# Section 17
# Assertions

## 17.1 Introduction (informative)

SystemVerilog adds features to specify assertions of a system. An assertion specifies a behavior of the system. Assertions are primarily used to validate the behavior of a design. In addition, assertions can be used to provide functional coverage and generate input stimulus for validation.

There are two kinds of assertions: concurrent and immediate.

— Immediate assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block. Immediate assertions are primarily intended to be used with simulation.

— Concurrent assertions are based on clock semantics and use sampled values of variables. One of the goals of SystemVerilog assertions is to provide a common semantic meaning for assertions so that they can be used to drive various design and verification tools. Many tools, such as formal verification tools, evaluate circuit descriptions using cycle-based semantics, which typically relies on a clock signal or signals to drive the evaluation of the circuit. Any timing or event behavior between clock edges is abstracted away. Concurrent assertions incorporate these clock semantics. While this approach generally simplifies the evaluation of a circuit description, there are a number of scenarios under which this cycle-based evaluation provides different behavior from the standard event-based evaluation of SystemVerilog.

This section describes both types of assertions.

## 17.2 Immediate assertions

The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code. The expression is non-temporal and is interpreted the same way as an expression in the condition of a procedural **`if`** statement. That is, if the expression evaluates to X, Z or 0, then it is interpreted as being false and the assertion is said to fail. Otherwise, the expression is interpreted as being true and the assertion is said to pass.

The immediate **`assert`** statement is a *statement_item* and can be specified anywhere a procedural statement is specified.

```
procedural_assertion_statement ::=                    // from Annex A.6.10
        ...
      | immediate_assert_statement
immediate_assert_statement ::=
        assert ( expression ) action_block
action_block ::=                                      // from Annex A.6.3
          statement _or_null
      | [ statement ] else statement
```

*Syntax 17-1—Immediate assertion syntax (excerpt from Annex A)*

The *action_block* specifies what actions are taken upon success or failure of the assertion. The statement associated with the success of the assert statement is the first statement. It is called the *pass statement* and is executed if the expression evaluates to true. The pass statement can, for example, record the number of successes for a coverage log, but can be omitted altogether. If the pass statement is omitted, then no user-specified action is taken when the assert expression is true. The statement associated with **`else`** is called a *fail statement* and is executed if the expression evaluates to false. The **`else`** statement can also be omitted. The action block is executed immediately after the evaluation of the assert expression.

The optional statement label (identifier and colon) creates a named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the `%m` format specification.

```
assert_foo : assert(foo) $display("%m passed"); else $display("%m failed");
```

Note: The assertion control system tasks are described in Section 23.9.

Since the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is *error*. Other severity levels can be specified by including one of the following severity system tasks in the fail statement:

— `$fatal` is a run-time fatal.

— `$error` is a run-time error.

— `$warning` is a run-time warning, which can be suppressed in a tool-specific manner.

— `$info` indicates that the assertion failure carries no specific severity.

The syntax for these system tasks is shown in Section 23.8.

If an assertion fails and no **`else`** clause is specified, the tool shall, by default, call `$error`, unless a tool-specific option, such as a command-line option, is enabled to suppress the failure.

All of these severity system tasks shall print a tool-specific message indicating the severity of the failure, and specific information about the specific failure, which shall include the following information:

— The file name and line number of the assertion statement.

— The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also include the simulation run-time at which the severity system task is called.

Each system task can also include additional user-specified information using the same format as the Verilog `$display`.

If more than one of these system tasks is included in the **`else`** clause, then each shall be executed as specified.

If the severity system task is executed at a time other than when the assertion fails, the actual failure time of the assertion can be recorded and displayed programmatically. For example:

```
time t;

always @(posedge clk)
   if (state == REQ)
      assert (req1 || req2)
      else begin
         t = $time;
         #5 $error("assert failed at time %0t",t);
      end
```

If the assertion fails at time 10, the error message shall be printed at time 15, but the user-defined string printed shall be “assert failed at time 10”.

The display of messages of warning and info types can be controlled by a tool-specific option, such as a command-line option.

Since the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it can also be used to signal a failure to another part of the testbench.

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;
```

 

```
assert (y == 0) else flag = 1;
```

## 17.3 Concurrent assertions overview

Concurrent assertions describe behavior that spans over time. Unlike immediate assertions, the evaluation model is based on a clock such that a concurrent assertion is evaluated only at the occurrence of a clock tick. The values of variables used in the evaluation are the sampled values. This way, a predictable result can be obtained from the evaluation, regardless of the simulator's internal mechanism of ordering events and evaluating events. This model of execution also corresponds to the synthesis model of hardware interpretation from an RTL description.

The values of variables used in assertions are sampled in the Preponed region of a time slot and the assertions are evaluated during the Observe region. This is explained in Section 14, Scheduling Semantics.

The timing model employed in a concurrent assertion specification is based on clock ticks and uses a generalized notion of clock cycles. The definition of a clock is explicitly specified by the user and can vary from one expression to another.

A *clock tick* is an atomic moment in time that itself spans no duration of time. A clock shall tick only once at any simulation time and the sampled values for that simulation time are used for evaluation of concurrent assertions. In an assertion, the sampled value is the only valid value of a variable at a clock tick. Figure 17-1 shows the values of a variable as the clock progresses. The value of signal `req` is low at clock ticks 1 and 2. At clock tick 3, the value is sampled as high and remains high until clock tick 6. The sampled value of variable `req` at clock tick 6 is low and remains low until clock tick 10. Notice that the simulation value transitions to high at clock tick 9. However, the sampled value at clock tick 9 is low.

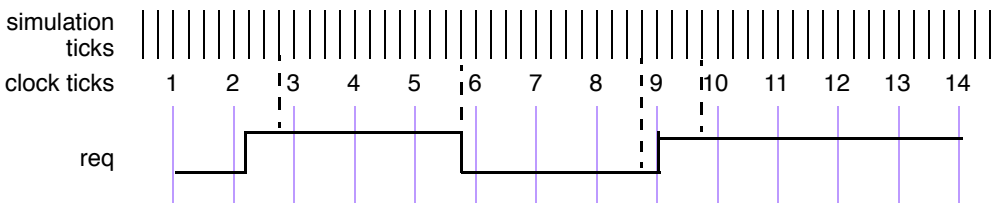


**Figure 17-1 — Sampling a variable on simulation ticks**

An expression used in an assertion is always tied to a clock definition. The sampled values are used to evaluate value change expressions or boolean subexpressions that are required to determine a match of a sequence.

Note:

— It is important to ensure that the defined clock behavior is glitch free. Otherwise, wrong values can be sampled.

— If a variable that appears in the expression for clock also appears in an expression with an assertion, the values of the two usages of the variable can be different. The current value of the variable is used in the clock expression, while the sampled value of the variable is used within the assertion.

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. Expressions such as `(clk && gating_signal)` and `(clk iff gating_signal)` can be used to represent a gated clock. Other more complex expressions are possible. However, in order to ensure proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the signals in a clock expression must be glitch-free and should only transition once at any simulation time.

An example of a concurrent assertion is:

```
base_rule1: assert property (cont_prop(rst,in1,in2)) pass_stat else fail_stat;
```

The keyword **property** distinguishes a concurrent assertion from an immediate assertion. The syntax of concurrent assertions is discussed in 17.13.

## 17.4 Boolean expressions

The expressions used in sequences are evaluated over sampled values of the variables that appear in the expressions. The outcome of the evaluation of an expression is boolean and is interpreted the same way as an expression is interpreted in the condition of a procedural **if** statement. That is, if the expression evaluates to `x`, `z`, or `0`, then it is interpreted as being false. Otherwise, it is true.

There are certain restrictions on the expressions that can appear in concurrent assertions. The restrictions on operand types, variables, and operators are specified in the following sections.

Expressions are allowed to include function calls, but certain semantic restrictions are imposed.

— Functions that appear in expressions cannot contain output or **ref** arguments (**const ref** are allowed).

— Functions should be automatic (or preserve no state information) and have no side effects.

### 17.4.1 Operand types

The following types are not allowed:

— non-integer types (**shortreal**, **real** and **realtime**)

— **string**

— **event**

— **chandle**

— **class**

— associative arrays

— dynamic arrays

Fixed size arrays, packed or unpacked, can be used as a whole or as part selects or as indexed bit or part selects. The indices can be constants, parameters, or variables.

The following example shows some possible forms of comparison of members of structures and unions:

```
typedef int [4] array;
typedef struct { int a, b, c,d } record;
union { record r; array a; } p, q;
```

The following comparisons are legal in expressions:

```
p.a == q.a
```

and

```
p.r == q.r
```

The following example provides further illustration of the use of arrays in expressions.

```
logic [7:0] arrayA [0:15], arrayB[0:15];
```

The following comparisons are legal:

```
arrayA == arrayB;
```

 

```
arrayA != arrayB;
arrayA[i] >= arrayB[j];
arrayB[i][j+:2] == arrayA[k][m-:2];
(arrayA[i] & (~arrayB[j])) == 0;
```

### 17.4.2 Variables

The variables that can appear in expressions must be static design variables or function calls returning values of types described in Section 17.4.1. Static variables declared in programs, interfaces or clocking blocks can also be accessed. If a reference is to a static variable declared in a task, that variable is sampled as any other variable, independent of calls to the task.

### 17.4.3 Operators

All operators that are valid for the types described in Section 17.4.1 are allowed with the exception of assignment operators and increment and decrement operators. SystemVerilog includes the C assignment operators, such as **+=**, and the C increment and decrement operators, **++** and **--**. These operators cannot be used in expressions that appear in assertions. This restriction prevents side effects.

## 17.5 Sequences

```
sequence_expr ::=                                                    // from Annex A.2.10
        cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | expression_or_dist [ boolean_abbrev ]
      | ( expression_or_dist {, sequence_match_item } ) [ boolean_abbrev ]
      | sequence_instance [ sequence_abbrev ]
      | ( sequence_expr {, sequence_match_item } ) [ sequence_abbrev ]
      | sequence_expr and sequence_expr
      | sequence_expr intersect sequence_expr
      | sequence_expr or sequence_expr
      | first_match ( sequence_expr {, sequence_match_item} )
      | expression_or_dist throughout sequence_expr
      | sequence_expr within sequence_expr
      | clocking_event sequence_expr
cycle_delay_range ::=
        ## integral_number
      | ## identifier
      | ## ( constant_expression )
      | ## [ cycle_delay_const_range_expression ]
sequence_match_item ::=
        operator_assignment
      | inc_or_dec_expression
      | subroutine_call
sequence_instance ::=
        ps_sequence_identifier [ ( [ actual_arg_list ] ) ]
actual_arg_list ::=
        actual_arg_expr { , actual_arg_expr }
      | . formal_identifier ( actual_arg_expr ) { , . formal_identifier ( actual_arg_expr ) }
actual_arg_expr ::=
        event_expression
      | $
boolean_abbrev ::=
        consecutive_repetition
      | non_consecutive_repetition
      | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression ]
non_consecutive_repetition ::= [= const_or_range_expression ]
goto_repetition ::= [-> const_or_range_expression ]
const_or_range_expression ::=
        constant_expression
      | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
        constant_expression : constant_expression
      | constant_expression : $
expression_or_dist ::= expression [ dist { dist_list } ]
```

*Syntax 17-2—Sequence syntax (excerpt from Annex A)*

 

Properties are often constructed out of sequential behaviors. The **sequence** feature provides the capability to build and manipulate sequential behaviors. The simplest sequential behaviors are linear. A *linear sequence* is a finite list of SystemVerilog boolean expressions in a linear order of increasing time. The linear sequence is said to match along a finite interval of consecutive clock ticks provided the first boolean expression evaluates to true at the first clock tick, the second boolean expression evaluates to true at the second clock tick, and so forth, up to and including the last boolean expression evaluating to true at the last clock tick. A single boolean expression is an example of a simple linear sequence, and it matches at a single clock tick provided the boolean expression evaluates to true at that clock tick.

More complex sequential behaviors are described by SystemVerilog sequences. A sequence is a regular expression over the SystemVerilog boolean expressions that concisely specifies a set of zero, finitely many, or infinitely many linear sequences. If at least one of the linear sequences from this set matches along a finite interval of consecutive clock ticks, then the sequence is said to match along that interval.

A property may involve 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. 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.

Sequences can be composed by concatenation, analogous to a concatenation of lists. The concatenation specifies a delay, using ##, from the end of the first sequence until the beginning of the second sequence.

The following is the syntax for sequence concatenation.

```
sequence_expr ::=                                                    // from Annex A.2.10
        cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      ...
cycle_delay_range ::=
        ## integral_number
      | ## identifier
      | ## ( constant_expression )
      | ## [ cycle_delay_const_range_expression ]
cycle_delay_const_range_expression ::=
        constant_expression : constant_expression
      | constant_expression : $
```

*Syntax 17-3—Sequence concatenation syntax (excerpt from Annex A)*

In this syntax:

— *constant_expression* is computed at compile time and must result in an integer value.

— *constant_expression* can only be 0 or greater.

— The **$** token is used to indicate the end of simulation. For formal verification tools, **$** is used to indicate a finite, unbounded, range.

— When a range is specified with two expressions, the second expression must be greater or equal to the first expression.

The context in which a sequence occurs determines when the sequence is evaluated. The first expression in a sequence is checked at the first occurrence of the clock tick at or after the expression that triggered evaluation of the sequence. Each successive element (if any) in the sequence is checked at the next subsequent occurrence of the clock.

A ## followed by a number or range specifies the delay from the current clock tick to the beginning of the

sequence that follows. The delay `##1` indicates that the beginning of the sequence that follows is one clock tick later than the current clock tick. The delay `##0` indicates that the beginning of the sequence that follows is at the same clock tick as the current clock tick.

When used as a concatenation between two sequences, the delay is from the end of the first sequence to the beginning of the second sequence. The delay `##1` indicates that the beginning of the second sequence is one clock tick later than the end of the first sequence. The delay `##0` indicates that the beginning of the second sequence is at the same clock tick as the end of the first sequence.

The following are examples of delay expressions. `'true` is a boolean expression that always evaluates to true and is used for visual clarity. It can be defined as:

```
`define true 1

##0 a       // means a
##1 a       // means `true ##1 a
##2 a       // means `true ##1 `true ##1 a
##[0:3]a    // means (a) or (`true ##1 a) or (`true ##1 `true ##1 a) or
               (`true ##1 `true ##1 `true ##1 a)
a ##2 b // means a ##1 `true ##1 b
```

The sequence:

```
req ##1 gnt ##1 !req
```

specifies that `req` be true on the current clock tick, `gnt` shall be true on the first subsequent tick, and `req` shall be false on the next clock tick after that. The `##1` operator specifies one clock tick separation. A delay of more than one clock tick can be specified, as in:

```
req ##2 gnt
```

This specifies that `req` shall be true on the current clock tick, and `gnt` shall be true on the second subsequent clock tick, as shown in Figure 17-2.

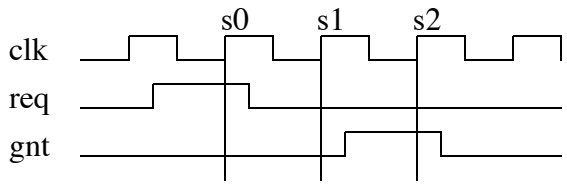


**Figure 17-2 — Concatenation of sequences**

The following specifies that signal `b` shall be true on the Nth clock tick after signal `a`:

```
a ##N b     // check b on the Nth sample
```

To specify a concatenation of overlapped sequences, where the end point of one sequence coincides with the start of the next sequence, a value of 0 is used, as shown below.

```
a ##1 b ##1 c // first sequence seq1
d ##1 e ##1 f // second sequence seq2
(a ##1 b ##1 c) ##0 (d ##1 e ##1 f) // overlapped concatenation
```

In the above example, `c` must be true at the endpoint of sequence `seq1`, and `d` must be true at the start of sequence `seq2`. When concatenated with 0 clock tick delay, c and d must be true at the same time, resulting in a concatenated sequence equivalent to:

 

```
a ##1 b ##1 c&&d ##1 e ##1 f
```

It should be noted that no other form of overlapping between the sequences can be expressed using the concatenation operation.

In cases where the delay can be any value in a range, a time window can be specified as follows:

```
req ##[4:32] gnt
```

In the above case, signal `req` must be true at the current clock tick, and signal `gnt` must be true at some clock tick between the 4th and the 32nd clock tick after the current clock tick.

The time window can extend to a finite, but unbounded, range by using `$` as in the example below.

```
req ##[4:$] gnt
```

A sequence can be unconditionally extended by concatenation with `'true`.

```
a ##1 b ##1 c ##3 'true
```

After satisfying signal `c`, the sequence length is extended by 3 clock ticks. Such adjustments in the length of sequences can be required when complex sequences are constructed by combining simpler sequences.

## 17.6 Declaring sequences

A **`sequence`** can be declared in

- a module
- an interface
- a program
- a clocking block
- a package
- a compilation-unit scope

Sequences are declared using the following syntax.:

```
concurrent_assertion_item_declaration ::=                    // from Annex A.2.10
        ...
      | sequence_declaration
sequence_declaration ::=
        sequence sequence_identifier [ ( [ list_of_formals ] ) ] ;
            { assertion_variable_declaration }
            sequence_expr ;
        endsequence [ : sequence_identifier ]
sequence_instance ::=
        ps_sequence_identifier [ ( [ actual_arg_list ] ) ]
actual_arg_list ::=
        actual_arg_expr { , actual_arg_expr }
      | . formal_identifier ( actual_arg_expr ) { , . formal_identifier ( actual_arg_expr ) }
actual_arg_expr ::=
        event_expression
      | $
assertion_variable_declaration ::=
        data_type list_of_variable_identifiers ;
```

*Syntax 17-4—Declaring sequence syntax (excerpt from Annex A)*

The *clocking_event* specifies the clock for the sequence.

A sequence is declared with optional formal arguments. When a sequence is instantiated, actual arguments can be passed to the sequence. The sequence gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. Semantic checks are performed to ensure that the expanded sequence with the actual arguments is legal.

An actual argument can replace an:

— identifier

— expression

— event control expression

— upper range as $

Note that variables used in a sequence that are not formal arguments to the sequence are resolved according to the scoping rules from the scope in which the sequence is declared.

```
sequence s1;
   @(posedge clk) a ##1 b ##1 c;
endsequence
sequence s2;
   @(posedge clk) d ##1 e ##1 f;
endsequence
sequence s3;
   @(negedge clk) g ##1 h ##1 i;
endsequence
```

In this example, sequences `s1` and `s2` are evaluated on successive posedge events of `clk`. The sequence `s3` is evaluated on successive negedge events of `clk`.

Another example of sequence declaration, which includes arguments is shown below:

 

```
sequence s20_1(data,en);
   (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
endsequence
```

Sequence `s20_1` does not specify a clock. In this case, a clock would be inherited from some external source, such as a **`property`** or an **`assert`** statement. A sequence can be referred to by its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. A sequence can be referenced in a **`property`**, an **`assert`** statement, or a **`cover`** statement.

To use a named sequence as a subsequence of another sequence, simply reference its name. The evaluation of a sequence that references a named sequence is performed in the same way as if the named sequence was contained as a lexical part of the referencing sequence, with the formal arguments of the named sequence replaced by the actual ones and the remaining variables in the named sequence resolved according to the scope of the declaration of the named sequence. An example is shown below:

```
sequence s;
   a ##1 b ##1 c;
endsequence
sequence rule;
   @(posedge sysclk)
   trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
```

Sequence `rule` in the preceding example is equivalent to:

```
sequence rule;
   @(posedge sysclk)
   trans ##1 start_trans ##1 a ##1 b ##1 c ##1 end_trans ;
endsequence
```

Any form of syntactic cyclic dependency of the sequence names is disallowed. The example below illustrates an illegal dependency of `s1` on `s2` and `s2` on `s1`, because it creates a cyclic dependency.

```
sequence s1;
   @(posedge sysclk) (x ##1 s2);
endsequence
sequence s2;
   @(posedge sysclk) (y ##1 s1);
endsequence
```

## 17.7 Sequence operations

### 17.7.1 Operator precedence

Operator precedence and associativity are listed in Table 17-1, below. The highest precedence is listed first.

**Table 17-1: Operator precedence and associativity**

| SystemVerilog expression operators | Associativity |
|---|---|
| `[* ]  [= ]  [-> ]` | ---- |
| **`##`** | left |
| **`throughout`** | right |
| **`within`** | left |

**Table 17-1: Operator precedence and associativity**

| | |
|---|---|
| **intersect** | left |
| **and** | left |
| **or** | left |

### 17.7.2 Repetition in sequences

Following is the syntax for sequence repetition.

```
sequence_expr ::=                                                    // from Annex A.2.10
        ...
      | expression_or_dist [ boolean_abbrev ]
      | ( expression_or_dist {, sequence_match_item } ) [ boolean_abbrev ]
      | sequence_instance [ sequence_abbrev ]
      | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
        ...
boolean_abbrev ::=
        consecutive_repetition
      | non_consecutive_repetition
      | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression ]
non_consecutive_repetition ::= [= const_or_range_expression ]
goto_repetition ::= [-> const_or_range_expression ]
const_or_range_expression ::=
        constant_expression
      | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
        constant_expression : constant_expression
      | constant_expression : $
```

*Syntax 17-5—Sequence repetition syntax (excerpt from Annex A)*

The number of iterations of a repetition can either be specified by exact count or be required to fall within a finite range. If specified by exact count, then the number of iterations is defined by a non-negative integer constant expression. If required to fall within a finite range, then the minimum number of iterations is defined by a non-negative integer constant expression and the maximum number of iterations is either defined by a non-negative integer constant expression or is `$`, indicating a finite, but unbounded maximum.

If both the minimum and maximum numbers of iterations are defined by non-negative integer constant expressions, then the minimum number must be less than or equal to the maximum number.

Three kinds of repetition are provided:

— *consecutive repetition* ( [* ): Consecutive repetition specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand.

— *goto repetition* ( `[->` ): Goto repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive

 

match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

— *non-consecutive repetition* ( `[=` ): Non-consecutive repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand.

The effect of consecutive repetition of a subsequence within a sequence can be achieved by explicitly iterating the subsequence, as:

```
a ##1 b ##1 b ##1 b ##1 c
```

Using the consecutive repetition operator `[*3]`, which indicates 3 iterations, this sequential behavior is specified more succinctly:

```
a ##1 b [*3] ##1 c
```

A consecutive repetition specifies that the operand sequence must match a specified number of times. The consecutive repetition operator `[*N]` specifies that the operand sequence must match N times in succession. For example:

```
a [*3] means a ##1 a ##1 a
```

Using 0 as the repetition number, an empty sequence results, as:

```
a [*0]
```

An *empty sequence* is one that does not match over any positive number of clocks. The following rules apply for concatenating sequences with empty sequences. An empty sequence is denoted as *empty* and a sequence is denoted as *seq*.

— (*empty* `##0` *seq*) does not result in a match

— (*seq* `##0` *empty*) does not result in a match

— (*empty* `##n` *seq*), where n is greater than 0, is equivalent to (##(n-1) seq)

— (*seq* `##n` *empty*), where n is greater than 0, is equivalent to (seq ##(n-1) 'true)

For example,

```
b ##1 ( a[*0] ##0 c)
```

produces no match of the sequence.

```
b ##1 a[*0:1] ##2 c
```

is equivalent to

```
(b ##2 c) or (b ##1 a ##2 c)
```

The syntax allows combination of a delay and repetition in the same sequence. The following are both allowed:

```
`true ##3 (a [*3])   // means `true ##1 `true ##1 `true ##1 a ##1 a ##1 a
(`true ##2 a) [*3]   // means (`true ##2 a) ##1 (`true ##2 a) ##1
                     // (`true ##2 a), which in turn means `true ##1 `true ##1
                     // a ##1 `true ##1 `true ##1 a ##1 `true ##1 `true ##1 a
```

A sequence can be repeated as follows:

```
(a ##2 b) [*5]
```

This is the same as:

```
(a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

A repetition with a range of `min` minimum and `max` maximum number of iterations can be expressed with the consecutive repetition operator `[* min:max]`.

As an example,

```
(a ##2 b)[*1:5]
```

is equivalent to

```
(a ##2 b)
or (a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

Similarly,

```
(a[*0:3] ##1 b ##1 c)
```

is equivalent to

```
(b ##1 c)
or (a ##1 b ##1 c)
or (a ##1 a ##1 b ##1 c)
or (a ##1 a ##1 a ##1 b ##1 c)
```

To specify a finite, but unbounded, number of iterations, the dollar sign ( `$` ) is used. For example, the repetition:

```
a ##1 b [*1:$] ##1 c
```

matches over an interval of three or more consecutive clock ticks if `a` is true on the first clock tick, `c` is true on the last clock tick, and `b` is true at every clock tick strictly in between the first and the last.

Specifying the number of iterations of a repetition by exact count is equivalent to specifying a range in which the minimum number of repetitions is equal to the maximum number of repetitions. In other words, `seq[*n]` is equivalent to `seq[*n:n]`.

The *goto repetition* (non-consecutive exact repetition) takes a boolean expression rather than a sequence as operand. It specifies the iterative matching of the boolean expression at clock ticks that are not necessarily consecutive and ends at the last iterative match. For example,

```
a ##1 b [->2:10] ##1 c
```

matches over an interval of consecutive clock ticks provided `a` is true on the first clock tick, `c` is true on the last clock tick, `b` is true on the penultimate clock tick, and, including the penultimate, there are at least 2 and at most 10 not-necessarily-consecutive clock ticks strictly in between the first and last on which `b` is true. This sequence is equivalent to:

```
a ##1 ((!b[*0:$] ##1 b) [*2:10]) ##1 c
```

The *non-consecutive repetition* is like the goto repetition except that `a` match does not have to end at the last iterative match of the operand boolean expression. The use of non-consecutive repetition instead of goto repe-

 

tition allows the match to be extended by arbitrarily many clock ticks provided the boolean expression is false on all of the extra clock ticks. For example,

```
a ##1 b [=2:10] ##1 c
```

matches over an interval of consecutive clock ticks provided `a` is true on the first clock tick, `c` is true on the last clock tick, and there are at least 2 and at most 10 not-necessarily-consecutive clock ticks strictly in between the first and last on which `b` is true. This sequence is equivalent to:

```
a ##1 ((!b [*0:$] ##1 b) [*2:10]) ##1 !b[*0:$] ##1 c
```

### 17.7.3 Sampled value functions

This section describes the system functions available for accessing sampled values of an expression. These functions include the capability to access current sampled value, access sampled value in the past, or detect changes in sampled value of an expression. Sampling of an expression is explained in Section 17.3. The following functions are provided.

```
$sampled(expression [, clocking_event])

$rose( expression [, clocking_event])

$fell( expression [, clocking_event])

$stable( expression [, clocking_event])

$past( expression1 [, number_of_ticks] [, expression2] [, clocking_event])
```

The use of these functions is not limited to assertion features; they can be used as expressions in procedural code as well. The clocking event, although optional as an explicit argument to the functions, is required for their semantics. The clocking event is used to sample the value of the argument expression.

The clocking event must be explicitly specified as an argument, or inferred from the code where it is used. The following rules are used to infer the clocking event:

— if used in an assertion, the appropriate clocking event from the assertion is used.

— if used in an action block of a singly-clocked assertion, the clock of the assertion is used.

— if used in a procedural block, the inferred clock, if any, for the procedural code (Section 17.13.5) is used.

Otherwise, default clocking (Section 15.11) is used.

When these functions are used in an assertion, the clocking event argument of the functions, if specified, shall be identical to the clocking event of the expression in the assertion. In the case of multi-clock assertion, the appropriate clocking event for the expression where the function is used, is applied to the function.

Function `$sampled` returns the sampled value of the expression with respect to the last occurrence of the clocking event. When `$sampled` is invoked prior to the occurrence of the first clocking event, the value of X is returned. The use of `$sampled` in assertions, although allowed, is redundant, as the result of the function is identical to the sampled value of the expression itself used in the assertion.

Three functions are provided to detect changes in sampled values: `$rose`, `$fell` and `$stable`.

A value change function detects the change in the sampled value of an expression. The clocking event is used to obtain the sampled value of the argument expression at a clock tick prior to the current simulation time unit. Here, the current simulation time unit refers to the simulation time unit in which the function is evaluated. This sampled value is compared against the value of the expression determined at the prepone time of the current simulation time unit. The result of a value change expression is true or false and can be used as a boolean expression.

`$rose` returns true if the least significant bit of the expression changed to 1. Otherwise, it returns false.

`$fell` returns true if the least significant bit of the expression changed to 0. Otherwise, it returns false.

`$stable` returns true if the value of the expression did not change. Otherwise, it returns false.

When these functions are called at or before the first clock tick of the clocking event, the results are computed by comparing the current sampled value of the expression to X.

Figure 17-3 illustrates two examples of value changes:

— Value change expression `e1` is defined as `$rose(req)`

— Value change expression `e2` is defined as `$fell(ack)`

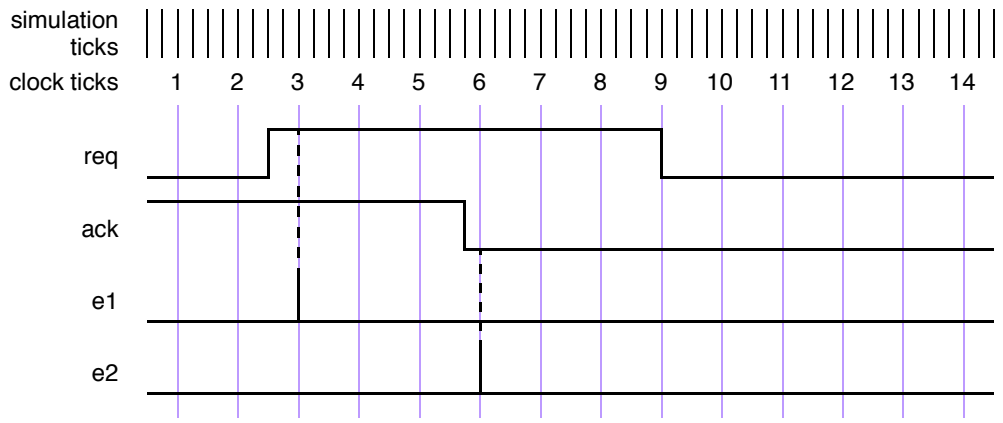


**Figure 17-3 — Value change expressions**

The clock ticks used for sampling the variables are derived from the clock for the property, which is different from the simulation ticks. Assume, for now, that this clock is defined elsewhere. At clock tick 3, `e1` occurs because the value of `req` at clock tick 2 was low and at clock tick 3, the value is high. Similarly, `e2` occurs at clock tick 6 because the value of `ack` was sampled as high at clock tick 5 and sampled as low at clock tick 6.

The example below illustrates the use of `$rose` in SystemVerilog code outside assertions.

```
always @(posedge clk)
  reg1 <= a & $rose(b);
```

In this example, the clocking event `(posedge clk)` is applied to `$rose`. `$rose` is true whenever the sampled value of `b` changed to 1 from its sampled value at the previous tick of the clocking event.

In addition to accessing value changes, the past values can be accessed with the `$past` function. The following three optional arguments are provided:

*expression2* is used as a gating expression for the clocking event

*number_of_ticks* specifies the number of clock ticks in the past

*clocking_event* specifies the clocking event for sampling expression1

*expression1* and *expression2* can be any expression allowed in assertions.

*number_of_ticks* must be one or greater. If *number_of_ticks* is not specified, then it defaults to 1. `$past` returns the sampled value of the expression that was present *number_of_ticks* prior to the time of evaluation of `$past`. A clock tick is based on clocking_event. If the specified clock tick in the past is before the start of simulation, the returned value from the `$past` function is a value of X.

 

The optional argument *clocking_event* specifies the clock for the function. The rules governing the usage of *clocking_event* are same as those described for the value change function.

When intermediate optional arguments between two arguments are not needed, a comma must be placed for each omitted argument. For example,

```
$past(in1, , enable);
```

Here, a comma is specified to omit *number_of_ticks*. The default of one is used for the empty *number_of_ticks* argument. Note that a comma for the omitted *clocking_event* argument is not needed, as it does not fall within the specified arguments.

`$past` can be used in any System Verilog expression. An example is shown below.

```
always @(posedge clk)
  reg1 <= a & $past(b);
```

In this example, the clocking event `(posedge clk)` is applied to `$past`. `$past` is evaluated in the current occurrence of `(posedge clk)`, and returns the value of `b` sampled at the previous occurrence of `(posedge clk)`.

When *expression2* is specified, the sampling of *expression1* is performed based on its clock gated with *expression2*. For example,

```
always @(posedge clk)
  if (enable) q <= d;

always @(posedge clk)
assert (done |=> (out == $past(q, 2,enable)) ;
```

In this example, the sampling of `q` for evaluating `$past` is based on the clocking expression

```
posedge clk iff enable
```

### 17.7.4 AND operation

The binary operator **and** is used when both operands are expected to match, but the end times of the operand sequences can be different.

```
sequence_expr ::=                                   // from Annex A.2.10
    ...
  | sequence_expr and sequence_expr
```

*Syntax 17-6—and operator syntax (excerpt from Annex A)*

The two operands of **and** are sequences. The requirement for the match of the **and** operation is that both the operands must match. The operand sequences start at the same time. When one of the operand sequences matches, it waits for the other to match. The end time of the composite sequence is the end time of the operand sequence that completes last.

When `te1` and `te2` are sequences, then the composite sequence:

```
te1 and te2
```

— Matches if `te1` and `te2` match.

— The end time is the end time of either `te1` or `te2`, whichever matches last.

The following example is a sequence with operator **and**, where the two operands are sequences.

```
(te1 ##2 te2) and (te3 ##2 te4 ##2 te5)
```

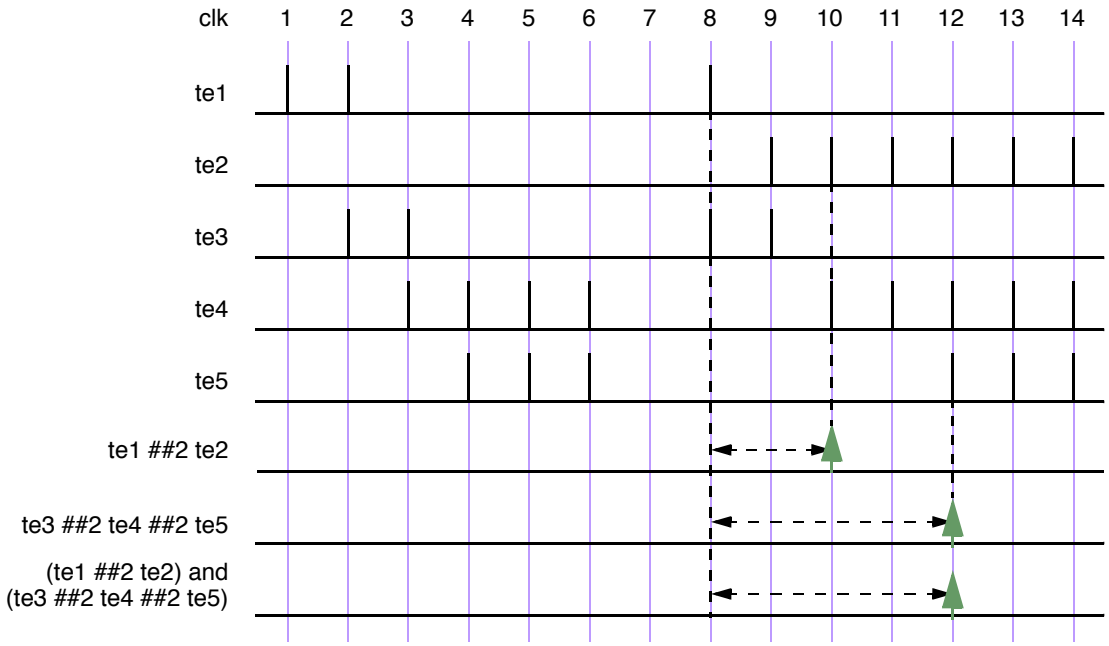


**Figure 17-4 — ANDing (and) two sequences**

The operation as illustrated in Figure 17-4 shows the evaluation attempt at clock tick 8. Here, the two operand sequences are `(te1 ##2 te2)` and `(te3 ##2 te4 ##2 te5)`. The first operand sequence requires that first `te1` evaluates to true followed by `te2` two clock ticks later. The second sequence requires that first `te3` evaluates to true followed by `te4` two clock ticks later, followed by `te5` two clock ticks later.

This attempt results in a match since both operand sequences match. The end times of matches for the individual sequences are clock ticks 10 and 12. The end time for the composite sequence is the later of the two end times, so a match is recognized for the composite sequence at clock tick 12.

In the following example, the first operand sequence has a concatenation operator with range from 1 to 5:

```
(te1 ##[1:5] te2) and (te3 ##2 te4 ##2 te5)
```

The first operand sequence requires that `te1` evaluate to true and that `te2` evaluate to true 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence is the same as in the previous example. To consider all possibilities of a match of the composite sequence, the following steps can be taken:

1) Five threads of evaluation are started for the five possible linear sequences associated with the first sequence operand.

2) The second operand sequence has only one associated linear sequence, so only one thread of evaluation is started for it.

3) Figure 17-5 shows the evaluation attempt beginning at clock tick 8. All five linear sequences for the first operand sequence match, as shown in a time window, so there are five matches of the first operand sequence, ending at clock ticks 9, 10, 11, 12 and 13 respectively. The second operand sequence matches at clock tick 12.

 

4) Each match of the first operand sequence is combined with the single match of the second operand sequence, and the rules of the and operation determine the end time of the resulting match of the composite sequence.

The result of this computation is five matches of the composite sequence, four of them ending at clock tick 12, and the fifth ending at clock tick 13. Figure 17-5 shows the matches of the composite sequence ending at clock ticks 12 and 13.

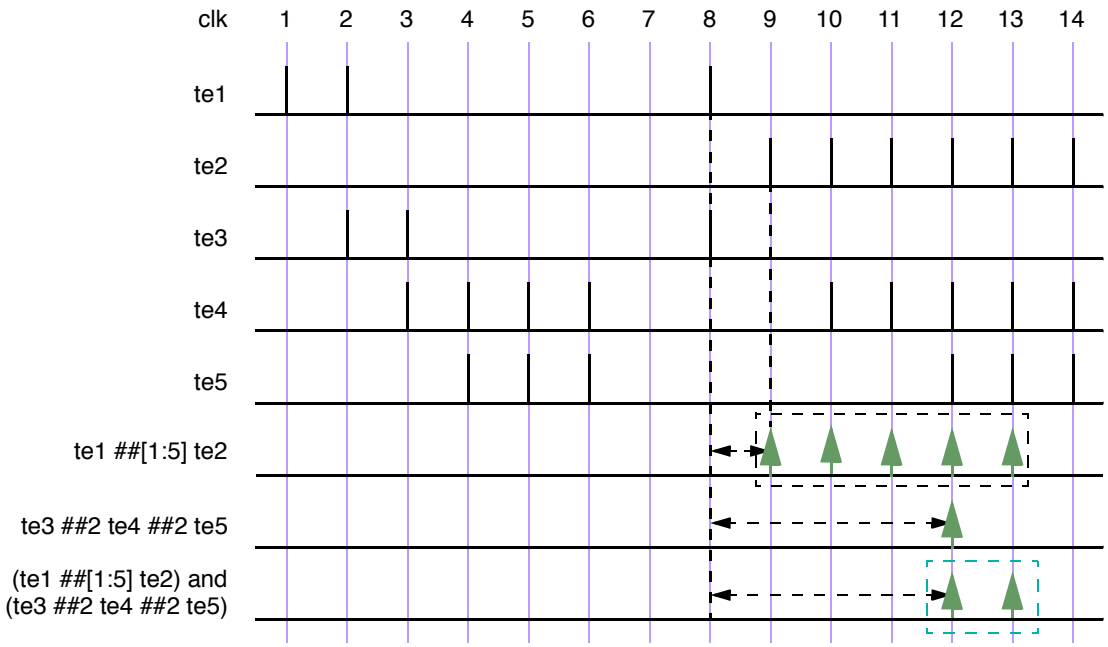


**Figure 17-5 — ANDing (and) two sequences, including a time range**

If `te1` and `te2` are sampled expressions (not sequences), the sequence (`te1` **`and`** `te2`) matches if `te1` and `te2` both evaluate to true.

An example is illustrated in Figure 17-6, which shows the results for attempts at every clock tick. The sequence matches at clock tick 1, 3, 8, and 14 because both `te1` and `te2` are simultaneously true. At all other clock ticks, match of the **`and`** operation fails because either `te1` or `te2` is false.

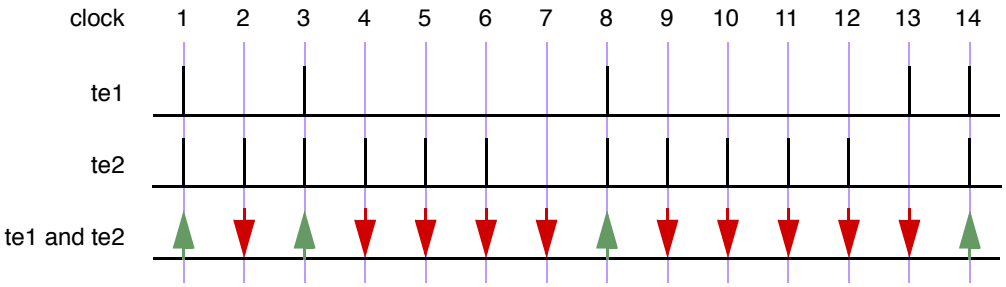


**Figure 17-6 — ANDing (and) two boolean expressions**

### 17.7.5 Intersection (AND with length restriction)

The binary operator **intersect** is used when both operand sequences are expected to match, and the end times of the operand sequences must be the same.

```
sequence_expr ::=                                    // from Annex A.2.10
    ...
  | sequence_expr intersect sequence_expr
```

*Syntax 17-7—intersect operator syntax (excerpt from Annex A)*

The two operands of **intersect** are sequences. The requirements for match of the **intersect** operation are:

— Both the operands must match.

— The lengths of the two matches of the operand sequences must be the same.

The additional requirement on the length of the sequences is the basic difference between **and** and **intersect**.

An attempted evaluation of an **intersect** sequence can result in multiple matches. The results of such an attempt can be computed as follows.

— Matches of the first and second operands that are of the same length are paired. Each such pair results in a match of the composite sequence, with length and endpoint equal to the shared length and endpoint of the paired matches of the operand sequences.

— If no such pair is found, then there is no match of the composite sequence.

Figure 17-7 is similar to Figure 17-5, except that **and** is replaced by **intersect**. In this case, unlike in Figure 17-5, there is only a single match at clock tick 12.

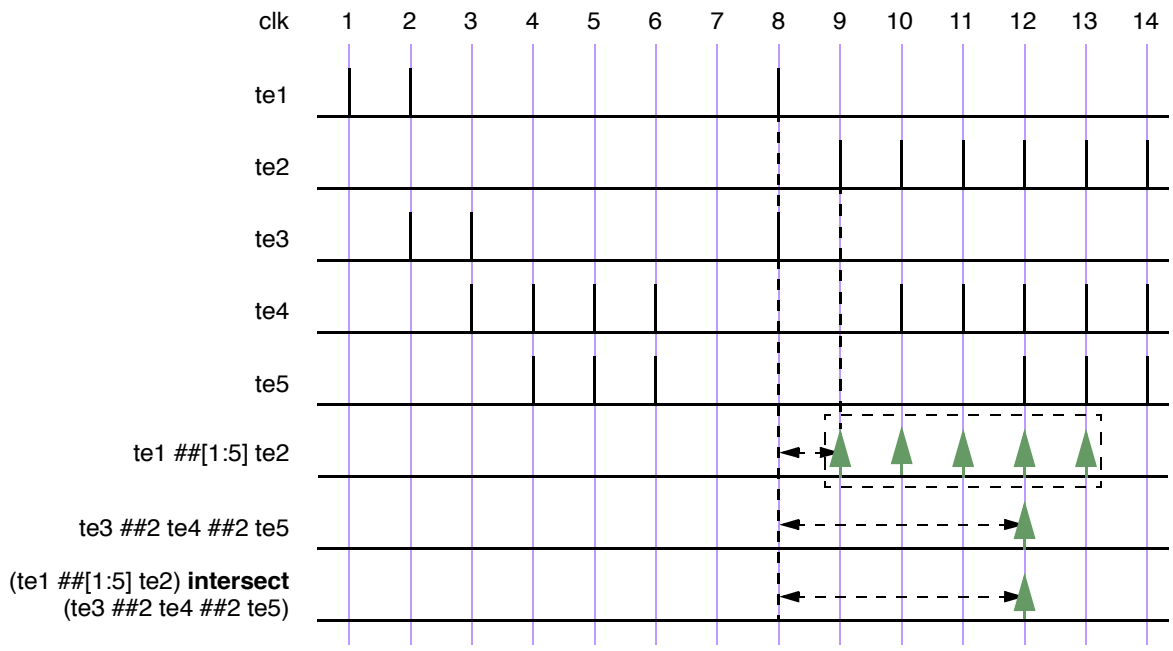


**Figure 17-7 — Intersecting two sequences**

 

### 17.7.6 OR operation

The operator **or** is used when at least one of the two operand sequences is expected to match.

```
sequence_expr ::=                                    // from Annex A.2.10
      ...
    | sequence_expr or sequence_expr
```

*Syntax 17-8—or operator syntax (excerpt from Annex A)*

The two operands of **or** are sequences.

If the operands `te1` and `te2` are expressions, then

```
te1 or te2
```

matches at any clock tick on which at least one of te1 and te2 evaluates to true.

Figure 17-8 illustrates an **or** operation for which the operands `te1` and `te2` are expressions. The composite sequence does not match at clock ticks 7 and 13 because `te1` and `te2` are both false at those times. At all other clock ticks, the composite sequence matches, as at least one of the two operands evaluates to true.

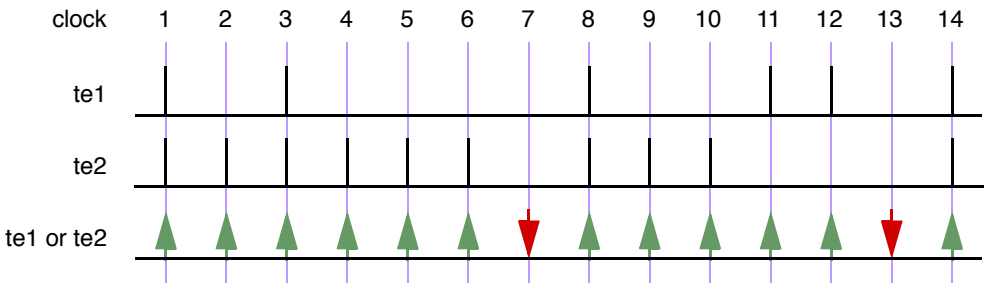


**Figure 17-8 — ORing (or) Two Sequences**

When `te1` and `te2` are sequences, then the sequence

```
te1 or te2
```

matches if at least one of the two operand sequences `te1` and `te2` matches. Each match of either `te1` or `te2` constitutes a match of the composite sequence, and its end time as a match of the composite sequence is the same as its end time as a match of `te1` or `of te2`. In other words, the set of matches of `te1` **or** `te2` is the union of the set of matches of `te1` with the set of matches of `te2`.

The following example shows a sequence with operator **or** where the two operands are sequences. Figure 17-9 illustrates this example.

```
(te1 ##2 te2) or (te3 ##2 te4 ##2 te5)
```

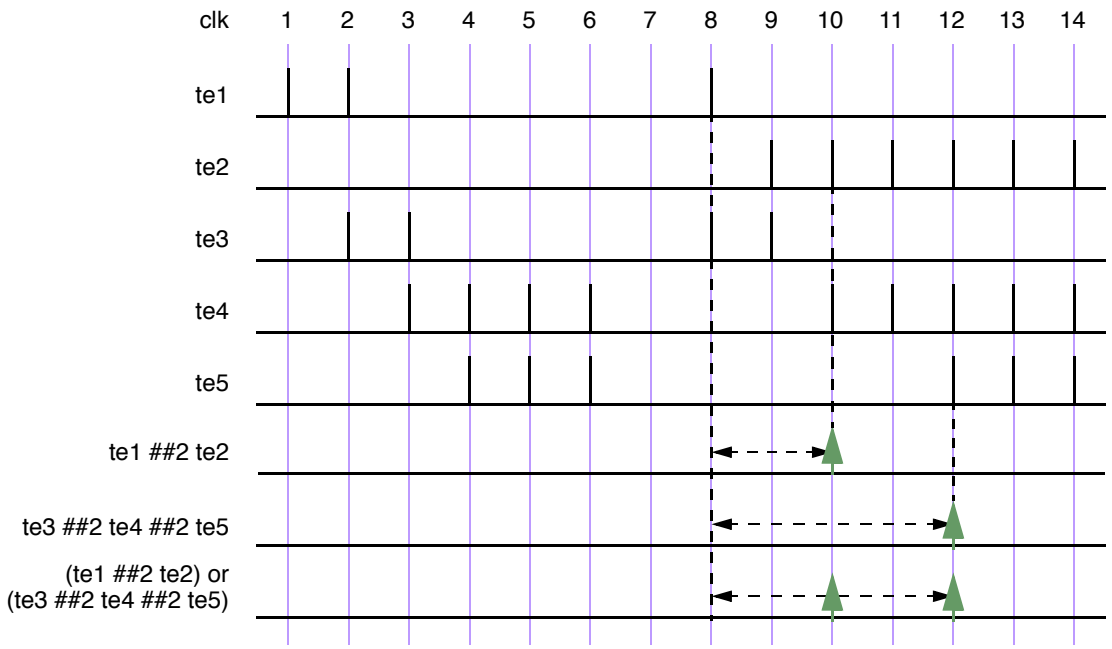


**Figure 17-9 — ORing (or) two sequences**

Here, the two operand sequences are: `(te1 ##2 te2)` and `(te3 ##2 te4 ##2 te5)`. The first sequence requires that `te1` first evaluates to true, followed by `te2` two clock ticks later. The second sequence requires that `te3` evaluates to true, followed by `te4` two clock ticks later, followed by `te5` two clock ticks later. In Figure 17-9, the evaluation attempt for clock tick 8 is shown. The first sequence matches at clock tick 10 and the second sequence matches at clock tick 12. So, two matches for the composite sequence are recognized.

In the following example, the first operand sequence has a concatenation operator with range from 1 to 5

```
(te1 ##[1:5] te2) or (te3 ##2 te4 ##2 te5)
```

The first operand sequence requires that `te1` evaluate to true and that `te2` evaluate to true 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence requires that `te3` evaluate to true, that `te4` evaluate to true 2 clock ticks later, and that `te5` evaluate to true another 2 clock ticks later. The composite sequence matches at any clock tick on which at least one of the operand sequences matches. As shown in Figure 17-10, for the attempt at clock tick 8, the first operand sequence matches at clock ticks 9, 10, 11, 12, and 13, while the second operand matches at clock tick 12. The composite sequence therefore has one match at each of clock ticks 9, 10, 11, and 13 and has two matches at clock tick 12.

 

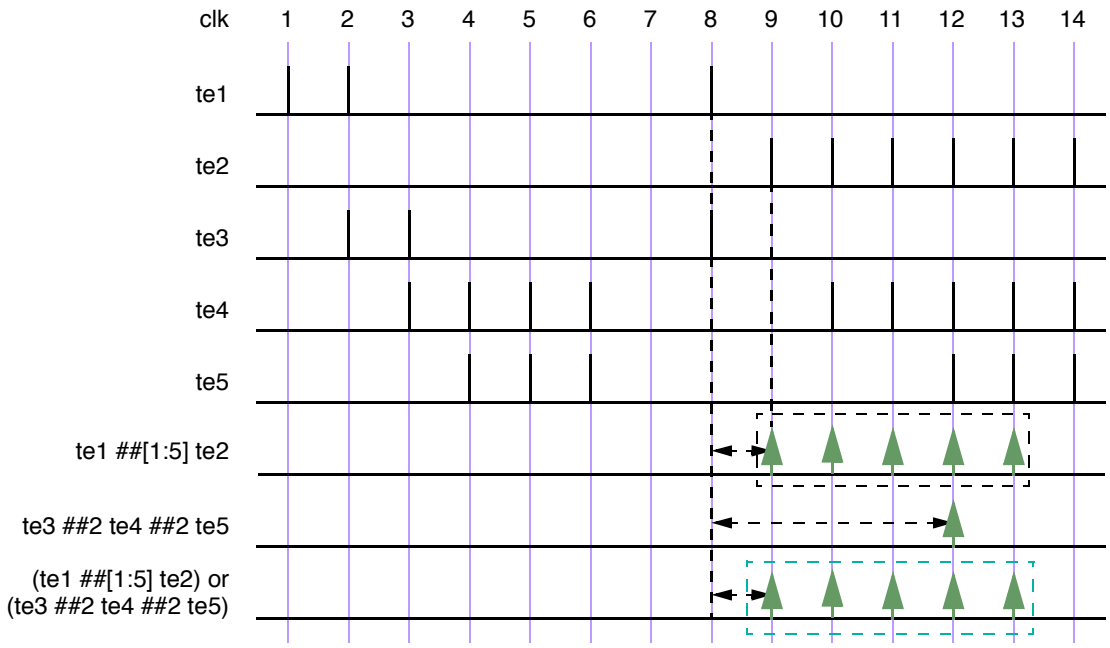


**Figure 17-10 — ORing (or) two sequences, including a time range**

### 17.7.7 first_match operation

The **`first_match`** operator matches only the first of possibly multiple matches for an evaluation attempt of its operand sequence. This allows all subsequent matches to be discarded from consideration. In particular, when a sequence is a subsequence of a larger sequence, then applying the **`first_match`** operator has significant effect on the evaluation of the enclosing sequence.

```
sequence_expr ::=                                          // from Annex A.2.10
    ...
  | first_match ( sequence_expr {, sequence_match_item} )
```

*Syntax 17-9—first_match operator syntax (excerpt from Annex A)*

An evaluation attempt of **`first_match`** (*seq*) results in an evaluation attempt for the operand *seq* beginning at the same clock tick. If the evaluation attempt for *seq* produces no match, then the evaluation attempt for **`first_match`** (*seq*) produces no match. Otherwise, the match of *seq* with earliest ending clock tick is a match of **`first_match`** (*seq*). If there are multiple matches of *seq* with the same ending clock tick as the earliest one, then all those matches are matches of **`first_match`** (*seq*).

The example below shows a variable delay specification.

```
sequence t1;
   te1 ## [2:5] te2;
endsequence
sequence ts1;
   first_match(te1 ## [2:5] te2);
endsequence
```

Here, `te1` and `te2` are expressions. Each attempt of sequence `t1` can result in matches for up to four of the following sequences:

```
te1 ##2 te2
te1 ##3 te2
te1 ##4 te2
te1 ##5 te2
```

However, sequence `ts1` can result in a match for only one of the above four sequences. Whichever match of the above four sequences ends first is a match of sequence `ts1`.

As another example:

```
sequence t2;
   (a ##[2:3] b) or (c ##[1:2] d);
endsequence
sequence ts2;
   first_match(t2);
endsequence
```

Each attempt of sequence `t2` can result in matches for up to four of the following sequences:

```
a ##2 b
a ##3 b
c ##1 d
c ##2 d
```

Sequence `ts2` matches only the earliest ending match of these sequences. If `a`, `b`, `c`, and `d` are expressions, then it is possible to have matches ending at the same time for both.

```
a ##2 b
c ##2 d
```

If both of these sequences match and (`c ##1 d`) does not match, then evaluation of `ts2` results in these two matches.

Sequence match items can be attached to the operand sequence of the **`first_match`** operator. The sequence match items are placed within the same set of parentheses that encloses the operand. Thus, for example, the local variable assignment `x = e` can be attached to the first match of `seq` via

```
first_match(seq, x = e)
```

which is equivalent to

```
first_match((seq, x = e))
```

See Sections 17.8 and 17.9 for discussion of sequence match items.

### 17.7.8 Conditions over sequences

Sequences often occur under the assumptions of some conditions for correct behavior. A logical condition must hold true, for instance, while processing a transaction. Also, occurrence of certain values is prohibited while processing a transaction. Such situations can be expressed directly using the following construct:

```
sequence_expr ::=                                    // from Annex A.2.10
      ...
    | expression_or_dist throughout sequence_expr
```

*Syntax 17-10—throughout construct syntax (excerpt from Annex A)*

 

The construct `exp` **`throughout`** `seq` is an abbreviation for:

```
(exp) [*0:$] intersect seq
```

The composite sequence, `exp` **`throughout`** `seq`, matches along a finite interval of consecutive clock ticks provided `seq` matches along the interval and `exp` evaluates to true at each clock tick of the interval.

The following example is illustrated in Figure 17-11.

```
sequence burst_rule1;
   @(posedge mclk)
      $fell(burst_mode) ##0
      (!burst_mode) throughout (##2 ((trdy==0)&&(irdy==0)) [*7]);
endsequence
```

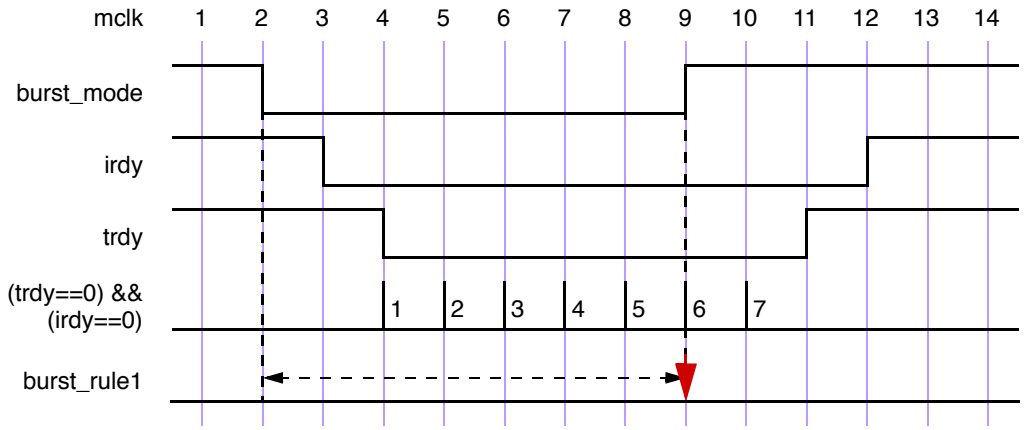


**Figure 17-11 — Match with throughout restriction fails**

Figure 17-12 illustrates the evaluation attempt for sequence `burst_rule1` beginning at clock tick 2. Since signal `burst_mode` is high at clock tick 1 and low at clock tick 2, `$fell(burst_mode)` is true at clock tick 2. To complete the match of `burst_rule1`, the value of `burst_mode` is required to be low throughout a match of the subsequence (`##2 ((trdy==0)&&(irdy==0)) [*7]`) beginning at clock tick 2. This subsequence matches from clock tick 2 to clock tick 10. However, at clock tick 9 `burst_mode` becomes high, thereby failing to match according to the rules for **`throughout`**.

If signal `burst_mode` were instead to remain low through at least clock tick 10, then there would be a match of `burst_rule1` from clock tick 2 to clock tick 10, as shown in Figure 17-12.

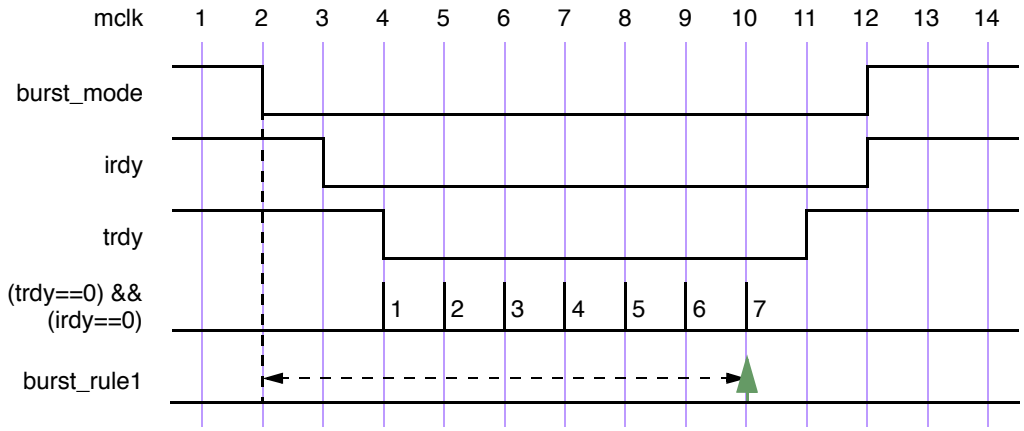

**Figure 17-12 — Match with throughout restriction succeeds**

### 17.7.9 Sequence contained within another sequence

The containment of a sequence within another sequence is expressed as follows:

```
sequence_expr ::=                                          // from Annex A.2.10
    ...
  | sequence_expr within sequence_expr
```

*Syntax 17-11—within construct syntax (excerpt from Annex A)*

The construct `seq1` **`within`** `seq2` is an abbreviation for:

```
(1[*0:$] ##1 seq1 ##1 1[*0:$]) intersect seq2
```

The composite sequence `seq1` **`within`** `seq2` matches along a finite interval of consecutive clock ticks provided `seq2` matches along the interval and `seq1` matches along some sub-interval of consecutive clock ticks. That is, the matches of `seq1` and `seq2` must satisfy the following:

— The start point of the match of `seq1` must be no earlier than the start point of the match of `seq2`.

— The end point of the match of `seq1` must be no later than the end point of the match of `seq2`.

For example, the sequence

```
!trdy[*7] within (($fell irdy) ##1 !irdy[*8])
```

matches from clock tick 3 to clock tick 11 on the trace shown in Figure 17-12.

### 17.7.10 Detecting and using endpoint of a sequence

There are two ways in which a complex sequence can be decomposed into simpler subsequences.

One is to instantiate a named sequence by referencing its name. Evaluation of such a reference requires the named sequence to match starting from the clock tick at which the reference is reached during the evaluation of the enclosing sequence. For example:

```
sequence s;
   a ##1 b ##1 c;
endsequence
sequence rule;
   @(posedge sysclk)
      trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
```

Sequence `s` is evaluated beginning one tick after the evaluation of `start_trans` in the sequence `rule`.

Another way to use a sequence is to detect its end point in another sequence. The end point of a sequence is reached whenever the ending clock tick of a match of the sequence is reached, regardless of the starting clock tick of the match. The reaching of the end point can be tested in any sequence by using the method `ended`.

The syntax of the ended method is:

```
sequence_instance.ended
```

`ended` is a method on a sequence. The result of its operation is true or false. When method `ended` is evaluated in an expression, it tests whether its operand sequence has reached its end point at that particular point in time.

 

The result of `ended` does not depend upon the starting point of the match of its operand sequence. An example is shown below:

```
sequence e1;
   @(posedge sysclk) $rose(ready) ##1 proc1 ##1 proc2 ;
endsequence
sequence rule;
   @(posedge sysclk) reset ##1 inst ##1 e1.ended ##1 branch_back;
endsequence
```

In this example, sequence `e1` must match one clock tick after `inst`. If the method `ended` is replaced with an instance of sequence `e1`, a match of `e1` must start one clock tick after `inst`. Notice that method `ended` only tests for the end point of `e1`, and has no bearing on the starting point of `e1`. `ended` can be used on sequences that have formal arguments. For example with the declarations

```
sequence e2(a,b,c);
   @(posedge sysclk) $rose(a) ##1 b ##1 c;
endsequence
sequence rule2;
   @(posedge sysclk) reset ##1 inst ##1 e2(ready,proc1,proc2).ended
      ##1 branch_back;
endsequence
```

`rule2` is equivalent to `rule2a` below:

```
sequence e2_instantiated;
  e2(ready,proc1,proc2);
endsequence
sequence rule2a;
  @(posedge sysclk) reset ##1 inst ##1 e2_instantiated.ended ##1 branch_back;
endsequence
```

There are additional restrictions on passing local variables into an instance of a sequence to which `ended` is applied. See Section 17.8.

## 17.8 Manipulating data in a sequence

The use of a static SystemVerilog variable implies that only one copy exists. If data values need to be checked in pipelined designs, then for each quantum of data entering the pipeline, a separate variable can be used to store the predicted output of the pipeline for later comparison when the result actually exits the pipe. This storage can be built by using an array of variables arranged in a shift register to mimic the data propagating through the pipeline. However, in more complex situations where the latency of the pipe is variable and out of order, this construction could become very complex and error prone. Therefore, variables are needed that are local to and are used within a particular transaction check that can span an arbitrary interval of time and can overlap with other transaction checks. Such a variable must thus be dynamically created when needed within an instance of a sequence and removed when the end of the sequence is reached.

The dynamic creation of a variable and its assignment is achieved by using the local variable declaration in a sequence or property declaration and making an assignment in the sequence.

```
sequence_expr ::=                                          // from Annex A.2.10
        ...
      | ( expression_or_dist {, sequence_match_item } ) [ boolean_abbrev ]
      | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
        ...
```

*Syntax 17-12—variable assignment syntax (excerpt from Annex A)*

The type of variable is explicitly specified. The variable can be assigned at the end point of any syntactic subsequence by placing the subsequence, comma separated from the sampling assignment, in parentheses. For example, if in

```
a ##1 b[->1] ##1 c[*2]
```

it is desired to assign `x = e` at the match of `b[->1]`, the sequence can be rewritten as

```
a ##1 (b[->1], x = e) ##1 c[*2]
```

The local variable can be reassigned later in the sequence, as in

```
a ##1 (b[->1], x = e) ##1 (c[*2], x = x + 1)
```

For every attempt, a new copy of the variable is created for the sequence. The variable value can be tested like any other SystemVerilog variable.

Hierarchical references to a local variable are not allowed.

As an example of local variable usage, assume a pipeline that has a fixed latency of 5 clock cycles. The data enters the pipe on `pipe_in` when `valid_in` is true, and the value computed by the pipeline appears 5 clock cycles later on the signal `pipe_out1`. The data as transformed by the pipe is predicted by a function that increments the data. The following property verifies this behavior:

```
property e;
   int x;
   (valid_in,(x = pipe_in)) |-> ##5 (pipe_out1 == (x+1));
endproperty
```

Property `e` is evaluated as :

1) When `valid_in` is true, `x` is assigned the value of `pipe_in`. If five cycles later, `pipe_out1` is equal to `x+1`, then property e is true. Otherwise, property e is false.

2) When is `valid_in` false, property e evaluates to true.

Variables can be used in sequences or properties.

```
sequence data_check;
   int x;
   a ##1 !a, x = data_in ##1 !b[*0:$] ##1 b && (data_out == x);
endsequence
property data_check_p
   int x;
   a ##1 !a, x = data_in |=> !b[*0:$] ##1 b && (data_out == x);
endproperty
```

Local variables can be written on repeated sequences and accomplish accumulation of values.

```
sequence rep_v;
   int x;
   `true,x = 0 ##0
   (!a [* 0:$] ##1 a, x = x+data)[*4] ##1 b ##1 c && (data_out == x);
endsequence
```

The local variables declared in one sequence are not visible in the sequence where it gets instantiated. An example below illustrates an illegal access to local variable `v1` of sequence `sub_seq1` in sequence `seq1`.

```
sequence sub_seq1;
   int v1;
```

 

```
   a ##1 !a, v1 = data_in ##1 !b[*0:$] ##1 b && (data_out == v1);
endsequence
sequence seq1;
   c ##1 sub_seq1 ##1 (do1 == v1); // error since v1 is not visible
endsequence
```

To access a local variable of a subsequence, a local variable must be declared and passed to the instantiated subsequence through an argument. An example below illustrates this usage.

```
sequence sub_seq2(lv);
   a ##1 !a, lv = data_in ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
sequence seq2;
   int v1;
   c ##1 sub_seq2(v1) ##1 (do1 == v1); // v1 is now bound to lv
endsequence
```

Local variables can be passed into an instance of a named sequence to which `ended` is applied and accessed in a similar manner. For example

```
sequence seq2a;
   int v1; c ##1 sub_seq2(v1).ended ##1 (do1 == v1); // v1 is now bound to lv
endsequence
```

There are additional restrictions when passing local variables into an instance of a named sequence to which `ended` is applied:

1) Local variables can be passed in only as entire actual arguments, not as proper subexpressions of actual arguments.

2) In the declaration of the named sequence, the formal argument to which the local variable is bound must not be referenced before it is assigned.

The second restriction is met by `sub_seq2` because the assignment `lv = data_in` occurs before the reference to `lv` in `data_out == lv`.

If a local variable is assigned before being passed into an instance of a named sequence to which `ended` is applied, then the restrictions prevent this assigned value from being visible within the named sequence. The restrictions are important because the use of `ended` means that there is no guaranteed relationship between the point in time at which the local variable is assigned outside the named sequence and the beginning of the match of the instance.

A local variable that is passed in as actual argument to an instance of a named sequence to which `ended` is applied will flow out of the application of ended to that instance provided both of the following conditions are met:

1) The local variable flows out of the end of the named sequence instance, as defined by the local variable flow rules for sequences. (See below and Annex H.)

2) The application of `ended` to this instance is a maximal boolean expression. In other words, the application of `ended` cannot have negation or any other expression operator applied to it.

Both conditions are satisfied by `sub_seq2` and `seq2a`. Thus, in `seq2a` the value in `v1` in the comparison `do1 == v1` is the value assigned to `lv` in `sub_seq2` by the assignment `lv = data_in`. However, in

```
sequence seq2b;
   int v1; c ##1 !sub_seq2(v1).ended ##1 (do1 == v1); // v1 unassigned
endsequence
```

the second condition is violated because of the negation applied to `sub_seq2(v1).ended`. Therefore, `v1`

does not flow out of the application of `ended` to this instance, and so the reference to `v1` in `do1 == v1` is to an unassigned variable.

In a single cycle, there can be multiple matches of a sequence instance to which `ended` is applied, and these matches can have different valuations of the local variables. The multiple matches are treated semantically the same way as matching both disjuncts of an **`or`** (see below). In other words, the thread evaluating the instance to which `ended` is applied will fork to account for such distinct local variable valuations.

Note that when a local variable is a formal argument of a sequence declaration, it is illegal to declare the variable, as shown below.

```
sequence sub_seq3(lv);
   int lv; // illegal since lv is a formal argument
   a ##1 !a, lv = data_in ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
```

There are special considerations when using local variables in sequences involving the branching operators **`or`**, **`and`**, and **`intersect`**. The evaluation of a composite sequence constructed from one of these operators can be thought of as forking two threads to evaluate the operand sequences in parallel. A local variable may have been assigned a value before the start of the evaluation of the composite sequence. Such a local variable is said to flow in to each of the operand sequences. The local variable may be assigned or reassigned in one or both of the operand sequences. In general, there is no guarantee that evaluation of the two threads results in consistent values for the local variable, or even that there is a consistent view of whether the local variable has been assigned a value. Therefore, the values assigned to the local variable before and during the evaluation of the composite sequence are not always allowed to be visible after the evaluation of the composite sequence.

In some cases, inconsistency in the view of the local variable's value does not matter, while in others it does. Precise conditions are given in Annex H to define static (i.e., compile-time computable) conditions under which a sufficiently consistent view of the local variable's value after the evaluation of the composite sequence is guaranteed. If these conditions are satisfied, then the local variable is said to flow out of the composite sequence. An intuitive description of the conditions for local variable flow follows.

1) Variables assigned on parallel threads cannot be accessed in sibling threads. For example:

```
sequence s4;
    int x;
    (a ##1 b, (x = data) ##1 c) or (d ##1 (e==x)); // illegal
endsequence
```

2) In the case of **`or`**, a local variable flows out of the composite sequence if and only if it flows out of each of the operand sequences. If the local variable is not assigned before the start of the composite sequence and it is assigned in only one of the operand sequences, then it does not flow out of the composite sequence.

3) Each thread for an operand of an **`or`** that matches its operand sequence continues as a separate thread, carrying with it its own latest assignments to the local variables that flow out of the composite sequence. These threads do not have to have consistent valuations for the local variables. For example:

```
sequence s5;
   int x,y;
   ((a ##1 b, x = data, y = data1 ##1 c)
      or (d ##1 `true, x = data ##0 (e==x))) ##1 (y==data2);
   // illegal since y is not in the intersection
endsequence
sequence s6;
   int x,y;
   ((a ##1 b, x = data, y = data1 ##1 c)
      or (d ##1 `true, x = data ##0 (e==x))) ##1 (x==data2);
   // legal since x is in the intersection
endsequence
```

 

4) In the case of **and** and **intersect**, a local variable that flows out of at least one operand shall flow out of the composite sequence unless it is blocked. A local variable is blocked from flowing out of the composite sequence if either:

   a) The local variable is assigned in and flows out of each operand of the composite sequence. Or,

   b) The local variable is blocked from flowing out of at least one of the operand sequences.

   The value of a local variable that flows out of the composite sequence is the latest assigned value. The threads for the two operands are merged into one at completion of evaluation of the composite sequence.

```
sequence s7;
   int x,y;
   ((a ##1 b, x = data, y = data1 ##1 c)
      and (d ##1 `true, x = data ##0 (e==x))) ##1 (x==data2);
   // illegal since x is common to both threads
endsequence
sequence s8;
   int x,y;
   (a ##1 b, x = data, y = data1 ##1 c)
      and (d ##1 `true, x = data ##0 (e==x))) ##1 (y==data2);
   // legal since y is in the difference
endsequence
```

## 17.9 Calling subroutines on match of a sequence

Tasks, task methods, void functions, void function methods, and system tasks can be called at the end of successful match of a sequence. The subroutine calls, like local variable assignments, appear in the comma-separated list that follows the sequence. The subroutine calls are said to be *attached* to the sequence. The sequence and the list that follows are enclosed in parentheses.

```
sequence_expr ::=                                          // from Annex A.2.10
        ...
      | ( expression_or_dist {, sequence_match_item } ) [ boolean_abbrev ]
      | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
        ...
sequence_match_item ::=
        operator_assignment
      | inc_or_dec_expression
      | subroutine_call
```

*Syntax 17-13—subroutine call in sequence syntax (excerpt from Annex A)*

For example,

```
sequence s1;
   logic v, w;
   (a, v = e) ##1
   (b[->1], w = f, $display("b after a with v = %h, w = %h\n", v, w));
endsequence
```

defines a sequence `s1` that matches at the first occurrence of `b` strictly after an occurrence of `a`. At the match, the system task $display is executed to write a message that announces the match and shows the values assigned to the local variables `v` and `w`.

All subroutine calls attached to a sequence are executed at every successful match of the sequence. For each

successful match, the attached calls are executed in the order they appear in the list. The subroutines are scheduled in the Reactive region, like an action block.

Each argument of a subroutine call attached to a sequence must either be passed by value as an input or be passed by reference (either **ref** or **const ref**; see Section 10.4.2). Actual argument expressions that are passed by value use sampled values of the underlying variables and are consistent with the variable values used to evaluate the sequence match.

Local variables can be passed into subroutine calls attached to a sequence. Any local variable that flows out of the sequence or that is assigned in the list following the sequence, but before the subroutine call, can be used in an actual argument expression for the call. If a local variable appears in an actual argument expression, then that argument must be passed by value.

## 17.10 System functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is “one-hot”. The following system functions are included to facilitate such common assertion functionality:

— `$onehot (<expression>)` returns true if only one bit of the expression is high.

— `$onehot0(<expression>)` returns true if at most one bit of the expression is high.

— `$isunknown (<expression>)` returns true if any bit of the expression is X or Z. This is equivalent to `^<expression> === 'bx`.

All of the above system functions have a return type of bit. A return value of `1'b1` indicates true, and a return value of `1'b0` indicates false.

Another useful function provided for the boolean expression is `$countones`, to count the number of 1s in a bit vector expression.

```
$countones ( expression)
```

An X and Z value of a bit is not counted towards the number of ones.

## 17.11 Declaring properties

A property defines a behavior of the design. A property can be used for verification as an assumption, a checker, or a coverage specification. In order to use the behavior for verification, an **assert**, **assume** or **cover** statement must be used. A property declaration by itself does not produce any result.

A property can be declared in

— a module

— an interface a program

— a clocking block

— a package

— a compilation-unit scope

To declare a property, the **property** construct is used as shown below:

 

```
concurrent_assertion_item_declaration ::=                    // from Annex A.2.10
        property_declaration
        ...
property_declaration ::=
        property property_identifier [ ( [ list_of_formals ] ) ] ;
            { assertion_variable_declaration }
            property_spec ;
        endproperty [ : property_identifier ]
list_of_formals ::= formal_list_item { , formal_list_item }
property_spec ::=
        [clocking_event ] [ disable iff ( expression_or_dist ) ] property_expr
property_expr ::=
        sequence_expr
      | ( property_expr )
      | not property_expr
      | property_expr or property_expr
      | property_expr and property_expr
      | sequence_expr |-> property_expr
      | sequence_expr |=> property_expr
      | if ( expression_or_dist ) property_expr [ else property_expr ]
      | property_instance
      | clocking_event property_expr
assertion_variable_declaration ::=
        data_type list_of_variable_identifiers ;
property_instance::=
        ps_property_identifier [ ( [ actual_arg_list ] ) ]
```

*Syntax 17-14—property construct syntax (excerpt from Annex A)*

A **`property`** is declared with optional formal arguments, as in a sequence declaration. When a property is instantiated, actual arguments can be passed to the property. The property gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. Semantic checks are performed to ensure that the expanded property with the actual arguments is legal.

The result of property evaluation is either true or false. There are seven kinds of property: sequence, negation, disjunction, conjunction, **`if`**...**`else`**, implication, and instantiation.

1) A property that is a sequence evaluates to true if and only if there is a non-empty match of the sequence. A sequence that admits an empty match is not allowed as a property. Since there is a match if and only if there is a first match, evaluation of such a property is the same as implicitly transforming its *sequence_expr* to `first_match(`*sequence_expr*`)`. As soon as a match of *sequence_expr* is determined, the evaluation of the property is considered to be true, and no other matches are required for that evaluation attempt.

2) A property is a negation if it has the form

```
not property_expr
```

   For each evaluation attempt of the property, there is an evaluation attempt of *property_expr*. The keyword **`not`** states that the evaluation of the property returns the opposite of the evaluation of the underlying *property_expr*. Thus, if *property_expr* evaluates to true, then **not** *property_expr* evaluates to false, and if *property_expr* evaluates to false, then **`not`** *property_expr* evaluates to true.

3) A property is a disjunction if it has the form

```
property_expr1 or property_expr2
```

The property evaluates to true if and only if at least one of *property_expr1* and *property_expr2* evaluates to true.

4) A property is a conjunction if it has the form

```
property_expr1 and property_expr2
```

The property evaluates to true if and only if both *property_expr1* and *property_expr2* evaluate to true.

5) A property is an **if**...**else** if it has either the form

```
if (expression_or_dist) property_expr1
```

or the form

```
if (expression_or_dist) property_expr1 else property_expr2
```

A property of the first form evaluates to true if and only if either *expression_or_dist* evaluates to false or *property_expr1* evaluates to true. A property of the second form evaluates to true if and only if either *expression_or_dist* evaluates to true and *property_expr1* evaluates to true or *expression_or_dist* evaluates to false and *property_expr2* evaluates to true.

6) A property is an implication if it has either the form

```
sequence_expr |-> property_expr
```

or the form

```
sequence_expr |=> property_expr
```

The meaning of implications is discussed in 17.11.1.

7) An instance of a named property can be used as a *property_expr* or *property_spec*. In general, the instance is legal provided the body *property_spec* of the named property can be substituted in place of the instance, with actual arguments substituted for formal arguments, and result in a legal *property_expr* or *property_spec*, ignoring local variable declarations. Thus, for example, if an instance of a named property is used as a *property_expr* operand for any property-building operator, then the named property must not have a **disable iff** clause. Similarly, clock events in a named property must conform to the rules of multiple clock support when the property is instantiated in a *property_expr* or *property_spec* that also involves other clock events.

The following table lists the property operators from highest to lowest precedence and shows the associativity of the non-unary operators.

**Table 17-2: Property operator precedence and associativity**

| SystemVerilog property operators | Associativity |
|---|---|
| **not** | ---- |
| **and** | left |
| **or** | left |
| **if**...**else** | right |
| **\|-> \|=>** | right |

 

A **disable iff** clause can be attached to a *property_expr* to yield a *property_spec*

```
disable iff (expression_or_dist) property_expr
```

The expression of the **disable iff** is called the reset expression. The **disable iff** clause allows asynchronous resets to be specified. For an evaluation of the *property_spec*, there is an evaluation of the underlying *property_expr*. If prior to the completion of that evaluation the reset expression becomes true, then the overall evaluation of the *property_spec* is true. Otherwise, the evaluation of the *property_spec* is the same as that of the *property_expr*. The reset expression is tested independently for different evaluation attempts of the *property_spec*. Nesting of **disable iff** clauses, explicitly or through property instantiations, is not allowed.

### 17.11.1 Implication

The implication construct specifies that the checking of a property is performed conditionally on the match of a sequential antecedent.

```
property_expr ::=                                          // from Annex A.2.10
    ...
  | sequence_expr |-> property_expr
  | sequence_expr |=> property_expr
```

*Syntax 17-15—implication syntax (excerpt from Annex A)*

This clause is used to precondition monitoring of a property expression and is allowed at the property level. The result of the implication is either true or false. The left-hand side operand *sequence_expr* is called the *antecedent*, while the right-hand side operand *property_expr* is called the *consequent*.

The following points should be noted for `|->` implication:

— From a given start point, the antecedent *sequence_expr* can have zero, one, or more than one successful match.

— If there is no match of the antecedent *sequence_expr* from a given start point, then evaluation of the implication from that start point succeeds vacuously and returns true.

— For each successful match of antecedent *sequence_expr*, the consequent *property_expr* is separately evaluated. The end point of the match of the antecedent *sequence_expr* is the start point of the evaluation of the consequent *property_expr*.

— From a given start point, evaluation of the implication succeeds and returns true if and only if for every match of the antecedent *sequence_expr* beginning at the start point, the evaluation of the consequent *property_expr* beginning at the endpoint of the match succeeds and returns true.

Two forms of implication are provided: overlapped using operator `|->`, and non-overlapped using operator `|=>`. For overlapped implication, if there is a match for the antecedent *sequence_expr*, then the end point of the match is the start point of the evaluation of the consequent *property_expr*. For non-overlapped implication, the start point of the evaluation of the consequent *property_expr* is the clock tick after the end point of the match. Therefore:

```
sequence_expr |=> property_expr
```

is equivalent to:

```
sequence_expr ##1 'true |-> property_expr
```

The use of implication when multi-clock sequences and properties are involved is explained in Section 17.12.

The following example illustrates a bus operation for data transfer from a master to a target device. When the

bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which `irdy` is asserted and either `trdy` or `stop` is asserted. Note that an asserted signal here implies a value of low. The end of a data phase can be expressed as:

```
property data_end;
   @(posedge mclk)
   data_phase |-> ((irdy==0) && ($fell(trdy) || $fell(stop))) ;
endproperty
```

Each time a data phase is true, a match for `data_phase` is recognized. The attempt at clock tick 6 is illustrated in Figure 17-13. The values shown for the signals are the sampled values with respect to the clock. At clock tick 6, `data_end` is true because `stop` gets asserted while `irdy` is asserted.

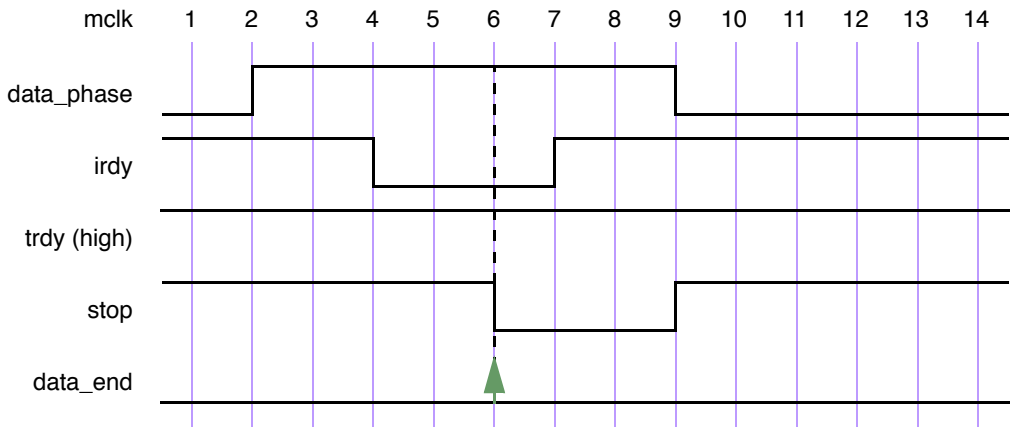


**Figure 17-13 — Conditional sequence matching**

In another example, `data_end_exp` is used to ensure that `frame` is de-asserted (value high) within 2 clock ticks after `data_end_exp` occurs. Further, it is also required that `irdy` is de-asserted (value high) one clock tick after `frame` is de-asserted.

A property written to express this condition is shown below.

```
`define data_end_exp (data_phase && ((irdy==0)&&($fell(trdy)||$fell(stop))))
property data_end_rule1;
   @(posedge mclk)
   `data_end_exp |-> ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```

property `data_end_rule1` first evaluates `data_end_exp` at every clock tick to test if its value is true. If the value is false, then that particular attempt to evaluate `data_end_rule1` is considered true. Otherwise, the following sequence is evaluated. The sequence:

```
##[1:2] $rose(frame) ##1 $rose(irdy)
```

specifies looking for the rising edge of `frame` within two clock ticks in the future. After `frame` toggles high, `irdy` must also toggle high after one clock tick. This is illustrated in Figure 17-14 for the evaluation attempt at clock tick 6. `` `data_end_exp `` is acknowledged at clock tick 6. Next, `frame` toggles high at clock tick 7. Since this falls within the timing constraint imposed by `[1:2]`, it satisfies the sequence and continues to evaluate further. At clock tick 8, `irdy` is evaluated. Signal `irdy` transitions to high at clock tick 8, matching the sequence specification completely for the attempt that began at clock tick 6.

 

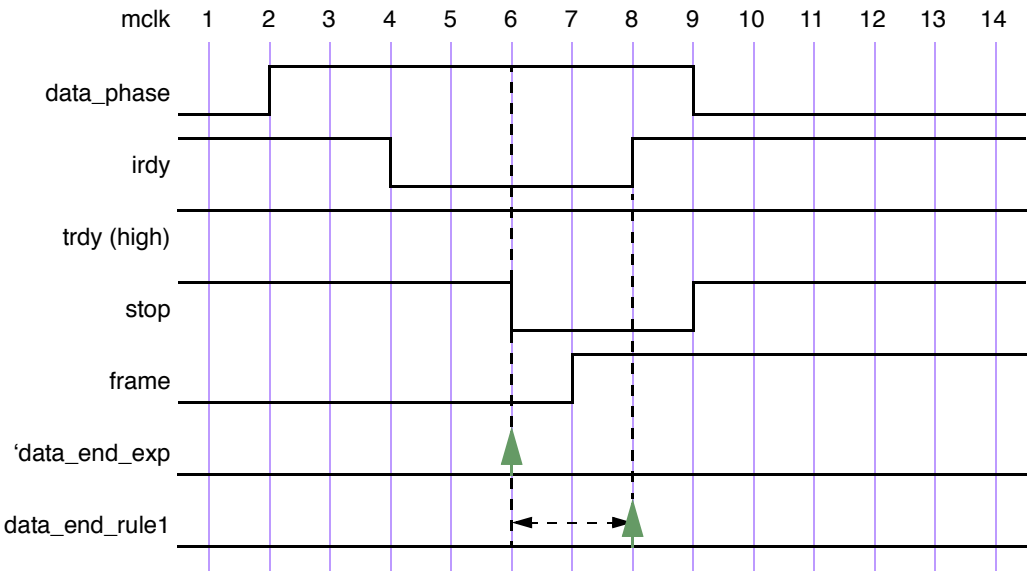


**Figure 17-14 — Conditional sequences**

Generally, assertions are associated with preconditions so that the checking is performed only under certain specified conditions. As seen from the previous example, the `|->` operator provides this capability to specify preconditions with sequences that must be satisfied before evaluating their consequent properties. The next example modifies the preceding example to see the effect on the results of the assertion by removing the precondition for the consequent. This is shown below, and illustrated in Figure 17-15.

```
property data_end_rule2;
   @(posedge mclk) ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```

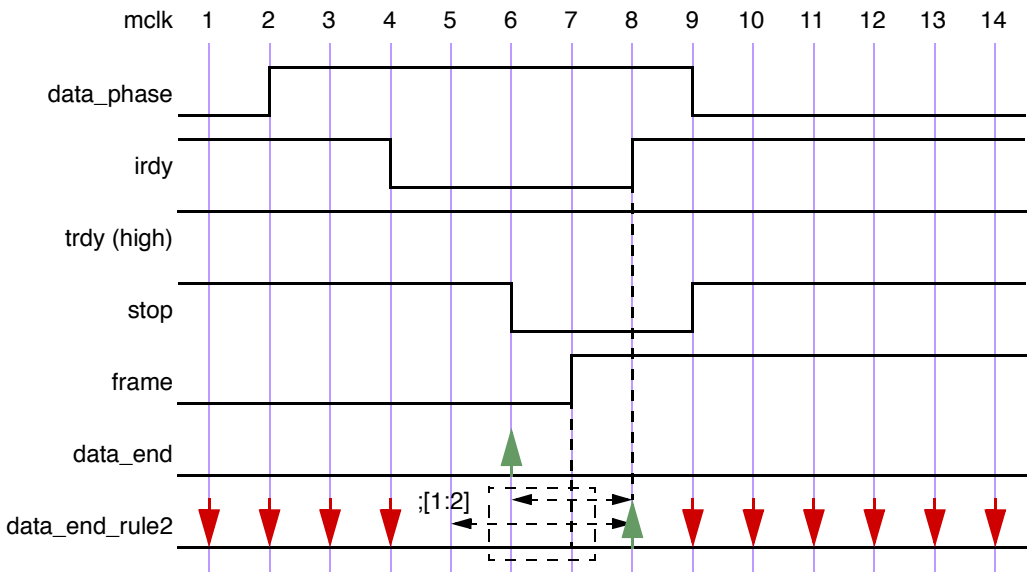


**Figure 17-15 — Results without the condition**

The property is evaluated at every clock tick. For the evaluation at clock tick 1, the rising edge of signal `frame`

does not occur at clock tick 1 or 2, so the property fails at clock tick 1. Similarly, there is a failure at clock ticks 2, 3, and 4. For attempts starting at clock ticks 5 and 6, the rising edge of signal `frame` at clock tick 7 allows checking further. At clock tick 8, the sequences complete according to the specification, resulting in a match for attempts starting at 5 and 6. All later attempts to match the sequence fail because `$rose(frame)` does not occur again.

Figure 17-15 shows that removing the precondition of checking `` `data_end_exp `` from the assertion causes failures that are not relevant to the verification objective. It is important from the validation standpoint to determine these preconditions and use them to filter out inappropriate or extraneous situations.

An example of implication where the antecedent is a sequence follows:

```
(a ##1 b ##1 c) |-> (d ##1 e)
```

If the sequence `(a ##1 b ##1 c)` matches, then the sequence `(d ##1 e)` must also match. On the other hand, if the sequence `(a ##1 b ##1 c)` does not match, then the result is true.

Another example of implication is:

```
property p16;
   (write_en & data_valid) ##0
   (write_en && (retire_address[0:4]==addr)) [*2] |->
   ##[3:8] write_en && !data_valid &&(write_address[0:4]==addr);
endproperty
```

This property can be coded alternatively as a nested implication:

```
property p16_nested;
   (write_en & data_valid) |->
      (write_en && (retire_address[0:4]==addr)) [*2] |->
         ##[3:8] write_en && !data_valid && (write_address[0:4]==addr);
endproperty
```

Multi-clock sequence implication is explained in Section 17.12.

### 17.11.2 Property examples

The following examples illustrate the property forms.

```
property rule1;
   @(posedge clk) a |-> b ##1 c ##1 d;
endproperty
property rule2;
   @(clkev) disable iff (foo) a |-> not(b ##1 c ##1 d);
endproperty
```

Property `rule2` negates the `sequence (b ##1 c ##1 d) in the consequent of the implication. clkev` specifies the clock for the property .

```
property rule3;
   @(posedge clk) a[*2] |-> ((##[1:3] c) or (d |=> e));
endproperty
```

Property `rule3` says that if `a` holds and `a` also held last cycle, then either `c` must hold at some point 1 to three cycles after the current cycle, or, if `d` holds in the current cycle, then `e` must hold one cycle later.

```
property rule4;
   @(posedge clk) a[*2] |-> ((##[1:3] c) and (d |=> e));
endproperty
```

 

Property `rule4` says that if `a` holds and `a` also held last cycle, then `c` must hold at some point 1 to three cycles after the current cycle, and if `d` holds in the current cycle, then `e` must hold one cycle later.

```
property rule5;
   @(posedge clk)
   a ##1 (b || c)[->1] |->
      if (b)
         (##1 d |-> e)
      else // c
         f ;
endproperty
```

Property `rule5` has `a` followed by the first match of either `b` or `c` as its antecedent. The consequent uses **`if`**...**`else`** to split cases on which of `b` or `c` is matched first.

```
property rule6(x,y);
   ##1 x |-> y;
endproperty
property rule5a;
   @(posedge clk)
   a ##1 (b || c)[->1] |->
      if (b)
         rule6(d,e)
      else // c
         f ;
endproperty
```

Property `rule5a` is equivalent to `rule5`, but it uses an instance of `rule6` as a property expression.

A property can optionally specify an event control for the clock. The clock derivation and resolution rules are described in Section 17.14.

A named property can be instantiated by referencing its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. Like sequence declarations, variables used within a property that are not formal arguments to the property are resolved hierarchically from the scope in which the property is declared.

Properties that use more than one clock are described in Section 17.12

### 17.11.3 Recursive properties

SystemVerilog allows recursive properties. A named property is recursive if its declaration involves an instantiation of itself. Recursion provides a flexible framework for coding properties to serve as ongoing assumptions, checkers, or coverage monitors.

For example,

```
property prop_always(p);
   p and (1'b1 |=> prop_always(p));
endproperty
```

is a recursive property that says that the formal argument property `p` must hold at every cycle. This example is useful if the ongoing requirement that property `p` hold applies after a complicated triggering condition encoded in sequence `s`:

```
property p1(s,p);
   s |=> prop_always(p);
endproperty
```

As another example, the recursive property

```
property prop_weak_until(p,q);
   q or (p and (1'b1 |=> prop_weak_until(p,q)));
endproperty
```

says that formal argument property p must hold at every cycle up to but not including the first cycle at which formal argument property q holds. Formal argument property q is not required ever to hold, though. This example is useful if p must hold at every cycle after a complicated triggering condition encoded in sequence s, but the requirement on p is lifted by q:

```
property p2(s,p,q);
   s |=> prop_weak_until(p,q);
endproperty
```

More generally, several properties can be mutually recursive. For example

```
property check_phase1;
   s1 |-> (phase1_prop and (1'b1 |=> check_phase2));
endproperty
property check_phase2;
   s2 |-> (phase2_prop and (1'b1 |=> check_phase1));
endproperty
```

There are three restrictions on recursive property declarations.

RESTRICTION 1: The negation operator **not** cannot be applied to any property expression that instantiates a recursive property. In particular, the negation of a recursive property cannot be asserted or used in defining another property.

Here are examples of illegal property declarations that violate Restriction 1:

```
property illegal_recursion_1(p);
   not prop_always(not p);
endproperty

property illegal_recursion_2(p);
   p and (1'b1 |=> not illegal_recursion_2(p));
endproperty
```

RESTRICTION 2: The operator **disable iff** cannot be used in the declaration of a recursive property. This restriction is consistent with the restriction that **disable iff** cannot be nested.

Here is an example of an illegal property declaration that violates Restriction 2:

```
property illegal_recursion_3(p);
   disable iff (b)
   p and (1'b1 |=> illegal_recursion_3(p));
endproperty
```

The intent of `illegal_recursion_3` can be written legally as

```
property legal_3(p);
   disable iff (b) prop_always(p);
endproperty
```

since `legal_3` is not a recursive property.

RESTRICTION 3: If `p` is a recursive property, then, in the declaration of `p`, every instance of `p` must occur

 

after a positive advance in time. In the case of mutually recursive properties, all recursive instances must occur after positive advances in time.

Here is an example of an illegal property declaration that violates Restriction 3:

```
property illegal_recursion_4(p);
   p and (1'b1 |-> illegal_recursion_4(p));
endproperty
```

If this form were legal, the recursion would be stuck in time, checking `p` over and over again at the same cycle.

Recursive properties can represent complicated requirements, such as those associated with varying numbers of data beats, out-of-order completions, retries, etc. Here is an example of using a recursive property to check complicated conditions of this kind.

EXAMPLE: Suppose that write data must be checked according to the following conditions:

— Acknowledgment of a write request is indicated by the signal `write_request` together with `write_request_ack`. When a write request is acknowledged, it gets a 4-bit tag, indicated by signal `write_reqest_ack_tag`. The tag is used to distinguish data beats for multiple write transactions in flight at the same time.

— It is understood that distinct write transactions in flight at the same time must be given distinct tags. For simplicity, this condition is not a part of what is checked in this example.

— Each write transaction can have between 1 and 16 data beats, and each data beat is 8 bits. There is a model of the expected write data that is available at acknowledgment of a write request. The model is a 128-bit vector. The most significant group of 8 bits represents the expected data for the first beat, the next group of 8 bits represents the expected data for the second beat (if there is a second beat), and so forth.

— Data transfer for a write transaction occurs after acknowledgment of the write request and, barring retry, ends with the last data beat. The data beats for a single write transaction occur in order.

— A data beat is indicated by the `data_valid` signal together with the signal `data_valid_tag` to determine the relevant write transaction. The signal `data` is valid with `data_valid` and carries the data for that beat. The data for each beat must be correct according to the model of the expected write data.

— The last data beat is indicated by signal `last_data_valid` together with `data_valid` and `data_valid_tag`. For simplicity, this example does not represent the number of data beats and does not check that `last_data_valid` is signaled at the correct beat.

— At any time after acknowledgement of the write request, but not later than the cycle after the last data beat, a write transaction can be forced to retry. Retry is indicated by the signal `retry` together with signal `retry_tag` to identify the relevant write transaction. If a write transaction is forced to retry, then its current data transfer is aborted and the entire data transfer must be repeated. The transaction does not re-request and its tag does not change.

— There is no limit on the number of times a write transaction can be forced to retry.

— A write transaction completes the cycle after the last data beat provided it is not forced to retry in that cycle.

Here is code to check these conditions:

```
property check_write;

   logic [0:127] expected_data;  // local variable to sample model data
   logic [3:0] tag;              // local variable to sample tag

   disable iff (reset)
   (
```

```
      write_request && write_request_ack,
      expected_data = model_data,
      tag = write_request_ack_tag
   )
   |=>
   check_write_data_beat(expected_data, tag, 4'h0);

endproperty

property check_write_data_beat
(
   expected_data,  // [0:127]
   tag,            // [3:0]
   i               // [3:0]
);

   first_match
   (
      ##[0:$]
      (
         (data_valid && (data_valid_tag == tag))
         ||
         (retry && (retry_tag == tag))
      )
   )
   |->
   (
      (
         (data_valid && (data_valid_tag == tag))
         |->
         (data == expected_data[i*8+:8])
      )
      and
      (
         if (retry && (retry_tag == tag))
         (
            1'b1 |=> check_write_data_beat(tag, expected_data, 4'h0)
         )
         else if (!last_data_valid)
         (
            1'b1 |=> check_write_data_beat(tag, expected_data, i+4'h1)
         )
         else
         (
            ##1 (retry && (retry_tag == tag))
            |=>
            check_write_data_beat(tag, expected_data, 4'h0)
         )
      )
   );

endproperty
```

### 17.11.4 Finite-length versus infinite-length behavior

The formal semantics in Annex H defines whether a given property holds on a given behavior. How the outcome of this evaluation relates to the design depends on the behavior that was analyzed. In dynamic verification, only behaviors that are finite in length are considered. In such a case, SystemVerilog defines four levels

 

of satisfaction of a property:

Holds strongly:

— no bad states have been seen

— all future obligations have been met

— the property will hold on any extension of the path

Holds (but does not hold strongly):

— no bad states have been seen

— all future obligations have been met

— the property may or may not hold on a given extension of the path

Pending:

— no bad states have been seen

— future obligations have not been met

— the property may or may not hold on a given extension of the path

Fails:

— a bad state has been seen

— future obligations may or may not have been met

— the property will not hold on any extension of the path

### 17.11.5 Non-degeneracy

It is possible to define sequences that can never be matched. For example:

```
(1'b1) intersect(1'b1 ##1 1'b1)
```

It is also possible to define sequences that admit only empty matches. For example:

```
1'b1[*0]
```

A sequence that admits no match or that admits only empty matches is called *degenerate*. A sequence that admits at least one non-empty match is called *non-degenerate*. A more precise definition of non-degeneracy is given in Annex H.

The following restrictions apply:

1) Any sequence that is used as a property must be non-degenerate and must not admit any empty match.

2) Any sequence that is used as the antecedent of an overlapping implication (`|->`) must be non-degenerate.

3) Any sequence that is used as the antecedent of a non-overlapping implication (`|=>`) must admit at least one match. Such a sequence can admit only empty matches.

The reason for these restrictions is that the use of degenerate sequences the forbidden ways results in counter-intuitive property semantics, especially when the property is combined with a **`disable iff`** clause.

## 17.12 Multiple clock support

Multiple clock sequences and properties can be specified using the following syntax.

### 17.12.1 Multiply-clocked sequences

Multiply-clocked sequences are built by concatenating singly-clocked subsequences using the single-delay concatenation operator `##1`. This operator is non-overlapping and synchronizes between the clocks of the two sequences. The single delay indicated by ##1 is understood to be from the endpoint of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins.

For example, consider

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

A match of this sequence starts with a match of `sig0` at posedge `clk0`. Then `##1` moves the time to the nearest strictly subsequent posedge `clk1`, and the match of the sequence ends at that point with a match of `sig1`. If `clk0` and `clk1` are not identical, then the clocking event for the sequence changes after `##1`. If `clk0` and `clk1` are identical, then the clocking event does not change after `##1` and the above sequence is equivalent to the singly-clocked sequence

```
@(posedge clk0) sig0 ##1 sig1
```

When concatenating differently-clocked sequences, the maximal singly-clocked subsequences are required to admit only non-empty matches. Thus, if `s1`, `s2` are sequence expressions with no clocking events, then the multiply-clocked sequence

```
@(posedge clk1) s1 ##1 @(posedge clk2) s2
```

is legal only if neither `s1` nor `s2` can match the empty word. The clocking event **`posedge`** `clk1` applies throughout the match of `s1`, while the clocking event posedge `clk2` applies throughout the match of `s2`. Since the match of `s1` is non-empty, there is an end point of this match at posedge `clk1`. The `##1` synchronizes between this end point and the first occurrence of posedge `clk2` strictly after it. That occurrence of **`posedge`** `clk2` is the start point of the match of `s2`.

The restriction that maximal singly-clocked subsequences not match the empty word ensures that any multiply-clocked sequence has well-defined starting and ending clocking events and well-defined clock changes. If `clk1` and `clk2` are not identical, then the following sequence

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1[*0:1]
```

is illegal because of the possibility of an empty match of `sig1[*0:1]`, which would make ambiguous whether the ending clocking event is posedge `clk0` or posedge `clk1`.

Differently-clocked or multiply-clocked sequence operands cannot be combined with any sequence operators other than `##1`. For example, if `clk1` and `clk2` are not identical, then the following are illegal:

```
@(posedge clk1) s1 ##0 @(posedge clk2) s2

@(posedge clk1) s1 ##2 @(posedge clk2) s2

@(posedge clk1) s1 intersect @(posedge clk2) s2
```

### 17.12.2 Multiply-clocked properties

As in the case of singly-clocked properties, the result of evaluating a multiply-clocked property is either true or false. Multiply-clocked properties can be formed in a number of ways.

Multiply-clocked sequences are themselves multiply-clocked properties. For example,

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

 

is a multiply-clocked property. If a multiply-clocked sequence is evaluated as a property starting at some point, the evaluation returns true if and only if there is a match of the multiply-clocked sequence beginning at that point.

The boolean property operators (**not**, **and**, **or**) can be used freely to combine singly- and multiply-clocked properties. The meanings of the boolean property operators are the usual ones, just as in the case of singly-clocked properties. For example,

```
(@(posedge clk0) sig0) and (@(posedge clk1) sig1)
```

is a multiply-clocked property, but it is not a multiply-clocked sequence. This property evaluates to true at a point if and only if the two sequences

```
@(posedge clk0) sig0
```

and

```
@(posedge clk1) sig1
```

both have matches beginning at the point.

The non-overlapping implication operator |=> can be used freely to create a multiply-clocked property from an antecedent sequence and a consequent property that are differently- or multiply-clocked. The meaning of multiply-clocked non-overlapping implication is similar to that of singly-clocked non-overlapping implication. For example, if `s0`, `s1` are sequences with no clocking event, then in

```
@(posedge clk0) s0 |=> @(posedge clk1) s1
```

|=> synchronizes between posedge `clk0` and posedge `clk1`. Starting at the point at which the implication is being evaluated, for each match of `s0` clocked by `clk0`, time is advanced from the end point of the match to the nearest strictly future occurrence of posedge `clk1`, and from that point there must exist a match of `s1` clocked by `clk1`.

The non-overlapping implication operator |=> can synchronize between the ending clock event of its antecedent and several leading clock events for subproperties of its consequent. For example, in

```
@(posedge clk0) s0 |=> (@(posedge clk1) s1) and (@(posedge clk2) s2)
```

|=> synchronizes between posedge `clk0` and both posedge `clk1` and posedge `clk2`.

Since synchronization between distinct clocks always requires strict advance of time, the two property building operators that require special care with multiple clocks are the overlapping implication |-> and **if**/**if**...**else**.

Since |-> overlaps the end of its antecedent with the beginning of its consequent, the clock for the end of the antecedent must be the same as the clock for the beginning of the consequent. For example, if `clk0` and `clk1` are not identical and `s0`, `s1`, `s2` are sequences with no clocking events, then

```
@(posedge clk0) s0 |-> @(posedge clk1) s1 ##1 @(posedge clk2) s2
```

is illegal, but

```
@(posedge clk0) s0 |-> @(posedge clk0) s1 ##1 @(posedge clk2) s2
```

is legal.

The **if**/**if**...**else** operators overlap the test of the boolean condition with the beginning of the **if** clause property and, if present, the **else** clause property. Therefore, whenever using **if** or **if**...**else**, the **if** and **else** clause properties must begin on the same clock as the test of the boolean condition. For example, if `clk0` and `clk1` are not identical and `s0`, `s1`, `s2` are sequences with no clocking events, then

```
@(posedge clk0) if (b) @(posedge clk0) s1
```

is legal, but

```
@(posedge clk0) if (b) @(posedge clk0) s1 else @(posedge clk1) s2
```

is illegal because the **else** clause property begins on a different clock than the **if** condition.

### 17.12.3 Clock flow

Throughout this subsection, *c*, *d* denote clocking event expressions and *v*, *w*, *x*, *y*, *z* denote sequences with no clocking events.

Clock flow allows the scope of a clocking event to extend in a natural way through various parts of multiply-clocked sequences and properties and reduces the number of places at which the same clocking event must be specified.

Intuitively, clock flow provides that in a multiply-clocked sequence or property the scope of a clocking event flows left-to-right across linear operators (e.g., repetition, concatenation, negation, implication) and distributes to the operands of branching operators (e.g., conjunction, disjunction, intersection, **if**...**else**) until it is replaced by a new clocking event.

For example,

```
@(c) x |=> @(c) y ##1 @(d) z
```

can be written more simply as

```
@(c) x |=> y ##1 @(d) z
```

because clock *c* is understood to flow across |=>.

Clock flow eliminates the need to write clocking events in positions where the clock is not allowed to change. For example,

```
@(c) x |-> @(c) y ##1 @(d) z
```

can be written as

```
@(c) x |-> y ##1 @(d) z
```

to reinforce the restriction that the clock not change across |->. Similarly,

```
@(c) if (b) @(c) w ##1 @(d) x else @(c) y ##1 @(d) z
```

can be written as

```
@(c) if (b) w ##1 @(d) x else y ##1 @(d) z
```

to reinforce the restriction that the clock not change from the boolean condition *b* to the beginnings of the **if** and **else** clause properties.

Clock flow also makes the adjointness relationships between concatenation and implication clean for multiply-clocked properties:

```
@(c) x ##1 y |=> @(d) z
```

is equivalent to

 

```
@(c) x |=> y |=> @(d) z
```

and

```
@(c) x ##0 y |=> @(d) z
```

is equivalent to

```
@(c) x |-> y |=> @(d) z
```

The scope of a clocking event flows into parenthesized subexpressions and, if the subexpression is a sequence, also flows left-to-right across the parenthesized subexpression. However, the scope of a clocking event does not flow out of enclosing parentheses.

For example, in

```
@(c) w ##1 (x ##1 @(d) y) |=> z
```

*w*, *x*, *z* are clocked at *c* and *y* is clocked at *d*. Clock *c* flows across `##1`, across the parenthesized subsequence (*x* `##1` `@`(*d*) *y*), and across `|=>`. Clock *c* also flows into the parenthesized subsequence, but it does not flow through `@`(*d*). Clock *d* does not flow out of its enclosing parentheses.

As another example, in

```
@(c) v |=> (w ##1 @(d) x) and (y ##1 z)
```

*v*, *w*, *y*, *z* are clocked at *c* and *x* is clocked at *d*. Clock *c* flows across `|=>`, distributes to both operands of the **`and`** (which is a property conjunction due to the multiple clocking), and flows into each of the parenthesized subexpressions. Within (*w* `##1` `@`(*d*) *x*), *c* flows across `##1` but does not flow through `@`(*d*). Clock *d* does not flow out of its enclosing parentheses. Within (*y* `##1` *z*), *c* flows across `##1`.

Similarly, the scope of a clocking event flows into an instance of a named sequence or property, and, if the instance is a sequence, also flows left-to-right across the instance. However, a clocking event in the declaration of a sequence or property does not flow out of an instance of that sequence or property.

Note that juxtaposing two clocking events nullifies the first of them:

```
@(d) @(c) x
```

is equivalent to

```
@(c) x
```

because the flow of clock *d* is immediately overridden by clock *c*.

### 17.12.4 Examples

The following are examples of multiple-clock specifications:

```
sequence s1;
   @(posedge clk1) a ##1 b; // single clock sequence
endsequence
sequence s2;
   @(posedge clk2) c ##1 d; // single clock sequence
endsequence
```

1) multiple-clock sequence

```
sequence mult_s;
   @(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
```

```
endsequence
```

2) property with a multiple-clock sequence

```
property mult_p1;
   @(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
endproperty
```

3) property with a named multiple-clock sequence

```
property mult_p2;
   mult_s;
endproperty
```

4) property with multiple-clock implication

```
property mult_p3;
   @(posedge clk) a ##1 @(posedge clk1) s1 |=> @(posedge clk2) s2;
endproperty
```

5) property with named sequences at different clocks. In this case, if `s1` contains a clock, then it must be identical to `(posedge clk1)`. Similarly, if `s2` contains a clock, it must be identical to `(posedge clk2)`.

```
property mult_p5
   @(posedge clk1) s1 |=> @(posedge clk2) s2;
endproperty
```

6) property with implication, where antecedent and consequent are named multi-clocked sequences

```
property mult_p6;
   mult_s |=> mult_s;
endproperty
```

7) property using clock flow and overlapped implication:

```
property mult_p7;
   @(posedge clk) a ##1 b |-> c ##1 @(posedge clk1) d;
endproperty
```

Here, `a`, `b`, and `c` are clocked at posedge `clk`.

8) property using clock flow and **`if`**...**`else`**:

```
property mult_p8;
   @(posedge clk) a ##1 b |->
   if (c)
      (1 |=> @(posedge clk1) d)
   else
      e ##1 @(posedge clk2) f ;
endproperty
```

Here, `a`, `b`, `c`, and `e` are clocked at posedge `clk`.

### 17.12.5 Detecting and using endpoint of a sequence in multi-clock context

To detect the end point of a sequence when the clock of the source sequence is different than the destination sequence, method `matched` on the source sequence is used. The end point of a sequence is reached whenever there is a match on its expression. The occurrence of the end point can be tested in any sequence expression by using the method `ended` when the clocks of the source and destination sequences are the same, while method `matched` is used when the clocks are different.

The syntax of the `matched` method is:

 

```
sequence_instance.matched
```

`matched` is a method on a sequence which return true or false. Unlike `ended`, `matched` uses synchronization between the two clocks, by storing the result of the source sequence match until the arrival of the first destination clock tick after the match. When method `matched` is applied, it tests whether the source sequence has reached the end point at that particular point in time. The result of `matched` does not depend upon the starting point of the source sequence.

Like `ended`, `matched` can be used on sequences that have formal arguments.

An example is shown below:

```
sequence e1(a,b,c);
   @(posedge clk) $rose(a) ##1 b ##1 c ;
endsequence
sequence e2;
   @(posedge sysclk) reset ##1 inst ##1 e1(ready,proc1,proc2).matched [->1]
      ##1 branch_back;
endsequence
```

In this example, source sequence `e1` is evaluated at clock `clk`, while the destination sequence `e2` is evaluated at clock `sysclk`. In `e2`, the end point of the instance `e1(ready,proc1,proc2)` is tested to occur sometime after the occurrence of `inst`. Notice that method `matched` only tests for the end point of `e1(ready,proc1,proc2)` and has no bearing on the starting point of `e1(ready,proc1,proc2)`.

Local variables can be passed into an instance of a named sequence to which `matched` is applied. The same restrictions apply as in the case of `ended`. Values of local variables sampled in an instance of a named sequence to which `matched` is applied will flow out under the same conditions as for `ended`. See Section 17.8.

As with `ended`, a sequence instance to which `matched` is applied can have multiple matches in a single cycle of the destination sequence clock. The multiple matches are treated semantically the same way as matching both disjuncts of an **`or`**. In other words, the thread evaluating the destination sequence will fork to account for such distinct local variable valuations.

## 17.13 Concurrent assertions

A property on its own is never evaluated for checking an expression. It must be used within a verification statement for this to occur. A verification statement states the verification function to be performed on the property. The statement can be one of the following:

— **`assert`** to specify the property as a checker to ensure that the property holds for the design

— **`assume`** to specify the property as an assumption for the environment

— **`cover`** to monitor the property evaluation for coverage

A concurrent assertion statement can be specified in:

— an always block or initial block as a statement, wherever these blocks can appear

— a module

— an interface

— a program

```
procedural_assertion_statement ::=                          // from Annex A.6.10
        concurrent_assertion_statement
      | immediate_assert_statement

concurrent_assertion_item ::=                               // from Annex A.2.10
        [ block_identifier : ] concurrent_assertion_statement
      concurrent_assertion_statement ::=
        assert_property_statement
      | assume_property_statement
      | cover_property_statement

assert_property_statement::=
        assert property ( property_spec ) action_block

assume_property_statement::=
        assume property ( property_spec ) ;

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

*Syntax 17-16—Concurrent assert construct syntax (excerpt from Annex A)*

The **assert**, **assume** or **cover** statements can be referenced by their optional name. A hierarchical name can be used consistent with the SystemVerilog naming conventions. When a name is not provided, a tool shall assign a name to the statement for the purpose of reporting. Assertion control system tasks are described in Section 23.9.

### 17.13.1 assert statement

The **assert** statement is used to enforce a **property** as a checker. When the property for the **assert** statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the *action_block* are executed. For example,

```
property abc(a,b,c);
   disable iff (a==2) not @clk (b ##1 c);
endproperty
env_prop: assert property (abc(rst,in1,in2)) pass_stat else fail_stat;
```

When no action is needed, a null statement (i.e.;) is specified. If no statement is specified for **else**, then `$error` is used as the statement when the assertion fails.

The *action_block* shall not include any concurrent **assert**, **assume**, or **cover** statement. The *action_block*, however, can contain immediate assertion statements.

Note: The pass and fail statements are executed in the Reactive region. The regions of execution are explained in the scheduling semantics section, Section 14.

### 17.13.2 assume statement

The purpose of the **assume** statement is to allow properties to be considered as assumptions for formal analysis as well as for dynamic simulation tools. When a property is assumed, the tools constrain the environment so that the property holds.

For formal analysis, there is no obligation to verify that the assumed properties hold. An assumed property can be considered as a hypothesis to prove the asserted properties.

For simulation, the environment must be constrained such that the properties that are assumed shall hold. Like an assert property, an assumed property must be checked and reported if it fails to hold. There is no require-

 

ment on the tools to report successes of the assumed properties.

Additionally, for random simulation, biasing on the inputs provides a way to make random choices. An expression can be associated with biasing as shown below

```
expression dist { dist_list } ;  // from Annex A.1.9
```

Distribution sets and the **dist** operator are explained in Section 12.4.4.

The biasing feature is only useful when properties are considered as assumptions to drive random simulation. When a property with biasing is used in an assertion or coverage, the dist operator is equivalent to inside operator, and the weight specification is ignored. For example,

```
a1:assume property @(posedge clk) req dist {0:=40, 1:=60} ;
property proto
   @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

This is equivalent to:

```
a1_assertion:assert property req inside {0, 1} ;
property proto_assertion
   @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

In the above example, signal `req` is specified with distribution in assumption `a1`, and is converted to an equivalent assertion `a1_assertion`.

It should be noted that the properties that are assumed must hold in the same way with or without biasing. When using an assume statement for random simulation, the biasing simply provides a means to select values of free variables, according to the specified weights, when there is a choice of selection at a particular time.

Consider an example specifying a simple synchronous request - acknowledge protocol, where variable `req` can be raised at any time and must stay asserted until `ack` is asserted. In the next clock cycle both `req` and `ack` must be de-asserted.

Properties governing `req` are:

```
property pr1;
   @(posedge clk) !reset_n |-> !req; //when reset_n is asserted (0),keep req 0
endproperty
property pr2;
   @(posedge clk) ack |=> !req; // one cycle after ack, req must be de-asserted
endproperty
property pr3;
   @(posedge clk) req |-> req[*1:$] ##0 ack; // hold req asserted until
                                             // and including ack asserted
endproperty
```

Properties governing `ack` are:

```
property pa1;
   @(posedge clk) !reset_n || !req |-> !ack;
endproperty
property pa2;
   @(posedge clk) ack |=> !ack;
endproperty
```

When verifying the behavior of a protocol controller which has to respond to requests on `req`, assertions

assert_req1 and assert_req2 should be proven while assuming that statements a1, assume_ack1, assume_ack2 and assume_ack3 hold at all times.

```
a1:assume property @(posedge clk) req dist {0:=40, 1:=60} ;
assume_ack1:assume property (pr1);
assume_ack2:assume property (pr2);
assume_ack3:assume property (pr3);

assert_req1:assert property (pa1)
   else $display("\n ack asserted while req is still de-asserted");
assert_req2:assert property (pa2)
   else $display("\n ack is extended over more than one cycle");
```

Note that **assume** does not provide an action block, as the actions for an assumption serve no purpose.

### 17.13.3 cover statement

To monitor sequences and other behavioral aspects of the design for coverage, the same syntax is used with the **cover** statement. The tools can gather information about the evaluation and report the results at the end of simulation. When the property for the **cover** statement is successful, the pass statements can specify a coverage function, such as monitoring all paths for a sequence. The pass statement shall not include any concurrent **assert**, **assume** or **cover** statement.

Coverage results are divided into two: coverage for properties, coverage for sequences.

For sequence coverage, the statement appears as:

```
cover property ( sequence_expr ) statement_or_null
```

The results of coverage statement for a property shall contain:

— Number of times attempted

— Number of times succeeded

— Number of times failed

— Number of times succeeded because of vacuity

In addition, *statement_or_null* is executed every time a property succeeds.

Vacuity rules are applied only when implication operator is used. A property succeeds non-vacuously only if the consequent of the implication contributes to the success.

Results of coverage for a sequence shall include:

— Number of times attempted

— Number of times matched (each attempt can generate multiple matches)

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.

### 17.13.4 Using concurrent assertion statements outside of procedural code

A concurrent assertion statement can be used outside of a procedural context. It can be used within a module, an interface, or a program. A concurrent assertion statement is an **assert**, an **assume**, or a **cover** statement. Such a concurrent assertion statement uses the **always** semantics.

The following two forms are equivalent:

 

```
assert property ( property_spec ) action_block

always assert property ( property_spec ) action_block ;
```

Similarly, the following two forms are equivalent:

```
cover property ( property_spec ) statement_or_null

always cover property ( property_spec ) statement_or_null
```

For example:

```
module top(input bit clk);
   logic a,b,c;
   property rule3;
      @(posedge clk) a |-> b ##1 c;
   endproperty
   a1: assert property (rule3);
   ...
endmodule
```

`rule3` is a property declared in module `top`. The assert statement `a1` starts checking the property from the beginning to the end of simulation. The property is always checked. Similarly,

```
module top(input bit clk);
   logic a,b,c;
   sequence seq3;
      @(posedge clk) b ##1 c;
   endsequence
   c1: cover property (seq3);
   ...
endmodule
```

The cover statement `c1` starts coverage of the sequence `seq3` from beginning to the end of simulation. The sequence is always monitored for coverage.

### 17.13.5 Embedding concurrent assertions in procedural code

A concurrent assertion statement can also be embedded in a procedural block. For example:

```
property rule;
   a ##1 b ##1 c;
endproperty

always @(posedge clk) begin
   <statements>
   assert property (rule);
end
```

If the statement appears in an **`always`** block, the property is always monitored. If the statement appears in an **`initial`** block, then the monitoring is performed only on the first clock tick.

Two inferences are made from the procedural context: clock from the event control of an **`always`** block, and the enabling conditions.

A clock is inferred if the statement is placed in an **`always`** or **`initial`** block with an event control abiding by the following rules:

— The clock to be inferred must be placed as the first term of the event control as an edge specifier (**posedge** *expression* or **negedge** *expression*).

— The variables in *expression* must not be used anywhere in the **always** or **initial** block.

For example:

```
property r1;
   q != d;
endproperty
always @(posedge mclk) begin
   q <= d1;
   r1_p: assert property (r1);
end
```

The above property can be checked by writing statement `r1_p` outside the always block, and declaring the property with the clock as:

```
property r1;
   @(posedge mclk)q != d;
endproperty
always @(posedge mclk) begin
   q <= d1;
end
r1p: assert property (r1);
```

If the clock is explicitly specified with a property, then it must be identical to the inferred clock, as shown below:

```
property r2;
   @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
   q <= d1;
   r2_p: assert property (r2);
end
```

In the above example, `(posedge mclk)` is the clock for property `r2`.

Another inference made from the context is the enabling condition for a property. Such derivation takes place when a property is placed in an **if**...**else** block or a **case** block. The enabling condition assumed from the context is used as the antecedent of the property.

```
property r3;
   @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
   if (a) begin
      q <= d1;
      r3_p: assert property (r3);
   end
end
```

The above example is equivalent to:

```
property r3;
   @(posedge mclk)a |-> (q != d);
endproperty
r3_p: assert property (r3);
```

 

```
always @(posedge mclk) begin
   if (a) begin
      q <= d1;
   end
end
```

Similarly, the enabling condition is also inferred from **case** statements.

```
property r4;
   @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
   case (a)
      1: begin q <= d1;
               r4p: assert property (r4);
         end
      default: q1 <= d1;
   endcase
end
```

The above example is equivalent to:

```
property r4;
   @(posedge mclk)(a==1) |-> (q != d);
endproperty
r4_p: assert property (r4);
always @(posedge mclk) begin
   case (a)
      1: begin q <= d1;
         end
      default: q1 <= d1;
   endcase
end
```

The enabling condition is inferred from procedural code inside an **always** or **initial** block, with the following restrictions:

1) There must not be a preceding statement with a timing control.

2) A preceding statement shall not invoke a task call which contains a timing control on any statement.

3) The concurrent assertion statement shall not be placed in a looping statement, immediately, or in any nested scope of the looping statement.

## 17.14 Clock resolution

There are a number of ways to specify a clock for a property:

— sequence instance with a clock, for example

```
sequence s2; @(posedge clk) a ##2 b; endsequence
property p2; not s2; endproperty
assert property (p2);
```

— property, for example:

```
property p3; @(posedge clk) not (a ##2 b); endproperty
assert property (p3);
```

— contextually inferred clock from a procedural block, for example:

```
always @(posedge clk) assert property (not (a ##2 b));
```

— clocking block, for example:

```
clocking master_clk @(posedge clk);
   property p3; not (a ##2 b); endproperty
endclocking
assert property (master_clk.p3);
```

— default clock, for example:

```
default clocking master_clk ;  // master clock as defined above
property p4; (a ##2 b); endproperty
assert property (p4);
```

For a multi-clocked assertion, the clocks are explicitly specified. No default clock or inferred clock is used. In addition, multi-clocked properties are not allowed to be defined within a clocking block.

A multi-clocked property assert statement must not be embedded in procedural code where a clock is inferred. For example, following forms are not allowed.

```
always @(clk) assert property (mult_clock_prop);// illegal
initial @(clk) assert property (mult_clock_prop);// illegal
```

The rules for an assertion with one clock are discussed in the following paragraphs.

The clock for an assertion statement is determined in the decreasing order of priority:

1) Explicitly specified clock for the assertion.

2) Inferred clock from the context of the code when embedded.

3) Default clock, if specified.

A concurrent assertion statement must resolve to a clock. Otherwise, the statement is considered illegal.

Sequences and properties specified in clocking blocks resolve the clock by the following rules:

1) Event control of the clocking block specifies the clock.

2) No explicit event control is allowed in any property or sequence declaration.

3) If a named sequence that is defined outside the clocking block is used , its clock, if specified, must be identical to the clocking block's clock.

4) Multi-clock properties are not allowed.

Resolution of clock for a sequence declaration assumes that only one explicit event control can be specified. Also, the named sequences used in the sequence declaration can, but do not need to, contain event control in their definitions.

```
sequence s;
   //sequence composed of two named subsequences
   @(posedge s_clk) e ##1 s1 ##1 s2 ##1 f;
endsequence
sequence s1;
   @(posedge clk1) a ##1 b; // single clock sequence
endsequence
sequence s2;
```

 

```
    @(posedge clk2) c ##1 d; // single clock sequence
endsequence
```

These example sequences are used in Table 17-3 to explain the clock resolution rules for a sequence declaration. The clock of any sequence when explicitly specified is indicated by X. Otherwise, it is indicated by a dash.

**Table 17-3: Resolution of clock for a sequence declaration**

| s_clk | clk1 | clk2 | Resolved clock | Semantic restriction |
|---|---|---|---|---|
| - | - | - | unclocked | - |
| X | - | - | s_clk | - |
| X | X | - | s_clk | s_clk and clk1 must be identical |
| X | X | X | s_clk | s_clk, clk1 and clk2 must be identical |
| X | - | X | s_clk | s_clk and clk2 must be identical |
| - | X | - | unclocked | - |
| - | X | X | unclocked | clk1 and clk2 must be identical |
| - | - | X | unclocked | - |

Once the clock for a sequence declaration is determined, the clock of a property declaration is resolved similar to the resolution for a sequence declaration. A single clocked property assumes that only one explicit event control can be specified. Also, the named sequences used in the property declaration can contain event control in their declarations. Table 17-4 specifies the rules for property declaration clock resolution. The property has the form:

```
property p;
   @(posedge p_clk) not s1 |=> s2;
endproperty
```

p_clk is the property for the clock, clk1 is the clock for sequence s1 and clk2 is the clock for sequence s2. The same rules apply for operator |->.

**Table 17-4: Resolution of clock for a declaration**

| p_clk | clk1 | clk2 | Resolved clock | Semantic restriction |
|---|---|---|---|---|
| - | - | - | unclocked | - |
| X | - | - | p_clk | - |
| X | X | - | p_clk | p_clk and clk1 must be identical |
| X | X | X | p_clk | p_clk, clk1 and clk2 must be identical |
| X | - | X | p_clk | p_clk and clk2 must be identical |
| - | X | - | unclocked | - |
| - | X | X | unclocked or multi-clock | clk1 and clk2 must be identical. If clk1 and clk2 are different for the case of operator \|=>, then it is considered a multi-clock implication |

**Table 17-4: Resolution of clock for a declaration**

| p_clk | clk1 | clk2 | Resolved clock | Semantic restriction |
|---|---|---|---|---|
| - | - | X | unclocked | - |

Resolution of clock for an **`assert`** statement is based on the following assumptions:

— **`assert`** can appear in an **`always`** block, **`initial`** block or outside procedural context

— clock is inferred from an **`always`** or **`initial`** block

— default clock can be specified using default clocking block

Table 17-5 specifies the rules for clock resolution when **`assert`** appears in an always or initial block, where `i_clk` is the inferred clock from an **`always`** or **`initial`** block, `d_clk` is the default clock, and `p_clk` is the property clock.

**Table 17-5: Resolution of clock in an always or initial block**

| i_clk | d_clk | p_clk | Resolved clock | Semantic restriction |
|---|---|---|---|---|
| - | - | - | unclocked | Error. An assertion must have a clock |
| X | - | - | i_clk | - |
| - | X | - | d_clk | |
| - | - | X | p_clk | |
| X | - | X | i_clk | `i_clk` and `p_clk` must be identical |
| X | X | - | i_clk | - |
| - | X | X | p_clk | |
| - | - | X | p_clk | - |

When the **`assert`** statement is outside any procedural block, there is no inferred clock. The rules for clock resolution are specified in Table 17-6.

**Table 17-6: Resolution of clock outside a procedural block**

| d_clk | p_clk | Resolved clock | Semantic restriction |
|---|---|---|---|
| – | – | unclocked | Error. An assertion must have a clock |
| X | - | d_clk | |
| – | X | p_clk | |
| X | X | p_clk | |

### 17.14.1 Clock resolution in multiply-clocked properties

Throughout this subsection, $s$, $s_1$, $s_2$ denote sequences without clocking events; $p$, $p_1$, $p_2$ denote properties without clocking events; $m$, $m_1$, $m_2$ denote multiply-clocked sequences, $q$, $q_1$, $q_2$ denote multiply-clocked

 

properties; and $c, c_1, c_2$ denote non-identical clocking event expressions.

Due to clock flow, juxtaposition of two clocks nullifies the first. This and the nesting of clocking events within other property building operators mean that there are subtleties in the general interpretation of the restrictions about where the clock can change in multiply-clocked properties. For example,

```
@(c) s |-> @(c) (p and @(c1) p1)
```

appears legal because the antecedent is clocked by $c$ and the consequent begins syntactically with the clocking event @($c$). However, the consequent sequence is equivalent to

```
(@(c) p) and (@(c1) p1)
```

and |-> cannot synchronize between clock $c$ from the antecedent and clock $c_1$ from the second conjunct of the consequent. Similarly,

```
@(c) s |-> @(c1) (@(c) p)
```

appears illegal due to the apparent clock change from $c$ to $c_1$ across |->. However, it is legal, although arguably misleading in style, because the consequent property is equivalent to @($c$) $p$.

This subsection gives a more precise treatment of the restrictions on multiply-clocked use of |-> and **if**/**if**...**else** than the intuitive discussion in Section 17.12. The present treatment depends on the notion of the set of semantic leading clocks for a multiply-clocked sequence or property.

Some sequences and properties have no explicit leading clock event. Their initial clocking event is inherited from an outer clocking event according to the flow of clocking event scope. In this case, the semantic leading clock is said to be *inherited*. For example, in the property

```
@(c) s |=> p and @(c1) p1
```

the semantic leading clock of the subproperty $p$ is *inherited* since the initial clock of $p$ is the clock that flows across |=>.

A multiply-clocked sequence has a unique semantic leading clock, defined inductively as follows.

— The semantic leading clock of s is *inherited*.

— The semantic leading clock of @($c$) $s$ is $c$.

— If *inherited* is the semantic leading clock of $m$, then the semantic leading clock of @($c$) $m$ is $c$. Otherwise, the semantic leading clock of @($c$) $m$ is equal to the semantic leading clock of $m$.

— The semantic leading clock of ($m$) is equal to the semantic leading clock of $m$.

— The semantic leading clock of $m_1$ ## $m_2$ is equal to the semantic leading clock of $m_1$.

The set of semantic leading clocks of a multiply-clocked property is defined inductively as follows.

— The set of semantic leading clocks of $m$ is {$c$}, where $c$ is the unique semantic leading clock of $m$.

— The set of semantic leading clocks of $p$ is {*inherited*}.

— If *inherited* is an element of the set of semantic leading clocks of $q$, then the set of semantic leading clocks of @($c$) $q$ is obtained from the set of semantic leading clocks of $q$ by replacing *inherited* by $c$. Otherwise, the set of semantic leading clocks of @($c$) $q$ is equal to the set of semantic leading clocks of $q$.

— The set of semantic leading clocks of ($q$) is equal to the set of semantic leading clocks of $q$.

— The set of semantic leading clocks of **not** $q$ is equal to the set of semantic leading clocks of $q$.

— The set of semantic leading clocks of $q_1$ and $q_2$ is the union of the set of semantic leading clocks of $q_1$ with the set of semantic leading clocks of $q_2$.

— The set of semantic leading clocks of $q_1$ or $q_2$ is the union of the set of semantic leading clocks of $q_1$ with the set of semantic leading clocks of $q_2$.

— The set of semantic leading clocks of $m$ |-> $p$ is equal to the set of semantic leading clocks of $m$.

— The set of semantic leading clocks of $m$ |=> $p$ is equal to the set of semantic leading clocks of $m$.

— The set of semantic leading clocks of **if** ($b$) $q$ is {*inherited*}.

— The set of semantic leading clocks of **if** ($b$) $q_1$ else $q_2$ is {*inherited*}.

— The set of semantic leading clocks of a property instance is equal to the set of semantic leading clocks of the multiply-clocked property obtained from the body of its declaration by substituting in actual arguments.

For example, the multiply-clocked sequence

```
@(c1) s1 ## @(c2) s2
```

has $c_1$ as its unique semantic leading clock, while the multiply-clocked property

```
not (p1 and (@(c2) p2)
```

has {*inherited*, $c_2$} as its set of semantic leading clocks.

In the presence of an incoming outer clock, the inherited semantic leading clock is always understood to refer to the incoming outer clock. On the other hand, if a property has only explicit semantic leading clocks, then the incoming outer clock has no effect on the clocking of the property since the explicit clock events replace the incoming outer clock. Therefore, the clocking of a property $q$ in the presence of incoming outer clock $c$ is equivalent to the clocking of the property @($c$) $q$.

The rules for using multiply-clocked overlapping implication and **if**/**if**...**else** in the presence of an incoming outer clock can now be stated more precisely.

1) Multiply-clocked overlapping implication.

   Let $c$ be the incoming outer clock. Then the clocking of $m$ |-> $q$ is equivalent to the clocking of @($c$) $m$ |-> $q$

   In the presence of the incoming outer clock, $m$ has a well-defined ending clock, and there is a well-defined clock that flows across |->. The multiply-clocked overlapped implication $m$ |-> $q$ is legal for incoming clock $c$ if and only if the following two conditions are met:

   a) Every explicit semantic leading clock of $q$ is identical to the ending clock of $m$.

   b) If *inherited* is a semantic leading clock of $q$, then the ending clock of $m$ is equal to the clock that flows across |->.

   For example

   ```
   @(c) s |-> p1 or @(c2) p2
   ```

   is not legal because the ending clock of the antecedent is $c$, while the consequent has $c_2$ as an explicit semantic leading clock.

   Also,

   ```
   @(c) s ## (@(c1) s1) |-> p
   ```

   is not legal because the set of semantic leading clocks of $p$ is {*inherited*}, the ending clock of the antecedent is $c_1$, and the clock that flows across |-> and is inherited by $p$ is $c$.

   On the other hand,

 

```
@(c) s |-> p1 or @(c) p2
```

and

```
@(c) s ## @(c1) s1 |-> p1 or @(c1) p2
```

are both legal.

2) Multiply-clocked **if**/**if**...**else**

Let $c$ be the incoming outer clock. Then the clocking of **if** ($b$) $q_1$ [ **else** $q_2$ ] is equivalent to the clocking of

```
@(c) if (b) q1 [ else q2 ]
```

The boolean condition $b$ is clocked by $c$, so the multiply-clocked **if**/**if**...**else if** ($b$) $q_1$ [ **else** $q_2$ ] is legal for incoming clock $c$ if and only if the following condition is met:

— Every explicit semantic leading clock of $q_1$ [ **or** $q_2$ ] is identical to $c$.

For example,

```
@(c) if (b) p1 else @(c) p2
```

is legal, but

```
@(c) if (b) @(c) (p1 and @(c2) p2)
```

is not.

## 17.15 Binding properties to scopes or instances

To facilitate verification separate from the design, it is possible to specify properties and bind them to specific modules or instances. The following are the goals of providing this feature:

— It allows verification engineers to verify with minimum changes to the design code/files.

— It allows a convenient mechanism to attach verification IP to a module or an instance.

— No semantic changes to the assertions are introduced due to this feature. It is equivalent to writing properties external to a module, using hierarchical path names.

With this feature, a user can bind a module, interface, or program instance to a module or a module instance.

The syntax of the **bind** construct is:

```
bind_directive ::= bind hierarchical_identifier constant_select bind_instantiation ;      // from Annex A.1.5
bind_instantiation ::=
        program_instantiation
      | module_instantiation
      | interface_instantiation
```

*Syntax 17-17—bind construct syntax (excerpt from Annex A)*

The **bind** directive can be specified in

— a module

— an interface

— a compilation-unit scope

A program block contains non-design code (either testbench or properties) and executes in the Reactive region, as explained in Section 16.

Example of binding a program instance to a module:

```
bind cpu fpu_props fpu_rules_1(a,b,c);
```

Where:

— `cpu` is the name of module.

— `fpu_props` is the name of the program containing properties.

— `fpu_rules_1` is the program instance name.

— Ports `(a, b,c)` get bound to signals `(a,b,c)` of module `cpu`.

— Every instance of `cpu` gets the properties.

Example of binding a program instance to a specific instance of a module:

```
bind cpu1 fpu_props fpu_rules_1(a,b,c);
```

By binding a program to a module or an instance, the program becomes part of the bound object. The names of assertion-related declarations can be referenced using the SystemVerilog hierarchical naming conventions.

Binding of a module instance or an interface instance works the same way as described for programs above.

```
interface range (input clk,enable, input int minval,expr);
   property crange_en;
      @(posedge clk) enable |-> (minval <= expr);
   endproperty
range_chk: assert property (crange_en);
endinteface

bind cr_unit range r1(c_clk,c_en,v_low,(in1&&in2));
```

In this example, interface `range` is instantiated in the module `cr_unit`. Effectively, every instance of module `cr_unit` shall contain the interface instance `r1`.

## 17.16 The expect statement

The **`expect`** statement is a procedural blocking statement that allows waiting on a property evaluation. The syntax of the **`expect`** statement accepts a named property or a property declaration, and is given below.

```
expect_property_statement ::=                                  // from Annex A.2.10
        expect ( property_spec ) action_block
```

*Syntax 17-18—expect statement syntax (excerpt from Annex A)*

The **`expect`** statement accepts the same syntax used to assert a property. 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 Observe 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.

```
program tst;
   initial begin
      # 200ms;
      expect( @(posedge clk) a ##1 b ##1 c ) else $error( "expect failed" );
      ABC: ...
   end
endprogram
```

In the above example, the **expect** statement specifies a property that consists of the sequence a ##1 b ##1 c. The expect statement (second statement in the initial block of program `tst`) blocks until the sequence `a ##1 b ##1 c` is matched, or is determined not to match. The property evaluation starts on the clocking event (posedge clk) following the 200ms delay. If the sequence is matched, the process is unblocked and continues to execute on the statement labeled ABC. If the sequence fails to match then the **else** clause is executed, which in this case generates a run-time error. For the expect above to succeed, the sequence `a ##1 b ##1 c` must match starting on the clocking event (posedge clk) immediately after time 200ms. The sequence will not match if `a`, `b`, or `c` are evaluated to be false at the first, second or third clocking event respectively.

The **expect** statement can be incorporated in any procedural code, including tasks or class methods. Because it is a blocking statement, the property can refer to automatic variables 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 by the automatic argument value. The second argument, `success`, is used to return the result of the **expect** statement: 1 for success and 0 for failure.

```
integer data;
...
task automatic wait_for( integer value, output bit success );
expect( @(posedge clk) ##[1:10] data == value ) success = 1;
   else success = 0;
endtask

initial begin
   bit ok;
   wait_for( 23, ok );  // wait for the value 23
   ...
end
```