2.3.3.4 [*n : $], [*] [+] Repetition range with infinity

🕮 **Rule:** The `[*n:$]` constructs is similar to the `[*n:m]` construct, except that `m` is infinite, instead of being bounded by an integer number, and `n` means that the repetition applies for a minimum of `n` cycles. The `[*]` is an abbreviation for `[*0:$]`. The `[+]` is an abbreviation for `[*1:$]`.

2.3.4 *goto* repetition, Boolean ([->n], [|->n:m])

🕮 **Rule**: The *goto* repetition operator (`Boolean[->n]`) allows a Boolean expression (and not a sequence) to be repeated in either consecutive or non-consecutive cycles, but the Boolean expression must hold on the last cycle of the expression being repeated. The number of repetitions can be a fixed constant or a fixed range.

Note: **`b [->m]` is equivalent to `( !b [*0:$] ##1 b)[*m]`**

`b[->1]` is equivalent to:
`!b[*0:$] ##1 b`

`b[->2]` is equivalent to:
`!b[*0:$] ##1 b ##1 !b[*0:$] ##1 b`

For example:

```
(a) |=>  (b[->2] ##1 c);
```

That above property states that if `a` then, starting from next cycle, there should be two occurrences of `b`, which can be consecutive or non-consecutive. However, `c` must occur in the cycle following the last occurrence of `b`. The following example meets an assertion of the above property.

```
cycle     1 2 3 4 5 6 7 8 9     --   (a) |=>  (b[->2] ##1 c));
a         0 1 0 0 0 0 0 0 0
b         - - 0 1 0 0 1 - -
c         - - - - - - - 1 -     // Succeeds

cycle     1 2 3 4 5 6 7 8 9     --   (a) |=>  (b[->2] ##1 c));
a         0 1 0 0 0 0 0 0 0
b         - - 0 1 0 0 1 - -
c         - - - - - - - 0 -     // Fails
```

Consider the following property expression:

```
a ##1 b[->1] |-> c; // This is equivalent
a ##1 !b[*0:$] ##1 b ] |-> c;
```

In this case, for uniqueness, a better approach would be to use an edge detect function on `a`. A **`first_match`** is not needed in this case because once `b==1`, there could not be another thread of `!b`.

🕭 **Guideline**:

1) Do not use a **`first_match`** function in a sequence that is an antecedent that has a *goto* repetition operator unless that sequence has other expressions that are multi-ranged (with a delay or a repeat). Thus,

```
a ##1 b[->1] |-> c;   // ✓ first_match is not needed here
first_match(a ##1 b[->1]) |-> c; // 👎👎 // first_match not needed
first_match(a[*1:2] ##1 b[->1]) |-> c; // ✓ first_match needed
```

2) Avoid using the *goto* repetition operator on a first term of an antecedent. That creates unnecessary computations because every false value of the first term starts a new attempt, instead of making the assertion vacuous.

```
a[->1] |-> b; // 👎👎   is equivalent to
!a[*0:$] ##1 a | -> b; // Successful attempt at every clocking event
$rose(a) |-> b;  // ✓✓ (Need to evaluate the applicability of the $rose)
```

3) Avoid consequents that can never fail, and consider rewriting in a style where it can fail. For example, in the following property statement specification, if `a` occurs and then `b` is not followed by `c`, then another search for a successful consequent is initiated.

```
$rose(a) |-> ##[0:$] b ##1 c ;  // is same as
$rose(a) |-> (##0 b ##1 c) or (##1 b ##1 c) or (##2 b ##1 c) or ..(##n b ##1 c);
```
// where n is infinity, and "or" is the sequence or operator.

Instead of the above, use the following property statement specification because it can fail:

```
$rose(a) |-> b[->1] ##1 c;  // The  first occurrence of b and then 1 cycle later c.
                            // That can fail if c is false after b.
```

4) Qualify consequents with the strong operator if they may never reach success. This may occur if not enough test cases were exercised. Thus,

```
$rose(a) |-> strong(b[->1] ##1 c);  // will fail if b==1'b0 during the simulation
```

2.3.5 Non-consecutive repetition, Boolean([=n], [|=n:m])

Rule: The sequence non-consecutive repetition operator (`[=n]`) is similar to the consecutive repetition operator, in that the Boolean expression is repeated in either consecutive or non-consecutive cycles. However, the non-consecutive repetition operator imposes no restriction on the relationship between the last cycle of the Boolean expression and the next expression. Thus, there are exactly *n* (or a range of) repetitions in a path of length *k*, where *k* can be greater than *n*. The number of repetitions can be a fixed constant or a fixed range. The item being repeated must be a Boolean, it cannot be a sequence.

Note: **`b[=m] is equivalent to  ( b [->m] ##1 !b [*0:$] )`**

For example:

```
property pREPT_EQUAL;
      (a) |=>  (b[=2] ##1 c);
endproperty : pREPT_EQUAL
```

```
b[=1] is equivalent to:
!b[*0:$] ##1 b ##1 !b[*0:$]
b[=2] is equivalent to:
!b[*0:$] ##1 b ##1 !b[*0:$] ##1 b ##1 !b[*0:$]
```

That property states that if `a` then, starting from next cycle, there should be two occurrences of `b` either consecutive or non-consecutive, and then some time later (i.e., next cycle or later) `c` occurs. The following examples satisfy the above property.

```
cycle    1 2 3 4 5 6 7 8 9 A   (a) |=>  (b[=2] ##1 c));
a        0 1 - - - - - - - -
b        - - 0 1 0 0 1 0 0  -      path of length k==10 has n==2 repetitions
c        - - - - - - - - 0 0 1

cycle    1 2 3 4 5 6 7 8 9  (a) |=>  (b[=2] ##1 c));
a        0 1 - - - - - - - -
b        - - 0 1 0 0 1 0 1       // A 3rd b without a c
c        - - - - - - - - 0 0     //  Fails
```

Note: if after the second occurrence of `b` a new `b` occurs (and NO `c`), the property fails. However, if after the second occurrence of `b`, `c` occurs anytime before a new `b`, or at the same time as the last `b`, then the property succeeds.

Guideline:

1) In an antecedent, DO USE a **`first_match`** function in a sequence with a sequence non-consecutive repetition operator. This guideline is because this operator can cause multiple threads. For example:

```
a ##1 b[=1] ##1 c |-> d;  //  is equivalent
a ##1 !b[*0:$] ##1 b ##1 !b[*0:$] ##1 c |-> d ;
```
// The (!b[*0:$] ##1 c) *can cause multiple threads*

```
first_match($rose(a) ##1 b[=1] ##1 c) |-> d; //
```
(evaluate applicability of **`$rose`**)

2) Avoid using the sequence non-consecutive repetition operator on a first term of an antecedent. That creates unnecessary computations because every false value of the first term starts a new attempt, instead of making the assertion vacuous. This can be a large performance overhead for a simulator to track, and it makes the coverage statistics less meaningful.

```
b[=1] ##1 c |-> d;   // 👎👎  is equivalent

!b[*0:$] ##1 b ##1 !b[*0:$] ##1 c |-> d ; // 👎👎  antecedent is multi-threaded
first_match($rose(b) ##1 !b[*0:$] ##1 c) |-> d;  // ✓✓
                // (Evaluate the applicability of the $rose)
```

2.4 Sequence composition operators

Sequences are one of the more powerful features of SystemVerilog Assertion. Basic protocol verifiers can easily be built using these sequences. However, complex protocols/scenarios require the ability to describe or specify interactions between sequences, such as choice or inclusion of sequences, and the use of local variables that are exclusively reserved for each attempt of the sequence. SystemVerilog provides sequence composition operators to combine individual sequences in a variety of ways that enhance code writing and readability. The sequence composition operators specify the relationships between sequences. These operators include:

- **`##` : sequence concatenation**
  - **`##0` sequence overlapping concatenation (a.k.a. fusion).** The `##0` constructs a sequence in which two sequences overlap by one cycle. Note: the ##0 operator fuses sequences, meaning that the last term of the left sequence is immediately (in the same cycle) followed by the first term of the right sequence. If those sequences are Boolean, many users use the logical `&&` operator instead. However, though equivalent in functionality, the `"Boolean_expression && Boolean_expression"` has a different connotation than the `"Boolean_expression ##0 Boolean_expression"`; the `&&` operator has a combinational logic inference, whereas the `##0` has a sequential inference (e.g., occurring after). Consider the following example where a new "`a`" must be repeated "`g`" times, where "`g`" is a module variable of type **`int`**:

```
property p_g; //
   int v;     //
   ($rose(a),v=g)|->(a==1'b1, v=v-1'b1)[*1:$] ##0 v==1'b0 ##0 a==1'b1;
endproperty
```

To assure a possible pass, consider using the first_match() operator, specifically:

```
 ($rose(a),v=g)|-> first_match((a==1'b1, v=v1'b1)[*1:$] ##0 v==1'b0)
                   ##0 a==1'b1;
```

  - **`##n` fixed cycle delay.** The next subsequence follows the preceding one after *n* cycles (e.g., `a ##2 b`); this is equivalent to (`a ##1 1'b1 ##1 b`).
  - **`##[m:n]` range cycle delay**. The next subsequence follows the preceding one after a minimum of *m* and a maximum of *n* cycles until the first element of the next subsequence is true (e.g., `a ##[2:5]b`). A range cycle delay can start multiple threads;

    `(a ##[2:5] b)` is equivalent to
    `(a ##2 b) or (a ##3 b) or (a ##4 b) or (a ##5 b).`

- **`or` : sequence ORing (a.k.a. disjunction).** The sequence **`or`** operator constructs a composite sequence that holds true if either of the two operand sequences hold at the current cycle. A sequence **`or`** operator between two sequences starts two concurrent threads, one for each sequence; each sequence in turn can be mulit-threaded.