Non-consecutive	Boolean[= n] sequence[=n] 💣	**a ##1 b[=2] ##1 c // a !b b !b !b !b b !b !b c satisfies sequence** // *a* holds, then any 2, not necessarily consecutive occurrences of b, and then any cycle later, *c* is true. When *c* is 1, b must be 0. (a **##**2)[=3] // 💣 is Illegal
goto repetition	Boolean[-> n] sequence[->n]💣	**a##1 b[->2] ##1 c // a !b !b b !b !b !b b c satisfies sequence** // *a* holds, then any 2 not necessarily consecutive *b* occurrences, and then at next cycle, *c* is true . (a **##**2 b)[->3] // 💣 is Illegal

2.3.1 Attempt / thread difference

Important !!!

It is very important to understand the difference between attempts and threads, as those are often used interchangeably, but really represent different concepts.[23] This difference is particularly emphasized in assertions where sequences are multi-ranged (with delay or repetition operators) and used as antecedents.

[1] A property may involve the checking of one or more sequential behaviors beginning at various times. An attempted evaluation of a sequence is a search for a match of the sequence beginning at a particular clock tick (called the leading clocking event). To determine whether such a match exists, appropriate Boolean expressions are evaluated beginning at the particular clock tick and continuing at each successive clock tick until either a match is found or it is deduced that no match can exist. Consider the following sequence and assertion:

```
sequence qAB; @ (posedge clk) a[*1:2]##1 b; endsequence : qAB
ap_qAB: assert property(qAB); // q_AB is a sequence processed as a property
                              // This is used to demonstrate concepts.
                              // sequences are rarely used as a property in an assertion
```

The sequence `qAB` is equivalent to `((a ##1 b) or (a ##1 a ##1 b))`. This is because whenever a range is defined, for either a repetition or delay range, that range specifies alternative search paths for success or failure. In an assertion statement, an attempt starts at each leading clocking event and spans the evaluation of the property. For the assertion `ap_qAB`, at every (`posedge clk`) clock tick there is ONE attempt started at that leading clocking event to evaluate that sequence. For each attempt in the evaluation of that sequence, there are two possible "ways" or "paths" to search for a match (*i.e., success of the sequence*), specifically the sequence "`a b`" or the sequence "`a a b`". These "ways" are called "threads". Thus, a thread is a search path for success within an evaluation attempt that starts at the leading clocking event of the property. Sequences can be single-threaded, or be multi-threaded.

- **Single-thread sequence**: A sequence is single-threaded if it has single search path. A single search path exists if a sequence does NOT include any of the following operators: **`and`**, **`intersect`**, **`or`**, range delay (*i.e., `##[1:5]`*), consecutive range repetition (i.e., *`a[*1:5]`*) , non-consecutive repetition (*i.e., `a[=3]`, `a[=1:3]`*). An example of a single-threaded sequence is: `a ##2 b ##1 c[->1]`

- **Multi-thread sequence**: A sequence is multi-threaded if it includes any of the following operators: **`and`**, **`intersect`**, **`or`**, range delay (*e.g., `a ##[1:2] b`*), consecutive range repetition (*e.g., `a[*1:2] ##1 b`*), non-consecutive repetition (*e.g., `a[=2] ##1 b`*).

[23] This topic was briefly introduced in Section 1.4.

Table 2.3.1 provides examples of single and multi-threaded sequences. Section 2.4 addresses the sequence composition operators (*e.g.*, ***and***, ***intersect***, ***or***).

Table 2.3.1 Examples of single and multi-thread sequences

Thread type	Example	Number of threads and comments	Equivalency
Single-thread	`a ##1 b ##3 c`	1 The thread starts at each attempt.	`a ##1 b ##3 c`
Multi-thread with **and**	`(a ##1 b ##3 c) and (d ##3 e ##5 f)`	2. The LHS and RHS sequences start at the first attempt but may end at different cycles.	`(a ##1 b ##3 c) and (d ##3 e ##5 f)`
Multi-thread with **intersect**	`(a ##1 b ##3 c) intersect (d ##3 e ##5 f)`	2. The LHS and RHS sequences start at the first attempt but must end at the same cycle.	`(a ##1 b ##3 c) intersect (d ##3 e ##5 f)`
Multi-thread with **or**	`(a ##1 b ##3 c) or (d ##3 e ##5 f)`	2. The LHS and RHS sequences start at the first attempt but may end at different cycles.	`(a ##1 b ##3 c) or (d ##3 e ##5 f)`
Multi-thread with range delay	`(a ##[1:3] b )`	3. At attempt, if `a==1` then there are three possible paths for the sequence to complete successfully because after `a==1, b` must be true in one to three cycles.	`(a ##1 b) or (a ##2 b) or (a ##3 b)`
Multi-thread with consecutive range repetition range	`(a ##1 b[*1:3])`	3. At attempt, if `a==1` then there are three possible paths for the sequence to complete successfully because after `a==1, b` must be repeated one to three times.	`(a ##1 b[*1]) or (a ##1 b[*2]) or (a ##1 b[*3])`
Multi-thread with non-consecutive range repetition range	`(a ##1 b[=2] ##1 c)`	*n.* At attempt, if `a==1` then if there are *n* cycles between the 2nd `b` and `c` then there are *n* paths in this sequence. *n* is greater than 0. If *n* is 0 or 1, then there is one path in this sequence.	`(a ##1 !b[*0:$] ##1 b ##1 !b[*0:$] ##1 b ##1 !b[*0:$] ##1 c)`

2.3.1.1 Important concepts on threads and sequences

Important !!!

- When a thread is evaluated, its outcome is either a *match* or *not a match* (also referred to as *non-match*). It is incorrect to address the evaluation of a thread as a pass/success, or failure; this is because a thread is not an assertion. A *match* has a value of *true*.

- The evaluation of a sequence is *true* or *false*; sometimes it is also referred to as a *match* or no *match*. Unlike the evaluation of a property, there is no vacuity in the evaluation of a sequence; in other words, a property that is only a sequence is always nonvacuous. For example, consider the following assertions:

```
default clocking cb_clk @ (posedge clk);  endclocking
ap_always_non_vacuous: assert property(a ##2 b); // property is a sequence.
ap_non_vacuous_if_c_is_1: assert property(   // property has an implication operator
  c |-> a ##2 b); // nonvacuous if c==1 because a ##2 b is a property that is a sequence
    // and is thus nonvacuous.  It is vacuous if c==0.
```

- A multi-threaded sequence may have several matches, with one match per thread.
  For example, in the sequence *(a ##1 b[*1:3] ##1 c)* all three possible threads may succeed if `a==1` at its attempt (cycle 1), and `b==1` at cycles 2, 3, 4, and `c==1` at cycles 3, 4, 5.

If a multi-threaded sequence is used in an antecedent (e.g., `a ##1 b[*1:3] ##1 c` |-> d), then all threads must be tested with its consequent for the assertion to terminate as successful.

- A multi-threaded sequence may be terminated upon a first match.
  An implicit first match (i.*e., a successful sequence evaluation*) occurs when the sequence is used as a property, or when the sequence is used as a consequent of a property (see 1.4.3) because a first match of the sequence completes the evaluation of the property. For example, the property (`w |-> a ##1 b[*1:3] ##1 c`) terminates successfully if `w` and `a` are true at cycle `1`, and `b=true` at cycle 2, and `c` is true at cycle `3`.

- The explicit **`first_match`** function will also terminate a multi-threaded sequence upon a match. (e.g., **`first_match`** `(a ##1 b[*1:3] ##1 c)` |-> d) A **`first_match`** function is not needed when the implicit **`first_match`** covers that aspect (*with the **`cover`** statement*). It generally is needed in a multi-threaded sequence that is an antecedent.

2.3.2 Impact of multi-threaded sequences in assertions

An assertion with a multi-threaded sequence is processed differently if that sequence is used as an antecedent rather than a consequent of the property. The following subsections explain that difference with examples. Section 2.5.1 addresses the topic of the **`first_match`** operator that resolves some of the issues of possible unexpected results with multi-threaded antecedents.

Consider the simple following requirement: "If a subsystem has a `bus_request` followed in `1` to `3` cycles later by an `acknowledge`, then a `done_data` transfer should occur within `5` to `10` cycles." Such a requirement can be poorly expressed using the following assertion:

```
ap_req_done: assert property(  //👎👎
        $rose(bus_request) ##[1:3] acknowledge |-> ##[1:2] done_data);
```

Because the antecedent has a range delay, the assertion `ap_req_done` is equivalent to the following assertion with no range delays:

```
ap_req_done_equivalnt: assert property(
        ($rose(bus_request) ##1 acknowledge) or
        ($rose(bus_request) ##2 acknowledge) or
        ($rose(bus_request) ##3 acknowledge) |->
                (##1 done_data) or (##2 done_data) );
```

In the `ap_req_done` assertion, the antecedent `is $rose(bus_request) ##[1:3] acknowledge`. At every clocking event (*called the leading clocking event*) the simulator tests for a successful attempt of **`$rose`**`(bus_request)`. If there is no new `bus_request` then this assertion has little significance, and is considered vacuous. However, if there is a successful **`$rose`**`(bus_request)` then three concurrent assertion threads are started:[24]

1. One thread with the `acknowledge` occurring one cycle later followed by the evaluation of the consequent, i.e., **`$rose`**`(bus_request) ##1 acknowledge |-> ##[1:2] done_data.`
2. Another thread with the `acknowledge` occurring `2` cycles later followed by the evaluation of the consequent, i.e., **`$rose`**`(bus_request) #2 acknowledge |-> ##[1:2] done_data.`
3. Another thread with the `acknowledge` occurring `3` cycles' later followed by the evaluation of the consequent, i.e., **`$rose`**`(bus_request) #3 acknowledge |-> ##[1:2] done_data.`

Each one of those threads is evaluated to determine the outcome; that outcome can be vacuous, pass, or fai*l*.

[24] A thread is a path that represents the possible states of a sequence or a property. A sequence by itself may have multiple paths; however, when a multi-threaded sequence is used as an antecedent to a property, then that creates multiple paths, with each one extending from the antecedent to the consequent.

🕮 **RULE:** An assertion started with a multi-threaded sequence has the following criteria for its outcome:

- All threads must be exercised in search of a matched antecedent /consequent pair, unless the assertion fails. In that case, the search is short-circuited (*i.e., stopped*).
  - The assertion is considered failed if there is a matched antecedent with a failed consequent.
- For the assertion to be successful, two requirements must be met:
  1. There must be no failure in the search of a matched antecedent with a consequent.
  2. All threads of the antecedent and consequent must be successful. There are two types of successes: vacuous and nonvacuous.
     a. VACUOUS SUCCESS:
        i. If all the threads of the antecedent result in a no match, then the assertion is considered vacuously true (*also referred to as vacuous success, or simply vacuous*).
        ii. Vacuity can also occur if a thread has a matched (*i.e., true*) antecedent but its consequent is vacuous (see 3.9 for discussion of vacuous properties).
     b. PASS / NON-VACUOUS: For a nonvacuous pass (*or success*), there must be at least one matched antecedent with its successful nonvacuous property, which represents the consequent. There must also be no failures in the consequent. Some threads may however be vacuous.

Consider the following example:

```
apAB:  assert property (a[*1:2] ##1 b |-> c); // with default clocking
```

At each leading clocking event, there is an attempt of the evaluation of the antecedent sequence. If the start of the attempt is true (i.e., `a==1`), two threads are started and the attempt is further evaluated. Otherwise (i.e., `a==0`), the attempt is vacuously true because the antecedent is false (*and all threads will thus be vacuous*). The simulation of that model and the identification of attempts and threads are annotated in Figure 2.3.2-1[25]

An assertion may 1) fail, 2) pass, 3) succeed vacuously, 4) fail vacuously, or 5) be disabled. Tools consider any failure, vacuous or nonvacuous, as a failure, and do not make any distinction between the two. Vacuous failure is implicitly addressed in 1800-2012, but it is discussed in Section 3.1 and in Appendix B.

🕭 **Guideline:** 1) When defining sequences that are used as antecedents, insure uniqueness in its triggering condition to avoid unexpected results, such as assertions that can never succeed or fail in unintended situations. For example, if only a new occurrence of a signal is needed to start the evaluation of an assertion use of an edge detect (e.g., **`$rose(signal)`**) of the first element of the sequence or a single trigger pulse condition, such as a one clock enable signal.

---

[25] For the benefit of the reader, a simulation of the following assertion is also provided in files ***ch2/s1c_ep.sv*** , ***ch2/2.3//ap_REP_all_threads_for_success_wave.bmp,*** **and** ***ch2/2.3/ap_REP_all_threads_for_success_thv.jpg***

```
ap_REP_all_threads_for_success: assert property(
            @ (posedge clk)  $rose(a) ##1 b[*2:5] |=> c);
```

2) Use of the **first_match** operator if the sequence is multi-threaded.

An assertion of a property that has an antecedent with an infinite range, and without a **first_match** operator, will never succeed, but can fail (a **first_match** ignores the other matches). Consider the following example:

```
ap_never_succeed_multithreaded_and_not_unique: assert property(
                                                a ##[2:$] b |=> c); // 👎👎
 // See first bulleted rule in this section.
ap_can_succeed_and_unique: assert property(
   first_match($rose(a) ##[2:$] b) |=> c); // ✓
```

An antecedent with an infinite range (delay range or repetition range) will initiate an infinite number of threads. For the assertion to not fail, all threads of the antecedent and consequent must be evaluated. Since there are an infinite number of threads to be tested, the assertion can never successfully terminate. However, a failure of a matched antecedent / consequent pair will cause the assertion to fail.

Files ***Ch2/2.3/s1a2.sv, s1a2_wave.bmp, S1A_ATV.jpg S1A2_ATV.jpg S1A_ATV2.jpg*** show the simulation and thread viewer results for the following assertions:

```
ap_never_succeed: assert property(@ (posedge clk)
      $rose(a) ##[2:$] b |=> c);

ap_all_threads_for_success: assert property(@ (posedge clk)
      $rose(a) ##[2:5] b |=> c);
```

The waveforms and thread viewer views are *QuestaSim* screen shot, Courtesy of *Mentor Graphics*.

2.3.2.1 Multi-threaded sequence in consequent

📖 **Rule:** If an assertion of a property has a multi-threaded sequence used as a consequent, any match of the consequent will terminate a thread of a matching antecedent. The assertion is considered a failure if the antecedent matches but after all possible threads of the consequent were exercised, and none of them had a match.

Thus, if the antecedent has a single thread and the consequent is multi-threaded, then the assertion of this property will succeed when the antecedent is a match and the consequent reaches a first match Consider the following assertion: *Ch2/an_conseq_matches.sv*, *an_conseq_matches.jpg*

```
ap_consequent_any_match: assert property(@(posedge clk)$rose(a)|-> ##[1:5] b);
// Equivalent to:
 ap_consequent_any_match_eq: assert property(@(posedge clk)
      $rose(a)|-> (##1 b) or (##2 b) or (##3 b) or (##4 b) or (##5 b) );
```

Upon a match of variable `a`, a match of any thread of the consequent terminates the attempt as successful, and the assertion passes. However, if upon a success of **$rose(a)**, there is no match of any thread of the consequent, assuming all of them were exercised, then the assertion fails.

a[*1:2] ##1 b |-> c
same as:
(a ##1 b) **or**
(a ##1 a ##1 b) |-> c

@300ns : Start of attempt
Attempt has 2 threads
- Thread1 succeeds @ 400 ns
- Thread2 succeeds @ 500 ns

attempt@300 → PASS @500

@400 ns : Start of attempt
Attempt has 2 threads
- Thread1 succeeds @ 500 ns
- Thread2 vacuous @ 500 ns
(a == 0 @ 500 ns)

attempt@400 → PASS @500

@500 ns : Start of attempt
Attempt has 0 threads,
Attempt is vacuous pass
(a == 0 @ 500 ns)

attempt@500 →
VACUOUS PASS @500

@600 ns : Start of attempt
Attempt has 2 threads
- Thread1 FAILS @ 700 ns
(b==1, c==0 @ 700 ns)
- Thread2 vacuous @ 600 ns
(a == 0 @ 700 ns)

attempt@600 → FAIL @700

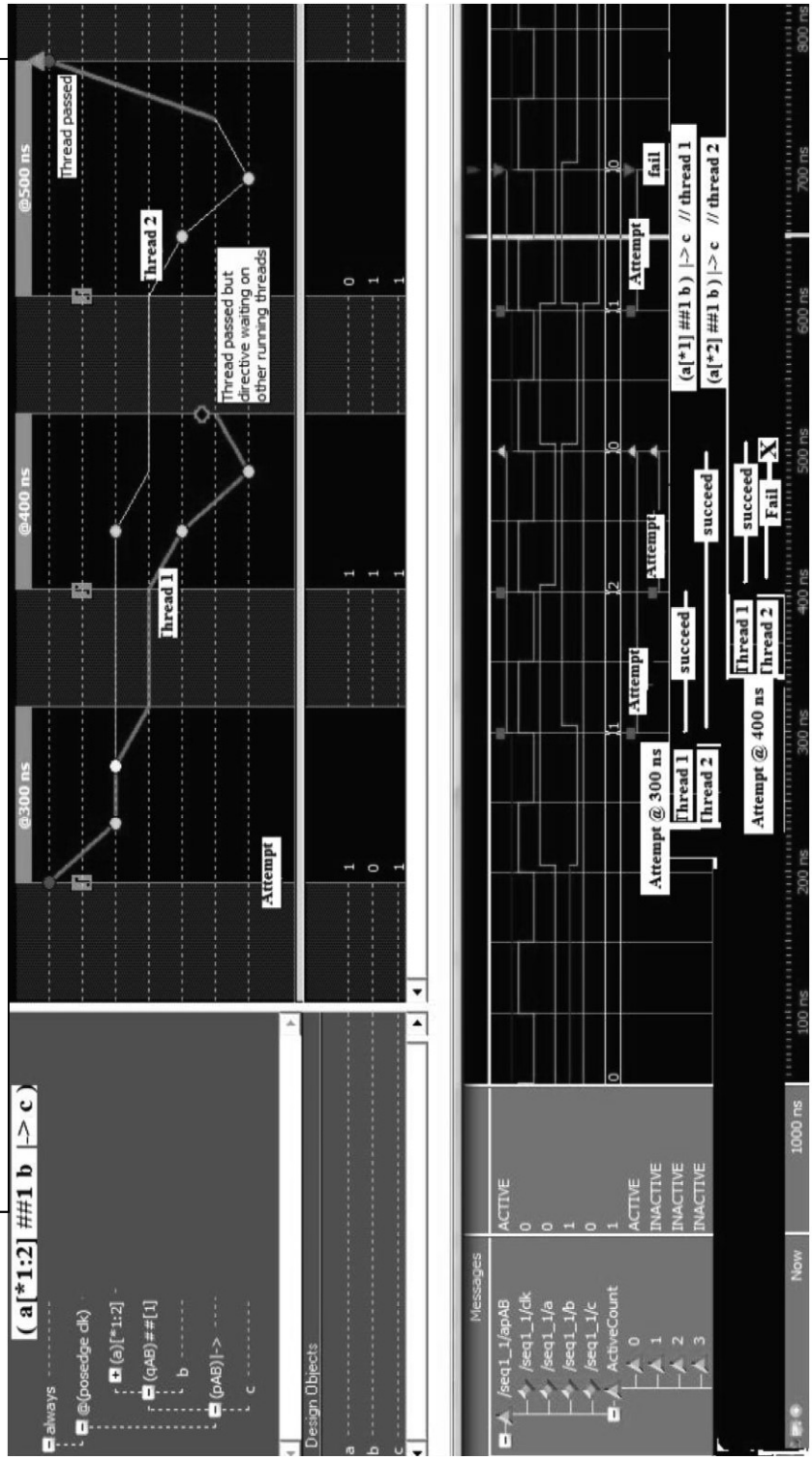

Figure 2.3.2-1 Simulation (a[*1:2] ##1 b |-> c)
(Ch2/2.3/sect3_1.sv, sect3_1_attempt_thread.jpg, sect3_1_attempt_thread300.jpg, sect3_1_attempt_thread400.jpg).
QuestaSim and ModelSim DE screen shot, Courtesy of Mentor Graphics