example, suppose that a cache controller performs `behavior A` when there is a cache hit (*e.g., fetch data from the cache*), or performs `behavior B` when there is a cache miss (e.g., *invalidate cache entry, fetch data from main memory through an interface, store data into the cache, supply the needed value*). To simplify the problem for the formal verification tool, one could constrain or "restrict" the problem to one of the two cache modes. Thus, the following can be written in one set of scenarios:

```
restrict property (@(posedge clk) cache_hit == 1'b0);   // cache miss scenario
```

Another formal verification session could use this restriction:

```
restrict property (@(posedge clk) cache_hit == 1'b1);   // cache hit scenario
```

Another example of a **`restrict`** statement is the *reset* signal that can be classified as "one-shot" (i.e., one-time) that has a specific behavior during initialization. In this case, one needs to restrict the signal `reset_n` to low for 1 to 100 cycles, and then to high forever. Thus,:

```
initial
ap_reset_then_hi : assume property ( @ (posedge clk)
     !reset_n[*1:100] ##1 reset_n |=>  always (reset_n));
```

This is a good use of the **`restrict`** because it is never illegal for `reset_n` to go active, but it is a common scenario to limit possible scenarios.

In formal tools, you can typically re-qualify each assertion as **`assume`**, **`assert`**, or **`cover`** because the role of some assertion in a whole system may actually be different depending on what "part" of the system you are currently examining with formal. However, there is no need for the same re-qualification for the **`restrict`** statements - they mean one and only one thing, and the **`restrict`** should not be used as an **`assert`** or **`assume`** statement..

Guideline: Use **`assume`** to define legal input states that prevent false failures. Use **`restrict`** in formal verification when you are just trying to reduce the state space by limiting a test to one of several legal scenarios.

4.5.1.4 cover statement

There exist three categories of cover statements, **`cover sequence`** and **`cover property,`** immediate **`cover`**. The **`cover sequence`** statement specifies sequence coverage, while the **`cover property`** statement specifies property coverage. The syntax for the **`cover`** statement is:

```
cover_property_statement::=
   cover property (property_spec) statement_or_null

cover_sequence_statement::=
   cover sequence (
    [clocking_event] [disable iff (expression_or_dist)]
                     sequence_expr) statement_or_null
simple_immediate_cover_statement ::=
   cover ( expression ) statement_or_null
deferred_immediate_cover_statement ::=
   cover #0 ( expression ) statement_or_null
 | cover final ( expression ) statement_or_null
```

statement_or_null is executed every time a property attempt succeeds nonvacuously

statement_or_null is executed every time a sequence thread succeeds. Use **`first_match`** (seq_expression) if interested in one match

Note: During simulation, it is possible to detect that a property is covered by querying the **`vpi_get()`** function with the **`vpiAssertSuccessCovered`** argument (see 6.2.7.2). For simulation efficiency, one can then turn off the *assertion_identifier* for the coverage (e.g., **`$assertoff`**`(1, cp_q_abc)`) when coverage is reached.

Rule: The pass statement in the action block must not include any concurrent **`assert`**, **`assume`** or **`cover`** statement. Coverage results are divided into two: coverage for properties, coverage for sequences.

[1] The results of coverage statement for a property contain:

```
default clocking @(posedge clk); endclocking
cp_ab: cover property(a |=> b);
```

- *Number of times attempted*
- *Number of times succeeded*
- *Number of times succeeded because of vacuity*

The following files represent an example of the cover property and the resulting statistics: *ch4/implication.sv*, *implication.bmp, implication.jpg, implication_assertion_report.txt, implication_fcover_report.txt*.

[1] Results of coverage for a sequence include:

```
cq_ab: cover sequence(a ##1 b);
```

- *Number of times attempted*
- *Number of times matched (each attempt can generate multiple matches)* (see 1.3.2 and 2.3.2). *In addition, statement_or_null gets executed for every match. If there are multiple matches at the same time, the statement gets executed multiple times, one for each match.*

[1] The immediate **cover** *statement specifies that successful evaluation of its expression is a coverage goal.*
The results of coverage for an immediate **cover** *statement shall contain the following:*

```
c_ab: cover (a && b);
```

- *Number of times evaluated*
- *Number of times succeeded*

A pass statement for an immediate **cover** *may be specified in statement_or_null. The pass statement shall*
be executed if the expression evaluates to true. The pass statement shall be enabled to execute immediately
after the evaluation of the expression of the immediate **cover**.

4.5.1.4.1 Understanding coverage

Difference between assertion and coverage?

Assertion and coverage have different significance in simulation versus formal verification (See Chapter 7).

- Assertion in simulation: In simulation an assertion states that under the set of stimuli provided by the testbench the underlying property must hold; if there is a test sequence that causes the assertion to fail, then an error message is provided. If a test sequence causes the assertion to succeed, a count that keeps tracks of those successes.
- Assertion in formal verification : In formal verification an assertion states says that all combinations of all inputs and sequences do not cause a failure of the underlying property.
- Coverage in simulation: In simulation, coverage provides feedback information about how well the design was exercised with sequences of inputs needed to verify that the design meets the requirements, which can be verified with assertions or supporting logic. Coverage can also be used to measure how well the design reacts to the stimulus environment; specifically, coverage can be used to measure latencies from inputs to points of interest in the design (e.g., outputs or internal states and using delay ranges in the cover, e.g., `cover sequence(a ##[10:15] b);`).
- Coverage in formal verification: Cover property simply states that there exists ONE way to satisfy the property – in other words it is reachable, it is not DEAD code.

Should all threads of an input sequence be covered?

The quick answer is "not necessarily"; in most cases, checking for just a successful match of a thread is sufficient because that is often difficult to reach. An acceptable alternative is to check for the boundaries (or corner cases). Consider a requirement with the following assertion where `req` and `rdy` are inputs and `ack` is an output:

```
default clocking cb_clk @ (posedge clk);  endclocking
ap_req_rdy_ack: assert property(
      first_match($rose(req) ##[1:5] rdy) |-> ##[1:2] ack);
```

In this example, the input sequence is specified by the requirements and consists of five possible threads: **($rose**(req) #1 rdy, **$rose**(req) #2 rdy, .., **$rose**(req) #5 rdy). The latency between req input and the ack output ranges between 2 to 7, and between a successful antecedent to the output of 1 to 2. The question then becomes: do we need to verify that the test environment subjects the design to all possible input threads for this assertion, or is it sufficient to ensure that just a successful antecedent thread is sufficient, or can just the boundaries in the delays be verified (in this case **($rose**(req) #1 rdy) and **($rose**(req) #5 rdy)? As a general case, not all threads of an input sequence need to be covered if the following conditions are true:

- The generation of the sequence is controlled within the specified range; for example:

```
class C;
    rand byte dly; // cycle delay between req and rdy
    constraint req2rdy_cst { dly > 0 && dly <= 5 ;}
endclass : C
```

- The design is not sensitive to this constraint; for example, there is no timeout clock that measures this delay and reacts accordingly. Thus, this constraint on the input sequence is more of a system requirement that is not checked in the RTL implementation.
- The FSM implementation for this requirement is autonomous, meaning that the RTL has a single FSM that waits on the occurrence of a signal before proceeding to the next state (such as waiting for the rdy signal).

Now consider a requirement with the same assertion ap_req_rdy_ack, where req, rdy, and ack are internal to a design, and are not input requirements as they emanate from independent FSMs. As a general case, all threads, or at least the boundaries of the input sequence delays, need to be covered if any of the following conditions are true:

- The design is sensitive to the constraints of the antecedent; for example, there is a timeout clock that measures this delay between req and rdy, and the FSM reacts accordingly.
- The FSM implementation for this requirement is generated with multiple coupled FSMs; thus the design may be sensitive to the cycle delays in the antecedent (such the delay between req and rdy signal).

Note: Typically, the implementation is not known or need not be considered in defining coverage. What is important are the requirements and a compromise between depth of coverage and simulation performance.

Which cover statement is needed: `cover property` or `cover sequence`?

The **cover** statement is similar to the **assert** statement with the main exception that the cover statement does not fail when the property is false, and thus does not report errors. Simulation tools typically provide coverage on **assert property** by reporting the number of successes and failures; thus, from a coverage perspective there is less of a need to ever use **cover property**. This is very useful from a performance perspective because it is not necessary to duplicate **assert property** and **cover property** statements. However, the coverage reporting can be misleading, regardless of the source of the reporting (i.e., from the **cover** or the **assert** statement). This is because a reported high coverage may not necessarily mean that all threads of the test sequences needed to verify the DUT (per the assertions) were indeed exercised. The example discussed previously can have the following coverage statements:

`cp_req_rdy: cover property(`
`  $rose(req) ##[1:5] rdy);`
`cq_req_rdy: cover sequence (`
`  $rose(req) ##[1:5] rdy);`
	Applicable when any match of the sequence is sufficient for verification.

`cq_req1_rdy: cover sequence ($rose(req) ##1 rdy);`
`cq_req2_rdy: cover sequence ($rose(req) ##2 rdy);`
`cq_req3_rdy: cover sequence ($rose(req) ##3 rdy);`
`cq_req4_rdy: cover sequence ($rose(req) ##4 rdy);`
`cq_req5_rdy: cover sequence ($rose(req) ##5 rdy);`
	Applicable when all matches of the sequence are needed for verification; thus all threads need to be checked for their occurrences.

Note: For the `cq_req_rdy` sequence coverage, a tool will identify *number of times attempted* and the *number of times matched,* but it will not identify which sequence matched or did not match. This is why it is necessary to write (or generate) a coverage for each delay. This can be done with the generate statement, as shown below, instead of the individual **`cover sequence`** statements.

```
generate for (genvar i=1; i<=5; i++)
        cq_aib: cover sequence ($rose(req) ##i rdy);
endgenerate
```

4.5.1.4.2 Using covergroup for data coverage

An alternative to measuring the input sequences is to use the covergroup with bins. This methodology is fairly tedious and is demonstrated in file ***ch4/4.5/bining.sv, bining.jpg***. Consider the following property that has range delays:

```
property l2_cache(N,M);
    int v_a;
    @(posedge clk) (c_miss, v_a = c_a) |-> (##[N:M] mm_rd && m_a==v_a);
endproperty
cp_l2_cache: cover property (l2_cache(2,10));
```

In this subsystem, a cache miss (`c_miss`) at the cache address (`c_a`) must be followed in `N` to `M` cycles by a memory read (`mm_rd`) at the memory address (m_a) that corresponds to the initial cache miss address (i.e., the original c_a). The `cp_l2_cache` cover property described above does what is intended. However, the coverage result would only identify how many times that property was covered, but it would not identify how many of the ranges 2 to 10 were covered. To provide more details, binning of that range using a **`cover property`** can be used. The key elements of this methodology include:

1. The declaration of an integer-like variable to be used for **`coverpoint:`**
   ```
   bit[3:0] l2_cache_miss_delay;
   ```

2. Definition of a sequence that updates the value of the coverpoint variable based on the number of cycles necessary to complete the sequence:
   ```
   sequence event_after_range_shift_bin_sample(N,M,e);  // ch4/4.5/bining.sv
    int m_delay = 0;
    @(posedge clk) ##N (!(e) , ++m_delay) [*0:M-N]
        ##1 (e , temp_bin_sample(N+m_delay) );
   endsequence
   ```

   Function call to update the covergroup variable (`l2_cache_miss_delay`) and then sampling of this covergoup (t_cg).

```
// -------------------- cover with temporal binning applied -----
   property l2_cache_bin_sample(N,M);
       int v_a;
       @(posedge clk) (c_miss, v_a = c_a)
           |->
           event_after_range_shift_bin_sample(N,M,(mm_rd && m_a==v_a));
   endproperty
   cp_cache: cover property (l2_cache_bin_sample(2,10));
```

3. Declaration and instantiation of a covergroup and the sampling for the covergroup.

```
covergroup temp_cg;
 type_option.merge_instances = 0;
 option.per_instance = 1;
 option.get_inst_coverage = 1;
 coverpoint l2_cache_miss_delay;
 endgroup
 temp_cg t_cg = new;   // instantiation of covergroup
function void temp_bin_sample(int M);
     l2_cache_miss_delay = M;   // update of covergroup variable
     t_cg.sample();   // Sampling of covergoup
 endfunction
```

Guideline: If it is necessary to ensure that coverage of separate threads are performed in a simulation write separate **cover sequence** statements for those sequences; a generate statement may be useful (Section 4.5.1.4.1). Relying on a property or cover statement of a multi-threaded property can lead to misleading coverage reporting, as explained above. Another option is to use a **covergroup** to measure the various covered delays, but this approach requires more supporting code and might be labor and simulation intensive.

4.5.1.5 Expect construct

Rule: The **expect** construct is not part of the "verification layer" because it does not make a statement about what should be done with a property in terms of verification. However, the **expect** statement makes use of a property. Specifically, *[1]* The **expect** statement can appear anywhere a **wait** statement can appear (e.g., **always** procedure, **task** (but not in classes!!!)). *The* **expect** *statement is a procedural blocking statement that allows waiting on a property evaluation. The* **expect** *statement accepts the same syntax used to assert a property.*

```
expect_property_statement ::=
       expect ( property_spec ) action_block
```

An **expect** *statement causes the executing process to block until the given property succeeds or fails. The statement following the* **expect** *is scheduled to execute after processing the Observed region in which the property completes its evaluation.*

When the property succeeds or fails, the process unblocks, and the property stops being evaluated (i.e., no property evaluation is started until that **expect** *statement is executed again). When executed, the* **expect** *statement starts a single thread of evaluation for the given property on the subsequent clocking event, that is, the first evaluation shall take place on the next clocking event. If the property fails at its clocking event, the optional* **else** *clause of the action block is executed. If the property succeeds, the optional pass statement of the action block is executed. The execution of pass and fail statements can be controlled by using assertion action control tasks.*

Thus, the **expect** statement is a blocking statement that includes inline a property, and an action is executed based on the result of the evaluation of the property. Because it is a blocking statement, the property can refer to **automatic** as well as **static** variables. For example, the task below waits between 1 and 10 clock ticks for the variable `data` to equal a particular value, which is specified