goal	Status
/cov_collector				
TYPE cg	8.3%	100	8.3%	
CVP cg::cp_a	0.0%	100	0.0%	
CVP cg::b	25.0%	100	25.0%	
CROSS cg::cx_ab	0.0%	100	0.0%	

Figure 6: `default` removes bins from coverage calculation.

Again, instead of using `default`, use `$` or `wildcard bins`. The `$` specifies the minimum or maximum possible values and the `wildcard` allows the use of wildcard patterns:

```
 bins huge           = { [1000:$] }; // Max values
 wildcard bins a[4] = { 'b?0 };     // Even values
```

4.3 Gotcha #3: Sequence coverage versus property coverage

With the rich temporal syntax of SVA, having the ability to use it to describe and cover behavior is a very useful feature. However, coverage of a property is treated slighty differently than coverage on a sequence. Coverage of properties is defined to include ([1], 17.13.3):

- Number of times attempted
- Number of times succeeded
- Number of times failed
- Number of times succeeded due to vacuity

where a vacuous success refers to a property that uses an implication and the implication precondition is not satisfied. Coverage of a sequence is defined as only covering:

- Number of times attempted
- Number of times matched

Often, when a property is written for an assertion, it will also be covered. For example, suppose a property is written to describe the correct read or write behavior, and then the same property is covered using `cover property` to record how many times the reads or writes occur. The intention in doing so is usually to record how many times the behavior is *matched*, but assertions typically use properties[2], which record the number of successes, including all the times when the precondition is not matched (i.e., vacuous successes). The result is that the coverage does not accurately reflect what is really happening because it records vacuous coverage. The same thing happens when a `cover property` is used inside of a procedure:

```
 always @( posedge clk )
    if ( a & b )
       cp1: cover property ( c );
```

Here, not only does the sampling event get inferred from the always block's sensitivity list, but there is an implicit implication because of the context. The equivalent `cover property` is:

```
 cover property ( @(posedge clk) a & b |-> c );
```

Because the implication operator is inferred, this cover property is treated as coverage of a property instead of coverage on a sequence (i.e., recording vacuous successes and not only successful matches).

Likewise, another time when a `cover property` records vacuous coverage is when a `disable iff` is used inside a `cover property`. The `disable iff` construct is only allowed inside a property and not a sequence. So for example, adding a `disable iff (reset)` inside a `cover property` automatically treats the coverage as property coverage, which includes vacuous successes. A disabled property is successful on every cycle that reset is asserted since the property is considered vacuously true. The solution would be to avoid using `disable iff` in a `cover property` and instead use the throughout sequence operator:

```
 !reset throughout ( my_seq )
```

On the other hand, covering a sequence may not always produce the desired results. While sequence coverage records the number of times matched, it may match many times on the same attempt—all of which are included in the coverage count. For example, if a range is used in a sequence and multiple matches are possible, then all matches will be recorded. Of course, the `first_match()` sequence operator could be used to avoid recording multiple matches on the same attempt.

Another issue arises from vagueness in the SV-2005 standard. The expression `a ##1 b` is a sequence, but properties can be made of both properties and sequences. So the question arises, is

```
 cover property ( a ##1 b );
```

covering a sequence or a property? Either type of coverage could be recorded for such a statement—it depends on the implementation.

The bottom line is, if vacuous coverage is unwanted, then avoid property operators in a `cover property` (e.g., implication, `disable iff`, etc.); if coverage of multiple matches on a sequence is unwanted, then make sure to use the `first_match()` operator.

[2] Any assertion that uses the implication operator (|-> or |=>) is automatically a property and not just a sequence.

5. SV 1800-2009 ENHANCEMENTS

The SystemVerilog standard was updated in 2009 [3], and several additions and modifications were made that affect coverage. First, coverage can now be explicitly specified on a sequence and not just a property:

```
cover sequence ( @(event) disable iff ( expr )
                 sequence_expr );
```

As with a `cover property` on a sequence, the number of matches are recorded instead of the vacuous successes. Notice, the syntax also allows for the use of `disable iff` while still collecting sequence coverage instead of property coverage.

Another change made to coverage is with covergroups. The `sample()` method for a `covergroup` can now be overridden, allowing different arguments to be passed into the `covergroup` based on different contexts. For example, a `covergroup` could be created as follows:

```
covergroup cg with function sample( T_state a,
                                    int     b );
   covergroup a;
   covergroup b;
   cross a, b;
endgroup

cg cg1 = new;
```

Now, the `sample()` method could be called using local variables as in a `cover property`:

```
property state_cov;
   int i;

   @(posedge clk) ( sel, i=mode ) ##1
          ( enable, cg1.sample( state, i ));
endproperty

cover property ( state_cov );
```

A few other subtle changes were also added to covergroup options like calculating cumulative coverage by merging instance coverage instead of using weighted averages (`type_option.merge_instances=1`) and enabling the tracking of instance coverage with the `get_instance_coverage()` method (`option.get_instance_coverage=1`).

6. CONCLUSION

Despite of the shortcomings mentioned in this paper, still SystemVerilog provides adequate support for gathering the coverage needed to verify a design. Covergroups have a rich set of options and syntax, which should meet the need of the most serious verification effort. In fact, with a few tricks most shortcomings can be worked around while still accomplishing the task at hand. There are a few gotchas to watch out for—one of which that has been solved with the latest SV-2009 standard, but these are not so much as shortcomings as simply behaviors to be aware of when defining coverage.

7. REFERENCES

[1] Dudani, S., Cerny, E., Korchemny, D., Seligman, E., and Piper, L. 2009. "Verification case studies: evolution from SVA 2005 to SVA 2009." Proceedings of DVCon (February 24-25, 2009).

[2] IEEE Std 1800^{TM}-2005. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE Computer Society, New York, 2005.

[3] IEEE Std 1800^{TM}-2009. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE Computer Society, New York, 2009.