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