Chapter 9
Functional Coverage

As designs become more complex, the only effective way to verify them effectively is with constrained-random testing (CRT). This approach elevates you above the tedium of writing individual directed tests, one for each feature in the design. However, if your testbench is taking a random walk through the space of all design states, how do you know if you have reached your destination? Even directed tests should be double checked with functional coverage. Over the life of a project, small changes in the DUT's timing or functionality can subtly alter the results from a directed test, so it no longer verifies the same features. Whether you are using random or directed stimulus, you can gauge progress using coverage.

Functional coverage is a measure of which design features have been exercised by the tests. Start with the design specification and create a verification plan with a detailed list of what to test and how. For example, if your design connects to a bus, your tests need to exercise all the possible interactions between the design and bus, including relevant design states, delays, and error modes. The verification plan is a map to show you where to go. For more information on creating a verification plan, see Bergeron (2006).

In many complex systems, you may never achieve 100% coverage as schedules don't allow you to reach every possible corner case. After all, you didn't have time to write directed tests to get sufficient coverage, and even CRT is limited by the time it takes you to create and debug test cases, and analyze the results.

Figure 9.1 shows the feedback loop to analyze the coverage results and decide on which actions to take in order to converge on 100% coverage. Your first choice is to run existing tests with more seeds; the second is to build new constraints. Only resort to creating directed tests if absolutely necessary.

Back when you exclusively wrote directed tests, the verification planning was limited. If the design specification listed 100 features, all you had to do was write 100 tests. Coverage was implicit in the tests — the "register move" test moved all combinations of registers back and forth. Measuring progress was easy: if you had completed 50 tests, you were halfway done. This chapter uses "explicit" and "implicit" to describe how coverage is specified. Explicit coverage is described

C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, DOI 10.1007/978-1-4614-0715-7_9,

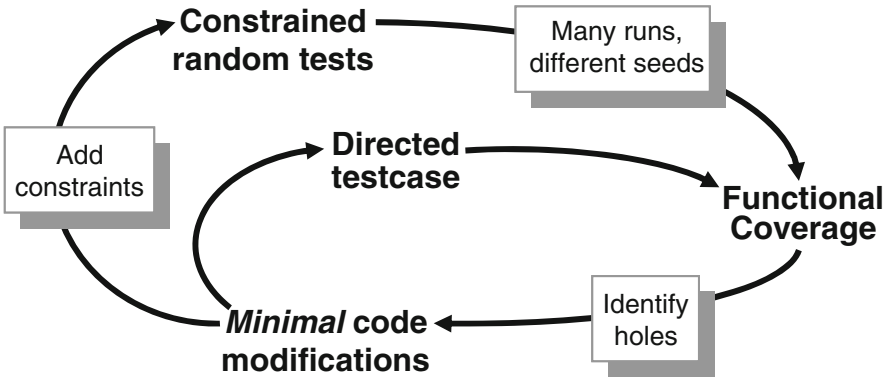

Fig. 9.1 Coverage convergence

directly in the test environment using SystemVerilog features. Implicit coverage is implied by a test — when the "register move" directed test passes, you have hopefully covered all register transactions.

With CRT, you are freed from hand crafting every line of input stimulus, but now you need to write code that tracks the effectiveness of the test with respect to the verification plan. You are still more productive, as you are working at a higher level of abstraction. You have moved from tweaking individual bits to describing the interesting design states. Reaching for 100% functional coverage forces you to think more about what you want to observe and how you can direct the design into those states.

9.1 Gathering Coverage Data

You can run the same random testbench over and over, simply by changing the random seed to generate new stimulus. Each individual simulation generates a database of functional coverage information, the trail of footprints from the random walk. You can then merge all this information together to measure your overall progress using functional coverage as shown in Figure 9.2.

You then analyze the coverage data to decide how to modify your tests. If the coverage levels are steadily growing, you may just need to run existing tests with new random seeds, or even just run longer tests. If the coverage growth has started to slow, you can add additional constraints to generate more "interesting" stimuli. When you reach a plateau, some parts of the design are not being exercised, so you need to create more tests. Lastly, when your functional coverage values near 100%, check the bug rate. If bugs are still being found, you may not be measuring true coverage for some areas of your design. Don't be in too big of a rush to reach 100% coverage, which just shows that you looked for bugs in all the usual places. While you are trying to verify your design, take many random walks through the stimulus space; this can create many unanticipated combinations, as shown in van der Schoot (2007).

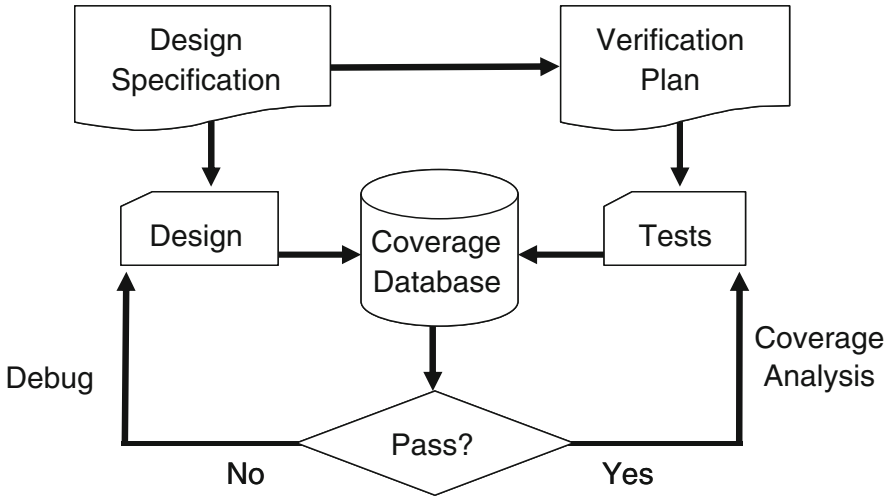

Fig. 9.2 Coverage flow

Each simulation vendor has its own format for storing coverage data and as well as its own analysis tools. You need to perform the following actions with those tools.

- **Run a test with multiple seeds**. For a given set of constraints and coverage groups, compile the testbench and design into a single executable. Now you need to run this constraint set over and over with different random seeds. You can use the Unix system clock as a seed, but be careful, as your batch system may start multiple jobs simultaneously. These jobs may run on different servers or may start on a single server with multiple processors. So combine all these values to make a truly unique seeds. The seed must be saved with the simulation and coverage results for repeatability.
- **Check for pass/fail**. Functional coverage information is only valid for a successful simulation. When a simulation fails because there is a design bug, the coverage information must be discarded. The coverage data measures how many items in the verification plan are complete, and this plan is based on the design specification. If the design does not match the specification, the coverage values are useless. Some verification teams periodically measure all functional coverage from scratch so that it reflects the current state of the design.
- **Analyze coverage across multiple runs**. You need to measure how successful each constraint set is, over time. If you are not yet getting 100% coverage for the areas that are targeted by the constraints, but the amount is still growing, run more seeds. If the coverage level has plateaued, with no recent progress, it is time to modify the constraints. Only if you think that reaching the last few test cases for one particular section may take too long for constrained-random simulation should you consider writing a directed test. Even then, continue to use random stimulus for the other sections of the design, in case this "background noise" finds a bug.

9.2 Coverage Types

Coverage is a generic term for measuring progress to complete design verification. Your simulations slowly paint the canvas of the design, as you try to cover all of the legal combinations. The coverage tools gather information during a simulation and then post-process it to produce a coverage report. You can use this report to look for coverage holes and then modify existing tests or create new ones to fill the holes. This iterative process continues until you are satisfied with the coverage level.

9.2.1 Code Coverage

The easiest way to measure verification progress is with code coverage. Here you are measuring how many lines of code have been executed (line coverage), which paths through the code and expressions have been executed (path coverage), which single-bit variables have had the values 0 or 1 (toggle coverage), and which states and transitions in a state machine have been visited (FSM coverage). You don't have to write any extra HDL code. The tool instruments your design automatically by analyzing the source code and adding hidden code to gather statistics. You then run all your tests, and the code coverage tool creates a database.

Most simulators include a code coverage tool. A post-processing tool converts the database into a readable form. The end result is a measure of how much your tests exercise the design code. Note that you are primarily concerned with analyzing the design code, not the testbench. Untested design code could conceal a hardware bug, or may be just redundant code.

Code coverage measures how thoroughly your tests exercised the "implementation" of the design specification, but not the verification plan. Just because your tests have reached 100% code coverage, your job is not done. What if you made a mistake that your test didn't catch? Worse yet, what if your implementation is missing a feature? The module in Sample 9.1 is for a D-flip flop. Can you see the mistake?

Sample 9.1 Incomplete D-flip flop model missing a path

```
module dff(output logic q, q_l,
           input  logic clk, d, reset_l);

  always @(posedge clk or negedge reset_l) begin
    q <= d;
    q_l <= !d;
  end
endmodule
```

The reset logic was accidently left out. A code coverage tool would report that every line had been exercised, yet the model was not implemented correctly. Go back to the functional specification that describes reset behavior and make sure your verification plan includes a requirement to verify this. Then gather functional coverage information on the design during reset.

9.2.2 Functional Coverage

The goal of verification is to ensure that a design behaves correctly in its real environment, be that an MP3 player, network router, or cell phone. The design specification details how the device should operate, whereas the verification plan lists how that functionality is to be stimulated, verified, and measured. When you gather measurements on what functions were covered, you are performing "design" coverage. For example, the verification plan for a D-flip flop would mention not only its data storage but also how it resets to a known state. Until your test checks both these design features, you will not have 100% functional coverage.

Functional coverage is tied to the design intent and is sometimes called "specification coverage," while code coverage measures how well you have tested the RTL code and is known as, "implementation coverage." These are two very different metrics. Consider what happens if a block of code is missing from the design. Code coverage cannot catch this mistake and could report that you have executed 100% of the lines, but functional coverage will show that the functionality does not exist.

9.2.3 Bug Rate

An indirect way to measure coverage is to look at the rate at which fresh bugs are found, show in the graph in Fig. 9.3. You should keep track of how many bugs you found each week, over the life of a project. At the start, you may find many bugs through inspection as you create the testbench. As you read the design spec, you may find inconsistencies, which hopefully are fixed before the RTL is written. Once the testbench is up and running, a torrent of bugs is found as you check each module in the system. The bug rate drops, hopefully to zero, as the design nears tape-out. However, you are not yet done. Every time the rate sags, it is time to find different ways to create corner cases.

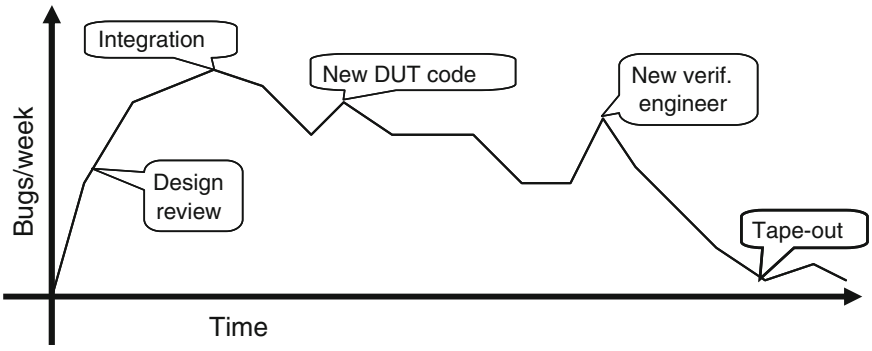

Fig. 9.3 Bug rate during a project

The bug rate can vary per week based on many factors such as project phases, recent design changes, blocks being integrated, personnel changes, and even vacation schedules. Unexpected changes in the rate could signal a potential problem.

As shown in Fig. 9.3, it is not uncommon to keep finding bugs even after tape-out, and even after the design ships to customers.

9.2.4 Assertion Coverage

Assertions are pieces of declarative code that check the relationships between design signals, either once or over a period of time. These can be simulated along with the design and testbench, or proven by formal tools. Sometimes you can write the equivalent check using SystemVerilog procedural code, but many assertions are more easily expressed using SystemVerilog Assertions (SVA).

Assertions can have local variables and perform simple data checking. If you need to check a more complex protocol, such as determining whether a packet successfully went through a router, procedural code is often better suited for the job. There is a large overlap between sequences that are coded procedurally or using SVA. See Vijayaraghavan and Ramanadhan (2005), Cohen et al. (2005), and Chapters 3 and 7 in the VMM book, Bergeron et al. (2006) for more information on SVA.

The most familiar assertions look for errors such as two signals that should be mutually exclusive or a request that was never followed by a grant. These error checks should stop the simulation as soon as they detect a problem. Assertions can also check arbitration algorithms, FIFOs, and other hardware. These are coded with the `assert property` statement.

Some assertions might look for interesting signal values or design states, such as a successful bus transaction. These are coded with the `cover property` statement. You can measure how often these assertions are triggered during a test by using assertion coverage. A cover property observes sequences of signals, whereas a cover group (described below) samples data values and transactions during the simulation. These two constructs overlap in that a cover group can trigger when a sequence completes. Additionally, a sequence can collect information that can be used by a cover group.

9.3 Functional Coverage Strategies

Before you write the first line of test code, you need to anticipate what are the key design features, corner cases, and possible failure modes. This is how you write your verification plan. Don't think in terms of data values only; instead, think about what information is encoded in the design. The plan should spell out the significant design states.

9.3.1 Gather Information, not Data

A classic example is a FIFO. How can you be sure you have thoroughly tested a 1K FIFO memory? You could measure the values in the read and write indices,

but there are over a million possible combinations. Even if you were able to simulate that many cycles, you would not want to read the coverage report.

At a more abstract level, a FIFO can hold from 0 to N–1 possible values. So what if you just compare the read and write indices to measure how full or empty the FIFO is? You would still have 1K coverage values. If your testbench pushed 100 entries into the FIFO, then pushed in 100 more, do you really need to know if the FIFO ever had 150 values? Not as long as you can successfully read out all values.

The corner cases for a FIFO are Full and Empty. If you can make the FIFO go from Empty (the state after reset) through Full and back down to Empty, you have covered all the levels in between. Other interesting states involve the indices as they pass between all 1's and all 0's. A coverage report for these cases is easy to understand.

You may have noticed that the interesting states are independent of the FIFO size. Once again, look at the information, not the data values.

Design signals with a large range (more than a few dozen possible values) should be broken down into smaller ranges, plus corner cases. For example, your DUT may have a 32-bit address bus, but you certainly don't need to collect 4 billion samples. Check for natural divisions such as memory and IO space. For a counter, pick a few interesting values, and always try to rollover counter values from all 1's back to 0.

9.3.2 Only Measure What you are Going to Use

Gathering functional coverage data can be expensive, so only measure what you will analyze and use to improve your tests. Your simulations may run slower as the simulator monitors signals for functional coverage, but this approach has lower overhead than gathering waveform traces and measuring code coverage. Once a simulation completes, the database is saved to disk. With multiple testcases and multiple seeds, you can fill disk drives with functional coverage data and reports. But if you never look at the final coverage reports, don't perform the initial measurements.

There are several ways to control cover data: at compilation, instantiation, or triggering. You could use switches provided by the simulation vendor, conditional compilation, or suppression of the gathering of coverage data. The last of these is less desirable because the post-processing report is filled with sections with 0% coverage, making it harder to find the few enabled ones.

9.3.3 Measuring Completeness

Like your kids in the backseat on a family vacation, your manager constantly asks you, "Are we there yet?" How can you tell if you have fully tested a design? You need to look at all coverage measurements and consider the bug rate to see if you have reached your destination.

At the start of a project, both code and functional coverage are low. As you develop tests, run them over and over with different random seeds until you no longer see increasing values of functional coverage. Create additional constraints and

tests to explore new areas. Save test/seed combinations that give high coverage, so that you can use them in regression testing.

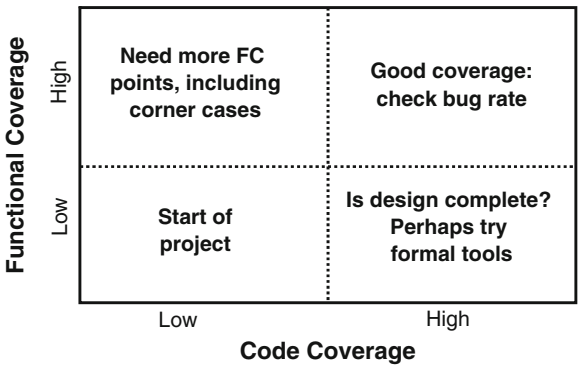

Fig. 9.4 Coverage comparison

What if the functional coverage is high but the code coverage is low as shown in the upper left of Figure 9.4? Your tests are not exercising the full design, perhaps from an inadequate verification plan. It may be time to go back to the hardware specifications and update your verification plan. Then you need to add more functional coverage points to locate untested functionality.

A more difficult situation is high code coverage but low functional coverage. Even though your testbench is giving the design a good workout, you are unable to put it in all the interesting states. First, see if the design implements all the specified functionality. If it is there, but your tests can't reach it, you might need a formal verification tool that can extract the design's states and create appropriate stimulus.

The goal is both high code and functional coverage. However, don't plan your vacation yet. What is the trend of the bug rate? Are significant bugs still popping up?

Worse yet, are they being found deliberately, or did your testbench happen to stumble across a particular combination of states that no one had anticipated? On the other hand, a low bug rate may mean that your existing strategies have run out of steam, and you should look into different approaches. Try different approaches such as new combinations of design blocks and error generators.

9.4 Simple Functional Coverage Example

To measure functional coverage, you begin with the verification plan and write an executable version of it for simulation. In your System Verilog testbench, sample the values of variables and expressions. These sample locations are known as cover points. Multiple cover points that are sampled at the same time (such as when a transaction completes) are placed together in a cover group.

Sample 9.2 has a transaction that comes in eight flavors. The testbench generates the `dst` variable randomly, and the verification plan requires that every value be tried.

Sample 9.2 Functional coverage of a simple object

```
program automatic test(busifc.TB ifc);

  class Transaction;
    rand bit [31:0] data;
    rand bit [ 2:0] dst;           // Eight dst port numbers
  endclass

  Transaction tr;                  // Transaction to be sampled

  covergroup CovDst2;
    coverpoint tr.dst;             // Measure coverage
  endgroup

  initial begin
    CovDst2 ck;
    ck = new();                    // Instantiate group
    repeat (32) begin              // Run a few cycles
      @ifc.cb;                     // Wait a cycle
      tr = new();
      `SV_RAND_CHECK(tr.randomize); // Create a transaction
      ifc.cb.dst  <= tr.dst;       //   and transmit
      ifc.cb.data <= tr.data;      //   onto interface
      ck.sample();                 // Gather coverage
    end
  end
endprogram
```

Sample 9.2 creates a random transaction and drives it out to an interface. The testbench samples the value of the `dst` field using the `CovDst2` cover group. Eight possible values, 32 random transactions — did your testbench generate them all? Samples 9.3 and 9.4 have part of a coverage report from VCS. Because of randomization, every simulator will give different results.

As you can see, the testbench generated `dst` values of 1, 2, 3, 4, 5, 6, and 7, but never generated a 0. The `at least` column specifies how many hits are needed before a bin is considered covered. See Section 9.10.3 for the `at_least` option.

To improve your functional coverage, the easiest strategies are to run more simulation cycles, or to try new random seeds. For Sample 9.2, the very next random transaction (#33) has a dst value of 0, giving 100% coverage. Or, if you started simulation with a different seed, you may reach 100% in fewer transactions, for this trivial case. On a real design, you may see a plateau in coverage, with most coverage points getting hit more and more, but a few stubborn points that are never hit, no matter how long you run, regardless of seed values. In this case, you probably have to try a new strategy, as the testbench is not creating the proper stimulus. The most important part of any coverage report are the points with 0 hits.

Sample 9.3 Coverage report for a simple object

```
Coverpoint Coverage report
CoverageGroup: CovDst2
    Coverpoint: tr.dst
Summary
    Coverage: 87.50
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 7
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin            # hits    at least
    ================================
    auto[1]        7         1
    auto[2]        7         1
    auto[3]        1         1
    auto[4]        5         1
    auto[5]        4         1
    auto[6]        2         1
    auto[7]        6         1
    ================================
```

Sample 9.4 Coverage report for a simple object, 100% coverage

```
Coverpoint Coverage report
CoverageGroup: CovDst2
    Coverpoint: tr.dst
Summary
    Coverage: 100
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 8
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin            # hits    at least
    ================================
    auto[0]        1         1
    auto[1]        7         1
    auto[2]        7         1
    auto[3]        1         1
    auto[4]        5         1
    auto[5]        4         1
    auto[6]        2         1
    auto[7]        6         1
    ================================
```

This book gives a rough explanation of how coverage is calculated. The LRM has a very detailed explanation of coverage computation across four pages, with more details across an entire chapter. Consult it for the most accurate details.

9.5 Anatomy of a Cover Group

A cover group is similar to a class — you define it once and then instantiate it one or more times. It contains cover points, options, formal arguments, and an optional trigger. A cover group encompasses one or more data points, all of which are sampled at the same time.

You should create very clear cover group names that explicitly indicate what you are measuring and, if possible, reference to the verification plan. The name `Parity_Errors_In_Hexaword_Cache_Fills` may seem verbose, but when you are trying to read a coverage report that has dozens of cover groups, you will appreciate the extra detail. You can also use the comment option for additional descriptive information, as shown in Section 9.9.2.

A cover group can be defined in a class or at the program or module level. It can sample any visible variable such as program/module variables, signals from an interface, or any signal in the design (using a hierarchical reference). A cover group inside a class can sample variables in that class, as well as data values from embedded objects.

Don't define the cover group in a data class, such as a transaction, as doing so can cause additional overhead when gathering coverage data. Imagine you are trying to track how many beers were consumed by patrons in a pub. Would you try to follow every bottle as it flowed from the loading dock, over the bar, and into each person? No, instead you could just have each patron check off the type and number of beers consumed, as shown in van der Schoot (2006).

In SystemVerilog, you should define cover groups at the appropriate level of abstraction. This level can be at the boundary between your testbench and the design, in the transactors that read and write data, in the environment configuration class, or wherever is needed. The sampling of any transaction must wait until it is actually received by the DUT. If you inject an error in the middle of a transaction, causing it to be aborted in transmission, you need to change how you treat it for functional coverage. You need to use a different cover point that has been created just for error handling.

A class can contain multiple cover groups. This approach allows you to have separate groups that can be enabled and disabled as needed. Additionally, each group may have a separate trigger, allowing you to gather data from many sources.

A cover group must be instantiated for it to collect data. If you forget, no error message about null handles is printed at run time, but the coverage report will not contain any mention of the cover group. This rule applies for cover groups defined either inside or outside of classes.

9.5.1 *Defining a Cover Group in a Class*

A cover group can be defined in a program, module, or class. In all cases, you must explicitly instantiate it to start sampling. If the cover group is defined in a class, it is known as an embedded covergroup. In this case, you do not make a separate name when you construct it; just use the original cover group name. You must construct an embedded covergroup in the class's constructor, as opposed to a non-embedded cover group that can be constructed at any time.

Sample 9.5 is very similar to the first example of this chapter except that it embeds a cover group in a transactor class, and thus does not need a separate instance name.

Sample 9.5 Functional coverage inside a class

```
class Transactor;
  Transaction tr;
  mailbox #(Transaction) mbx;
  covergroup CovDst5;
    coverpoint tr.dst;
  endgroup

  function new(input mailbox #(Transaction) mbx);
    CovDst5 = new();              // Instantiate covergroup
    this.mbx = mbx;
  endfunction

  task run();
    forever begin
      mbx.get(tr);                // Get next transaction
      @ifc.cb;
      ifc.cb.dst <= tr.dst;       // Send into DUT
      ifc.cb.data <= tr.data;
      CovDst5.sample();           // Gather coverage
    end
  endtask

endclass
```

9.6 Triggering a Cover Group

The two major parts of functional coverage are the sampled data values and the time when they are sampled. When new values are ready (such as when a transaction has completed), your testbench triggers the cover group. This can be done directly with the `sample` function, as shown in Sample 9.5, or by using a coverage event in the `covergroup` definition. The coverage event can use a `@` to block on signals or events.

Use `sample` if you want to explicitly trigger coverage from procedural code, if there is no existing signal or event that tells when to sample, or if there are multiple instances of a cover group that trigger separately.

Use the coverage event in the `covergroup` declaration if you want to tap into existing events or signals to trigger coverage.

9.6.1 Sampling Using a Callback

One of the better ways to integrate functional coverage into your testbench is to use callbacks, as originally shown in Section 8.7. This technique allows you to build a flexible testbench without restricting when coverage is collected. You can decide for every point in the verification plan where and when values are sampled. And if you need an extra "hook" in the environment for a callback, you can always add one in an unobtrusive manner, as a callback only "fires" during simulations when the test registers a callback object. You can create many separate callbacks for each cover group, with little overhead. As explained in Section 8.7.4, callbacks are superior to using a mailbox to connect the testbench to the coverage objects. You might need multiple mailboxes to collect transactions from different points in your testbench. A mailbox requires a transactor to receive transactions, and multiple mailboxes cause you to juggle multiple threads. Instead of an active transactor, use a passive callback.

Sample 8.26–8.28 shows a driver class that has two callback points, before and after the transaction is transmitted. Sample 8.26 shows the base callback class, and Sample 8.28 has a test with an extended callback class that sends data to a scoreboard. Make your own extension, `Driver_cbs_coverage`, of the base callback class, `Driver_cbs`, to call the sample task for your cover group in `post_tx`. Push an instance of the coverage callback class into the driver's callback queue, and your coverage code triggers the cover group at the right time. Samples 9.6 and 9.7 define and use the callback `Driver_cbs_coverage`.

Sample 9.6 Test using functional coverage callback

```
program automatic test;
  Environment env;

  initial begin
    Driver_cbs_coverage dcc;

    env = new();
    env.gen_cfg();
    env.build();

    // Create and register the coverage callback
    dcc = new();
    env.drv.cbs.push_back(dcc); // Put into driver's Q

    env.run();
    env.wrap_up();
  end

endprogram
```

The UVM recommends gathering coverage by monitoring the DUT and sending transactions to a coverage component through an analysis port, similar to a mailbox.

Sample 9.7 Callback for functional coverage

```
class Driver_cbs_coverage extends Driver_cbs;
  covergroup CovDst7;
    ...
  endgroup

  virtual task post_tx(ref Transaction tr);
    CovDst7.sample();          // Sample coverage values
  endtask
endclass
```

9.6.2 Cover Group with a User Defined Sample Argument List

In Sample 9.5, the cover group samples a variable in transaction object that is defined inside the class. If your cover group is defined outside of a class, you can pass variables through the `sample` method by defining your own argument list. Now you can sample variables from anywhere in the testbench.

In Sample 9.8, the cover group is expanded to also cover the low data bit. The last statement of the `run` method passes the destination address and also the configuration variable for high speed mode.

Sample 9.8 Defining an argument list to the sample method

```
covergroup CovDst8 with function sample(bit [2:0] dst, bit hs);
  coverpoint dst;
  coverpoint hs;                          // High speed mode
endgroup

class Transactor;
  CovDst8 covdst;
  task run();
    forever begin
      mbx.get(tr);                        // Get next transaction
      ifc.cb.dst  <= tr.dst;              // Send into DUT
      ifc.cb.data <= tr.data;
      covdst.sample(tr.dst, high_speed);  // Gather coverage
    end
  endtask
endclass
```

9.6.3 Cover Group with an Event Trigger

In Sample 9.9, the cover group `CovDst9` is sampled when the testbench triggers the `trans_ready` event.

Sample 9.9 Cover group with a trigger

```
event trans_ready;
covergroup CovDst9 @(trans_ready);
    coverpoint ifc.cb.dst;   // Measure coverage
endgroup
```

The advantage of using an event over calling the `sample` method directly is that you may be able to use an existing event such as one triggered by an assertion, as shown in Sample 9.11.

9.6.4 Triggering on a System Verilog Assertion

If you already have an SVA that looks for useful events like a complete transaction, you can add an event trigger to wake up the cover group as shown in 9.10.

Sample 9.10 Module with SystemVerilog Assertion

```
module mem(simple_bus sb);
  bit [7:0] data, addr;
  event write_event;

  cover property
   (@(posedge sb.clk) sb.write_ena==1)
    -> write_event;
endmodule
```

Sample 9.11 Triggering a cover group with an SVA

```
program automatic test(simple_bus sb);

  covergroup Write_cg @($root.top.m1.write_event);
    coverpoint $root.top.m1.data;
    coverpoint $root.top.m1.addr;
  endgroup

  Write_cg wcg;

  initial begin
    wcg = new();
    sb.write_ena <= 1;   // Apply stimulus here
    #10000ns $finish;
  end
endprogram
```

9.7 Data Sampling

How is coverage information gathered? When you specify a variable or expression in a cover point, SystemVerilog creates a number of "bins" to record how many times each value has been seen. These bins are the basic units of measurement for functional coverage. If you sample a one-bit variable, a maximum of two bins are created. You can imagine that System Verilog drops a token in one or the other bin every time the cover group is triggered. At the end of each simulation, a database is created with all bins that have a token in them. You then run an analysis tool that reads all databases and generates a report with the coverage for each part of the design and for the total coverage.

9.7.1 Individual Bins and Total Coverage

To calculate the coverage for a point, you first have to determine the total number of possible values, also known as the domain. There may be one value per bin or multiple

values. Coverage is the number of sampled values divided by the number of bins in the domain.

A cover point that is a 3-bit variable has the domain 0:7 and is normally divided into eight bins. If, during simulation, values belonging to seven bins are sampled, the report will show 7/8 or 87.5% coverage for this point. All these points are combined to show the coverage for the entire group, and then all the groups are combined to give a coverage percentage for all the simulation databases.

This is the status for a single simulation. You need to track coverage over time. Look for trends so you can see where to run more simulations or add new constraints or tests. Now you can better predict when verification of the design will be completed.

9.7.2 Creating Bins Automatically

As you saw in the report in Sample 9.3, System Verilog automatically creates bins for cover points. It looks at the domain of the sampled expression to determine the range of possible values. For an expression that is N bits wide, there are 2^N possible values. For the 3-bit variable `dst`, there are 8 possible values. The range of an enumerated type is shown in Section 9.6.8. The domain for enumerated data types is the number of named values. You can also explicitly define bins as shown in Section 9.6.5.

9.7.3 Limiting the Number of Automatic Bins Created

The cover group option `auto_bin_max` specifies the maximum number of bins to automatically create, with a default of 64 bins. If the domain of values in the cover point variable or expression is greater than this option, System Verilog divides the range into `auto_bin_max` bins. For example, a 16-bit variable has 65,536 possible values, so each of the 64 bins covers 1024 values.

In reality, you may find this approach impractical, as it is very difficult to find the needle of missing coverage in a haystack of auto-generated bins. Lower this limit to 8 or 16, or better yet, explicitly define the bins as shown in Section 9.6.5.

Sample 9.12 takes the chapter's first example and adds a cover point option that sets `auto_bin_max` to two bins. The sampled variable is still `dst`, which is three bits wide, for a domain of eight possible values. The first bin holds the lower half of the range, 0–3, and the other hold the upper values, 4–7.

Sample 9.12 Using `auto_bin_max` set to 2

```
covergroup CovDst12;
  coverpoint tr.dst
    { option.auto_bin_max = 2; }  // Divide into 2 bins
endgroup
```

The coverage report from VCS shows the two bins. This simulation achieved 100% coverage because the eight `dst` values were mapped to two bins. Since both bins have sampled values, your coverage is 100% as shown in Sample 9.13.

Sample 9.13 Report with `auto_bin_max` set to 2

```
Bin              # hits    at least
================================
auto[0:3]        15        1
auto[4:7]        17        1
```

Sample 9.12 used `auto_bin_max` as an option for the cover point only. You can also use it as an option for the entire group as shown in Sample 9.14.

Sample 9.14 Using `auto_bin_max` for all cover points

```
covergroup CovDst14;
  option.auto_bin_max = 2; // Affects dst & data
  coverpoint tr.dst;       // autobin[0:3], autobin[4:7]
  coverpoint tr.data;      // autobin[0:7], autobin[8:15]
endgroup
```

9.7.4 Sampling Expressions

You can sample expressions, but always check the coverage report to be sure you are getting the values you expect. You may have to adjust the width of the computed expression, as shown in Section 2.16. For example, sampling a 3-bit header length (0:7) plus a 4-bit payload length (0:15) creates only 2^4 or 16 bins, which may not be enough if your transactions can actually be from 0 to 22 bytes long.

Sample 9.15 Using an expression in a cover point

```
class Packet;
  rand bit [2:0] hdr_len;      // range: 0:7
  rand bit [3:0] payload_len;  // range: 0:15
  rand bit [3:0] kind;
endclass

Packet p;

covergroup CovLen15;
  len16: coverpoint (p.hdr_len + p.payload_len);
  len32: coverpoint (p.hdr_len + p.payload_len + 5'b0);
endgroup
```

Sample 9.15 has a cover group that samples the total packet length. The cover point has a label to make it easier to read the coverage report. Also, the expression has an additional dummy constant so that the transaction length is computed with 5-bit precision, for a maximum of 32 auto-generated bins.

A long run with random packets showed that the `len16` had 100% coverage, but this is across only 16 bins. (The cover point only has 16 bins as the sum of a 3-bit and 4-bit value is only 4-bits in Verilog.) The cover point `len32` had 72% coverage across 32 bins. (The addition of a 5-bit value to the expression for `bin32` results in a 5-bit result.) Neither of these cover points are correct, as the maximum length has a domain of 0:22 (0+0:7+15). The auto-generated bins just don't work, as the maximum length is not a power of 2. You need a way to precisely define bins.

9.7.5 User-Defined Bins Find a Bug

Automatically generated bins are okay for anonymous data values, such as counter values, addresses, or values that are a power of 2. For other values, you should explicitly name the bins to improve accuracy and ease coverage report analysis. System Verilog automatically creates bin names for enumerated types, but for other variables you need to give names to the interesting states. The easiest way to specify bins is with the `[]` syntax, as shown in Sample 9.16.

Sample 9.16 Defining bins for transaction length

```
covergroup CovLen16;
  len: coverpoint (p.hdr_len + p.payload_len + 5'b0)
    {bins len[] = {[0:23]}; }  // Bug?? See below
endgroup
```

After sampling many random transactions, the group has 95.83% coverage. A quick look at the report in Sample 9.17 shows the problem — the length of 23 (17 hex) was never seen. The longest header is 7, and the longest payload is 15, for a total of 22, not 23! If you change to the bins declaration to use 0:22, the coverage jumps to 100%. The user-defined bins found a bug in the test.

Sample 9.17 Coverage report for transaction length

```
Bin         # hits    at least
============================
len_00      13        1
len_01      36        1
len_02      51        1
len_03      60        1
len_04      72        1
len_05      88        1
len_06      127       1
len_07      122       1
len_08      133       1
len_09      138       1
len_0a      115       1
len_0b      128       1
len_0c      125       1
len_0d      111       1
len_0e      115       1
len_0f      134       1
len_10      107       1
len_11      102       1
len_12      70        1
len_13      65        1
len_14      39        1
len_15      30        1
len_16      19        1
len_17      0         1
============================
```

9.7.6 Naming the Cover Point Bins

Sample 9.18 samples a 4-bit variable, `kind`, that has 16 possible values. The first bin is called `zero` and counts the number of times that `kind` is 0 when sampled. The next four values, 1–3 and 5, are all grouped into a single bin, `lo`. The upper eight values, 8–15, are kept in separate bins, `hi_8`, `hi_9`, `hi_a`, `hi_b`, `hi_c`, `hi_d`, `hi_e`, and `hi_f`. Note how `$` in the `hi` bin expression is used as a shorthand notation for the largest value for the sampled variable. Lastly, `misc` holds all values that were not previously chosen: 4, 6, and 7.

Sample 9.18 Specifying bin names

```
covergroup CovKind18;
  coverpoint p.kind {
    bins zero = {0};          // 1 bin for kind==0
    bins lo   = {[1:3], 5};   // 1 bin for values 1:3, 5
    bins hi[] = {[8:$]};      // 8 separate bins: 8...15
    bins misc = default;      // 1 bin for rest, does not count
  }                           // No semicolon
endgroup
```

Note that the additional information about the `coverpoint` is grouped using curly braces: `{}`. This is because the bin specification is declarative code, not procedural code that would be grouped with `begin…end`. Lastly, the final curly brace is NOT followed by a semicolon, just as an `end` never is.

Now you can easily see in Sample 9.19 which bins have no hits — `hi_8` in this case.

Sample 9.19 Report showing bin names

```
Bin        # hits    at least
============================
hi_8         0         1
hi_9         5         1
hi_a         3         1
hi_b         4         1
hi_c         2         1
hi_d         2         1
hi_e         9         1
hi_f         4         1
lo          16         1
misc        15         1
zero         1         1
```

When you define the bins, you are restricting the values used for coverage to those that are interesting to you. SystemVerilog no longer automatically creates bins, and it ignores values that do not fall into a predefined bin. More importantly, only the bins you create are used to calculate functional coverage. You get 100% coverage only as long as you get a hit in every specified bin.

Values that do not fall into any specified bin are ignored. This rule is useful if the sampled value, such as transaction length, is not a power of 2. If you are specifying bins, you can use the `default` bin specifier to catch values that you may have forgotten. However, the LRM says that `default` bins are not used in coverage calculation.

In Sample 9.18, the range for `hi` uses a dollar sign (`$`) on the right side to specify the upper value. This is a very useful shortcut - now you can let the compiler calculate

the limits for a range. You can use the dollar sign on the left side of a range to specify the lower limit. In Sample 9.20, the `$` in the range for bin `neg` represents the negative number furthest from zero: 32'h8000_0000, or -2,147,483,648, whereas the $ in bin `pos` represents the largest signed positive value, 32'h7FFF_FFFF, or 2,147,483,647.

Sample 9.20 Specifying ranges with $

```
int i;
covergroup range_cover;
  coverpoint i {
    bins neg  = {[$:-1]};  // Negative values
    bins zero = {0};       // Zero
    bins pos  = {[1:$]};   // Positive values
  }
endgroup
```

9.7.7 Conditional Coverage

You can use the `iff` keyword to add a condition to a cover point. The most common reason for doing so is to turn off coverage during reset so that stray triggers are ignored. Sample 9.21 gathers only values of `dst` when `rst` is 0, where `rst` is active-high.

Sample 9.21 Conditional coverage — disable during reset

```
covergroup CovDst21;
  // Don't gather coverage when rst==1
  coverpoint tr.dst iff (!bus_if.rst);
endgroup
```

Alternately, you can use the `start` and `stop` functions to control individual instances of cover groups as shown in Sample 9.22.

Sample 9.22 Using `stop` and `start` functions

```
initial begin
  CovDst22 ck = new();       // Instantiate cover group

  // Reset sequence stops collection of coverage data
  #1ns ck.stop();
  bus_if.rst <= 1;

  #100ns bus_if.rst <= 0; // End of reset
  ck.start();
  ...
end
```

9.7.8 Creating Bins for Enumerated Types

For enumerated types, SystemVerilog creates a bin for each value as you can see in Sample 9.23.

Sample 9.23 Functional coverage for an enumerated type

```
typedef enum {INIT, DECODE, IDLE} fsmstate_e;
fsmstate_e pstate, nstate;   // declare typed variables
covergroup CovFsm23;
  coverpoint pstate;
endgroup
```

Here is part of the coverage report from VCS, Sample 9.24 showing the bins for the enumerated types.

Sample 9.24 Coverage report with enumerated types

```
Bin              # hits     at least
================================
auto_DECODE      11         1
auto_IDLE        11         1
auto_INIT        10         1
================================
```

If you want to group multiple values into a single bin, you have to define your own bins. Any bins outside the enumerated values are ignored unless you define a bin with the `default` specifier. When you gather coverage on enumerated types, `auto_bin_max` does not apply.

9.7.9 Transition Coverage

You can specify state transitions for a cover point. In this way, you can tell not only what interesting values were seen but also the sequences. For example, you can check if `dst` ever went from 0 to 1, 2, or 3 as shown in Sample 9.25.

Sample 9.25 Specifying transitions for a cover point

```
covergroup CovDst25;
  coverpoint tr.dst {
    bins t1 = (0 => 1), (0 => 2), (0 => 3);
  }
endgroup
```

You can quickly specify multiple transitions using ranges. The expression `(1, 2 => 3, 4)` creates the four transitions `(1=>3)`, `(1=>4)`, `(2=>3)`, and `(2=>4)`.

You can specify transitions of any length. Note that you have to sample once for each state in the transition. So `(0 => 1 => 2)` is different from `(0 => 1 => 1 => 2)`

or `(0 => 1 => 1 => 1 => 2)`. If you need to repeat values, as in the last sequence, you can use the shorthand form: `(0 => 1[*3] => 2)`. To repeat the value 1 for 3, 4, or 5 times, use `1[*3:5]`.

9.7.10 Wildcard States and Transitions

You use the `wildcard` keyword to create multiple states and transitions. Any `X`, `Z`, or `?` in the expression is treated as a wildcard for 0 or 1. Sample 9.26 creates a cover point with a bin for even values and one for odd.

Sample 9.26 Wildcard bins for a cover point

```
bit [2:0] dst;
covergroup CovDst26;
  coverpoint tr.dst {
    wildcard bins even = {3'b??0};
    wildcard bins odd  = {3'b??1};
  }
endgroup
```

9.7.11 Ignoring Values

With some cover points, you never get all possible values. For instance, a 3-bit variable may be used to store just six values, 0–5. If you use automatic bin creation, you never get beyond 75% coverage. There are two ways to solve this problem. You can explicitly define the bins that you want to cover as shown in Section 9.6.5. Alternatively, you can let SystemVerilog automatically create bins, and then use `ignore_bins` to tell which values to exclude from functional coverage calculation like in Sample 9.27.

Sample 9.27 Cover point with ignore_bins

```
covergroup CovDst27;
  coverpoint tr.dst {
    ignore_bins hi = {6,7};   // Ignore upper 2 bins
  }
endgroup
```

The original range of `low_ports_0_5`, a three-bit variable is 0:7. The `ignore_bins` excludes the last two bins, which reduces the range to 0:5. So total coverage for this group is the number of bins with samples, divided by the total number of bins, which is 5 in this case.

Sample 9.28 Cover point with `auto bin max` and `ignore bins`

```
covergroup CovDst28;
  coverpoint tr.dst {
    option.auto_bin_max = 4; // 0:1, 2:3, 4:5, 6:7
    ignore_bins hi = {6,7};  // Ignore upper 2 values
  }
endgroup
```

If you define bins either explicitly or by using the `auto_bin_max` option, and then ignore them, the ignored bins do not contribute to the calculation of coverage. In Sample 9.28, four bins are initially created using the `auto_bin_max` option: 0:1, 2:3, 4:5, and 6:7. However, then the uppermost bin is eliminated by `ignore_bins`, so in the end only three bins are created. This cover point can have coverage of 0%, 33%, 66%, or 100%.

9.7.12 Illegal Bins

Some sampled values not only should be ignored, but also should cause an error if they are seen. This is best done in the testbench's monitor code, but can also be done by labeling a bin with `illegal_bins` as shown in Sample 9.29. Use this to catch states that were missed by the test's error checking. This also double-checks the accuracy of your bin creation: if an illegal value is found by the cover group, it is a problem either with the testbench or with your bin definitions.

Sample 9.29 Cover point with `illegal_bins`

```
covergroup CovDst29;
  coverpoint tr.dst {
    illegal_bins hi = {6,7}; // Give error if seen
  }
endgroup
```

9.7.13 State Machine Coverage

You should have noticed that if a cover group is used on a state machine, you can use bins to list the specific states, and transitions for the arcs. However, this does not mean you should use SystemVerilog's functional coverage to measure state machine coverage. You would have to extract the states and arcs manually. Even if you did this correctly the first time, you might miss future changes to the design code. Instead, use a code coverage tool that extracts the state register, states, and arcs automatically, saving you from possible mistakes.

However, an automatic tool extracts the information exactly as coded, mistakes and all. You may want to monitor small, critical state machines manually using functional coverage.

9.8 Cross Coverage

A cover point records the observed values of a single variable or expression. You may want to know not only what bus transactions occurred but also what errors happened during those transactions, and their source and destination. For this you need cross coverage that measures what values were seen for two or more cover points at the same time. Note that when you measure cross coverage of a variable with `N` values, and of another with `M` values, SystemVerilog needs `N×M` cross bins to store all the combinations.

9.8.1 Basic Cross Coverage Example

Previous examples have measured coverage of the transaction kind, and destination port number, but what about the two combined? Did you try every kind of transaction into every port? The `cross` construct in SystemVerilog records the combined values of two or more cover points in a group. The `cross` statement takes only cover points or a simple variable name. If you want to use expressions, hierarchical names or variables in an object such as `handle.variable`, you must first specify the expression in a `coverpoint` with a label and then use the label in the `cross` statement.

Sample 9.30 creates cover points for `tr.kind` and `tr.dst`. Then the two points are crossed to show all combinations. SystemVerilog creates a total of 128 (8×16) bins. Be careful: even a simple cross can result in a very large number of bins.

Sample 9.30 Basic cross coverage

```
class Transaction;
  rand bit [3:0] kind;
  rand bit [2:0] dst;
endclass

Transaction tr;

covergroup CovDst30;
  kind: coverpoint tr.kind;  // Create cover point kind
  dst: coverpoint tr.dst;    // Create cover point dst
  cross kind, dst;           // Cross kind and dst
endgroup
```

A random testbench created 56 transactions and produced the coverage report in Sample 9.31. Note that even though all possible `kind` and `dst` values were generated, only 1/3 of the cross combinations were seen. This is a very typical result. Also note that the total coverage for the group is the cross coverage plus the coverage for `kind` and `dst`.

Sample 9.31 Coverage summary report for basic cross coverage

```
Cumulative report for Transaction::CovDst30
Summary:
    Coverage: 78.91
    Goal: 100

Coverpoint           Coverage     Goal          Weight
=======================================================
kind                   100.00      100          1
dst                    100.00      100          1
=======================================================
Cross                  Coverage    Goal         Weight
=======================================================
Transaction::CovDst30  78.91       100          1

Cross Coverage report
CoverageGroup: Transaction::CovDst30
    Cross: Transaction::CovDst30
Summary
    Coverage: 36.72
    Goal: 100
    Coverpoints Crossed: kind dst
    Number of Expected Cross Bins: 128
    Number of User Defined Cross Bins: 0
    Number of Automatically Generated Cross Bins: 47


    Automatically Generated Cross Bins

    kind       dst            # hits       at least
 ================================================
    auto[0]    auto[0]        1            1
    auto[0]    auto[1]        2            1
    auto[0]    auto[2]        1            1
    auto[0]    auto[5]        1            1
...
```

9.8.2 *Labeling Cross Coverage Bins*

If you want more readable cross coverage bin names, you can label the individual cover point bins as demonstrated in Sample 9.32, and SystemVerilog will use these names when creating the cross bins.

Sample 9.32 Specifying cross coverage bin names

```
covergroup CovDstKind32;
  dst: coverpoint tr.dst
   {bins dst[] = {[0:$]};
   }
  kind: coverpoint tr.kind
   {bins zero = {0};        // 1 bin for kind==0
    bins lo   = {[1:3]};    // 1 bin for values 1:3
    bins hi[] = {[8:$]};    // 8 separate bins
    bins misc = default;    // 1 bin for rest, does not count
  }
  cross kind, dst;
endgroup
```

If you define bins that contain multiple values, the coverage statistics change. In the report below, the number of bins has dropped from 128 to 80. This is because `kind` has 10 bins: `zero`, `lo`, `hi_8`, `hi_9`, `9hi_a`, `hi_b`, `hi_c`, `hi_d`, `hi_e`, and `hi_f`. Remember that the `misc` bin, which defined its values with `default`, does not add to the coverage total. The percentage of coverage jumped from 87.5% to 90.91% as shown in Sample 9.33 because any single value in the `lo` bin, such as 2, allows that bin to be marked as covered, even if the other values, 1 or 3, are not seen.

Sample 9.33 Cross coverage report with labeled bins

```
Summary
    Coverage: 90.91
    Number of Coverpoints Crossed: 2
    Coverpoints Crossed: kind dst
    Number of Expected Cross Bins: 88
    Number of Automatically Generated Cross Bins: 80
    Automatically Generated Cross Bins

    dst          kind           # hits       at least
 ========================================================
    dst_0        hi_8           3            1
    dst_0        hi_a           1            1
    dst_0        hi_b           4            1
    dst_0        hi_c           4            1
    dst_0        hi_d           4            1
    dst_0        hi_e           1            1
    dst_0        lo             7            1
    dst_0        zero           1            1
    dst_1        hi_8           3            1
...
```

9.8.3 Excluding Cross Coverage Bins

To reduce the number of bins, use `ignore_bins`. With cross coverage, you specify the cover point with `binsof` and the set of values with `intersect` so that a single `ignore_bins` construct can sweep out many individual bins.

Sample 9.34 Excluding bins from cross coverage

```
covergroup CovDst34;
  dst: coverpoint tr.dst
   {bins dst[] = {[0:$]};
   }
  kind: coverpoint tr.kind {
    bins zero = {0};        // 1 bin for kind==0
    bins lo   = {[1:3]};    // 1 bin for values 1:3
    bins hi[] = {[8:$]};    // 8 separate bins
    bins misc = default;    // 1 bin for rest, does not count
  }
  cross kind, dst {
    ignore_bins hi = binsof(dst) intersect {7};
    ignore_bins md = binsof(dst) intersect {0} &&
                     binsof(kind) intersect {[9:11]};
    ignore_bins lo = binsof(kind.lo);
  }
endgroup
```

The first `ignore_bins` in Sample 9.34 just excludes bins where `dst` is 7 and any value of `kind`. Since `kind` is a 4-bit value, this statement excludes 12 bins, as `misc`'s values of 4–7 don't count because of the `default`. The second `ignore_bins` is more selective, ignoring bins where `dst` is 0 and `kind` is 9, 10, or 11, for a total of 3 bins.

The `ignore_bins` can use the bins defined in the individual cover points. The `ignore_bins lo` uses bin names to exclude `kind.lo` that is 1, 2, or 3. The bins must be names defined at compile time, such as `zero` and `lo`. The bins `hi_8`, `hi_9`, `hi_a`,... `hi_f`, and any automatically generated bins do not have names that can be used at compile time in other statements such as `ignore_bins`; these names are created at run time or during the report generation.

Note that `binsof` uses parentheses `()` while `intersect` specifies a range and therefore uses curly braces `{}`.

9.8.4 Excluding Cover Points from the Total Coverage Metric

The total coverage for a group is based on all simple cover points and cross coverage. If you are only sampling a variable or expression in a `coverpoint` to be used in a `cross` statement, you should set its weight to 0 so that it does not contribute to the total coverage.

Sample 9.35 Specifying cross coverage weight

```
covergroup CovDst35;
  kind: coverpoint tr.kind
   {bins zero = {0};
    bins lo   = {[1:3]};
    bins hi[] = {[8:$]};
    type_option.weight = 5;      // Count in total
  }
  dst: coverpoint tr.dst
   {bins dst[] = {[0:$]};
    type_option.weight = 0;      // Don't count towards total
  }
  cross kind, dst
    {type_option.weight = 10;}  // Give cross extra weight
endgroup
```

There are two types of options: those that are specific to an instance of a covergroup and those that specify an option for the covergroup type as a whole. The instance specific options are like local variables and are specified with the `option` keyword, as in `option.auto_bin_max=2` from Sample 9.12. The alternatives are specified with the `type_option` keyword and are tied to the cover group, like static variables in a class. In Sample 9.35, `type_option.weight` applies to all instances of this group. The LRM has a detailed explanation of the difference, and this book shows the most common options and their usage.

9.8.5 Merging Data from Multiple Domains

One problem with cross coverage is that you may need to sample values from different timing domains. You might want to know if your processor ever received an interrupt in the middle of a cache fill. The interrupt hardware is separate from and may use different clocks than the cache hardware, making it difficult to know when to trigger the cover group. On the other hand, you want to make sure you have tested this case, as a previous design had a bug of this very sort.

The solution is to create a timing domain separate from the cache or interrupt hardware. Make copies of the signals into temporary variables and then sample them in a new coverage group that measures the cross coverage.

9.8.6 Cross Coverage Alternatives

As your cross coverage definition becomes more elaborate, you may spend considerable time specifying which bins should be used and which should be ignored. You may have two random bits, `a` and `b` with three interesting states, `{a==0, b==0}`, `{a==1, b==0}`, and `{b==1}`.

Sample 9.36 shows how you can name bins in the cover points and then gather cross coverage using those bins.

Sample 9.36 Cross coverage with bin names

```
class Sample;
  rand bit a, b;
endclass

Sample sam;

covergroup CrossBinNames;
  a: coverpoint sam.a
     { bins a0 = {0};
       bins a1 = {1};
       option.weight=0;}    // Don't count this coverpoint
  b: coverpoint sam.b
     { bins b0 = {0};
       bins b1 = {1};
       option.weight=0;}    // Don't count this coverpoint
  ab: cross a, b
     { bins a0b0 = binsof(a.a0) && binsof(b.b0);
       bins a1b0 = binsof(a.a1) && binsof(b.b0);
       bins b1   = binsof(b.b1); }
endgroup
```

Sample 9.37 gathers the same cross coverage, but now uses `binsof` to specify the cross coverage values.

Sample 9.37 Cross coverage with `binsof`

```
covergroup CrossBinsofIntersect;
  a: coverpoint sam.a
     { option.weight=0; }   // Don't count this coverpoint
  b: coverpoint sam.b
     { option.weight=0; }   // Don't count this coverpoint
  ab: cross a, b
       { bins a0b0 = binsof(a) intersect {0} &&
                     binsof(b) intersect {0};
         bins a1b0 = binsof(a) intersect {1} &&
                     binsof(b) intersect {0};
         bins b1   = binsof(b) intersect {1}; }
endgroup
```

Alternatively, you can make a cover point that samples a concatenation of values. Then you only have to define bins using the less complex cover point syntax.

Sample 9.38 Mimicking cross coverage with concatenation

```
covergroup CrossManual;
  ab: coverpoint {sam.a, sam.b}
    { bins a0b0 = {2'b00};
      bins a1b0 = {2'b10};
      wildcard bins b1 = {2'b?1};
    }
endgroup
```

Use the style in Sample 9.36 if you already have bins defined for the individual cover points and want to use them to build the cross coverage bins. Use Sample 9.37 if you need to build cross coverage bins but have no pre-defined cover point bins. Use Sample 9.38 if you want the tersest format.

9.9 Generic Cover Groups

As you start writing cover groups, you will find that some are very similar to one another. SystemVerilog allows you to create a generic cover group so that you can specify a few unique details when you instantiate it.

9.9.1 Pass Cover Group Arguments by Value

Sample 9.39 shows a cover group that uses an argument to split the range into two halves. Just pass the midpoint value to the cover groups' `new` function.

Sample 9.39 Covergroup with simple argument

```
class Transaction;
  bit [2:0] dst;      // Values: 0:7
endclass
Transaction tr;

covergroup CovDst39 (int mid);
  coverpoint tr.dst
   {bins lo = {[0:mid-1]};
    bins hi = {[mid:$]};
  }
endgroup

CovDst39 cp;
initial
  cp = new(5);            // lo=0:4, hi=5:7
```

9.9.2 Pass Cover Group Arguments by Reference

You can specify a variable to be sampled with pass-by-reference. Here you want the cover group to sample the value during the entire simulation, not just to use the value when the constructor is called.

Sample 9.40 Pass-by-reference

```
bit [2:0] dst_a, dst_b;

covergroup CovDst40 (ref bit [2:0] dst, input int mid);
  coverpoint dst {
    bins lo = {[0:mid-1]};
    bins hi = {[mid:$]};
  }
endgroup

CovDst40 cpa, cpb;
initial
  begin
    cpa = new(dst_a, 4);  // dst_a, lo=0:3, hi=4:7
    cpb = new(dst_b, 2);  // dst_b, lo=0:1, hi=2:7
  end
```

Like a task or function, the arguments to a cover group have a sticky direction. In Sample 9.40, if you forgot the `input` direction, the `mid` argument will have the direction `ref`. The example would not compile because you cannot pass a constant (4 or 2) into a `ref` argument.

9.10 Coverage Options

You can specify additional information in the cover group using options. There are two flavors of options: instance options that apply to a specific cover group instance and type options that apply to all instances of the cover group, and are analogous to static data members of classes. Options can be placed in the cover group so that they apply to all cover points in the group, or they can be put inside a single cover point for finer control. You have already seen the `auto_bin_max` and `weight` options. Here are several more.

9.10.1 Per-Instance Coverage

If your testbench instantiates a coverage group multiple times, by default SystemVerilog groups together all the coverage data from all the instances. However, if you

have several generators, each creating very different streams of transactions, you will need to see separate reports. For example, one generator may be creating long transactions while another makes short ones. The cover group in Sample 9.41 can be instantiated in each separate generator. It keeps track of coverage for each instance, and has a unique comment string with the hierarchical path to the cover group instance.

Sample 9.41 Specifying per-instance coverage

```
covergroup CoverLength(ref bit [2:0] len);
  coverpoint len;
  option.per_instance = 1;
endgroup
```

The per-instance option can only be given in the cover group, not in the cover point or cross point.

9.10.2 Cover Group Comment

You can add a comment into coverage reports to make them easier to analyze. A comment could be as simple as the section number from the verification plan to tags used by a report parser to automatically extract relevant information from the sea of data. If you have a cover group that is only instantiated once, use the `type_option` as shown in Sample 9.42.

Sample 9.42 Specifying comments for a cover group

```
covergroup CovDst42;
  type_option.comment = "Section 3.2.14 Dst port numbers";
  coverpoint tr.dst;
endgroup
```

However, if you have multiple instances, you can give each a separate comment, as long as you also use the `per-instance` option as shown in Sample 9.43.

Sample 9.43 Specifying comments for a cover group instance

```
covergroup CovDst43(int lo,hi, string comment);
  option.comment = comment;
  option.per_instance = 1;
  coverpoint tr.dst
    {bins range = {[lo:hi]};
     }
endgroup
...
CovDst43 cd_lo = new(0,3, "Low dst numbers");
CovDst43 cd_hi = new(4,7, "High dst numbers");
```

9.10.3 Coverage Threshold

You may not have sufficient visibility into the design to gather robust coverage information. Suppose you are verifying that a DMA state machine can handle bus errors. You don't have access to its current state, but you know the range of cycles that are needed for a transfer. So if you repeatedly cause errors during that range, you have probably covered all the states. So you could set `option.at_least` to 8 or more to specify that after 8 hits on a bin, you are confident that you have exercised that combination.

If you define `option.at_least` at the cover group level, it applies to all cover points. If you define it inside a point, it only applies to that single point.

However, as Sample 9.2 showed, even after 32 attempts, the random `kind` variable still did not hit all possible values. So only use `at_least` if there is no direct way to measure coverage, like when the testbench can not probe the DUT details.

9.10.4 Printing the Empty Bins

By default, the coverage report shows only the bins with samples. Your job is to verify all that is listed in the verification plan, so you are actually more interested in the bins without samples. Use the option `cross_num_print_missing` to tell the simulation and report tools to show you all bins, especially the ones with no hits. Set it to a large value, as shown in Sample 9.44, but no larger than you are willing to read.

Sample 9.44 Report all bins including empty ones

```
covergroup CovDst44;
  kind: coverpoint tr.kind;
  dst: coverpoint tr.dst;
  cross kind, dst;
  option.cross_num_print_missing = 1_000;
endgroup
```

9.10.5 Coverage Goal

The goal for a cover group or point is the level at which the group or point is considered fully covered. The default is 100% coverage. If you set this level below 100% like in Sample 9.45, you are requesting less than complete coverage, which is probably not desirable. This option affects only the coverage report.

Sample 9.45 Specifying the coverage goal

```
covergroup CovDst45;
  coverpoint tr.dst;
  option.goal = 90;  // Settle for partial coverage
endgroup
```

9.11 Analyzing Coverage Data

In general, assume you need more seeds and fewer constraints. After all, it is easier to run more tests than to construct new constraints. If you are not careful, new constraints can easily restrict the search space.

If your cover point has only zero or one sample, your constraints are probably not targeting these areas at all. You need to add constraints that "pull" the solver into new areas. In Sample 9.16, the transaction length had an uneven distribution. Sample 9.46 shows the full class. This situation is similar to the distribution seen when you roll two dice and look at the total value.

Sample 9.46 Original class for packet length

```
class Packet;
  rand bit [2:0] hdr_len;
  rand bit [3:0] payload_len;
  rand bit [4:0] len;
  constraint length {len == hdr_len + payload_len; }
endclass
```

The problem with this class is that `len` is not evenly weighted. Look in the coverage report and note how the low and high values are rarely hit. Figure 9.5 is a graph of the values from the report.

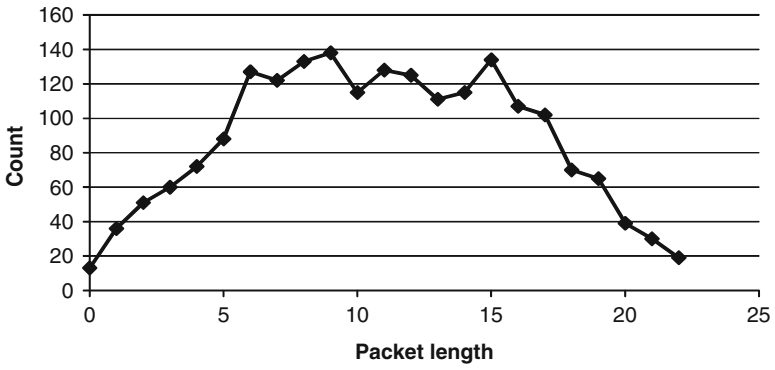

Fig. 9.5 Uneven probability for packet length

If you want to make the total length be evenly distributed, use a `solve...before` constraint as shown in Sample 9.47 and plotted in Fig. 9.6.

Sample 9.47 `solve...before` constraint for packet length

```
constraint length
  {len == hdr_len + payload_len;
   solve len before hdr_len, payload_len; }
```

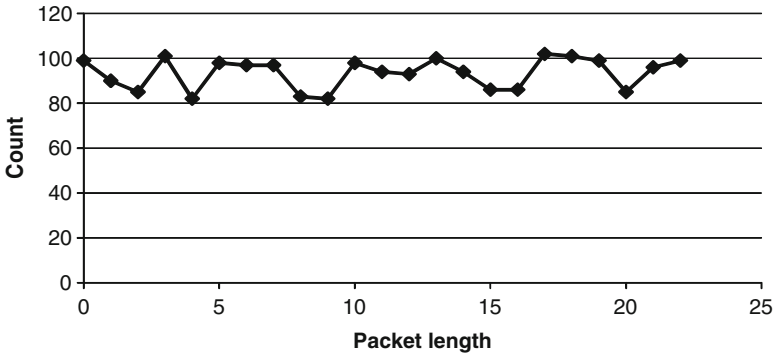

Fig. 9.6 Even probability for packet length with `solve...before`

The normal alternative to `solve...before` is the `dist` constraint. However, this does not work, as `len` is also being constrained by the sum of the two lengths.

9.12 Measuring Coverage Statistics During Simulation

You can query the level of functional coverage on the fly during simulation. This allows you to check whether you have reached your coverage goals, and possibly to control a random test.

At the global level, you can get the total coverage of all cover groups with `$get_coverage`, which returns a real number between 0. and 100. This system task looks across all cover groups.

You can narrow down your measurements with the `get_coverage()` and `get_inst_coverage()` methods. The first function works with both cover group names and instances to give coverage across all instances of a cover group, for example `CoverGroup::get_coverage()` or `cgInst.get_coverage()`. The second function returns coverage for a specific cover group instance, for example `cgInst.get_inst_coverage()`. You need to specify `option.per_instance=1` if you want to gather per-instance coverage.

The most practical use for these functions is to monitor coverage over a long test. If the coverage level does not advance after a given number of transactions or cycles, the test should stop. Hopefully, another seed or test will increase the coverage.

While it would be nice to have a test that can perform some sophisticated actions based on functional coverage results, it is very hard to write this sort of test. Each test + random seed pair may uncover new functionality, but it may take many runs to reach a goal. If a test finds that it has not reached 100% coverage, what should it do? Run for more cycles? How many more? Should it change the stimulus being generated? How can you correlate a change in the input with the level of functional coverage? The one reliable thing to change is the random seed, which you should only do once per simulation. Otherwise, how can you reproduce a design bug if the stimulus depends on multiple random seeds?

You can query the functional coverage statistics if you want to create your own coverage database. Verification teams have built their own SQL databases that are fed functional coverage data from simulation. This setup allows them greater control over the data, but requires a lot of work outside of creating tests.

Some formal verification tools can extract the state of a design and then create input stimulus to reach all possible states. Don't try to duplicate this in your testbench!

9.13 Conclusion

When you switch from writing directed tests, hand-crafting every bit of stimulus, to constrained-random testing, you might worry that the tests are no longer under your command. By measuring coverage, especially functional coverage, you regain control by knowing what features have been tested.

Using functional coverage requires a detailed verification plan and much time creating the cover groups, analyzing the results, and modifying tests to create the proper stimulus. This may seem like a lot of work, but is less effort than would be required to write the equivalent directed tests. Additionally, the time spent in gathering coverage helps you better track your progress in verifying your design.

9.14 Exercises

1. For the class below, write a covergroup to collect coverage on the test plan requirement, "All ALU opcodes must be tested." Assume the opcodes are valid on the positive edge of signal `clk`.

```
typedef enum {ADD, SUB, MULT, DIV} opcode_e;

class Transaction;
  rand opcode_e opcode;
  rand byte operand1;
  rand byte operand2;
endclass

Transaction tr;
```

2. Expand the solution to Exercise 1 to cover the test plan requirement, "Operand1 shall take on the values maximum negative (−128), zero, and maximum positive (127)." Define a coverage bin for each of these values as well as a default bin. Label the coverpoint `operand1_cp`.
3. Expand the solution to Exercise 2 to cover the following test plan requirements:
   a. "The opcode shall take on the values ADD or SUB" (hint: this is 1 coverage bin).
   b. "The opcode shall take on the values ADD followed by SUB" (hint: this is a second coverage bin).
   Label the coverpoint `opcode_cp`.
4. Expand the solution to Exercise 3 to cover the test plan requirement, "Opcode must not equal DIV" (hint: report an error using `illegal_bins`).
5. Expand the solution to Exercise 4 to collect coverage on the test plan requirement, "The opcode shall take on the values ADD or SUB when operand1 is maximum negative or maximum positive value." Weight the cross coverage by 5.
6. Assuming that your covergroup is called `Covcode` and the instantiation name of the covergroup is `ck`, expand Exercise 4 to:
   a. Display the coverage of coverpoint `operand1_cp` referenced by the instantiation name.
   b. Display the coverage of coverpoint `opcode_cp` referenced by the covergroup name.