Correcting usage of terms, grammatical errors and omissions

These changes are related to the use of terms “sequence” vs. “sequence expression”, “named” vs. “defined” and “declaration” vs. “definition”, grammatical errors and omissions.

1.1 Page 198, Section 17.1

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 a 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 this clock ~~semantic~~ 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.

1.2 Page 198, Section 17.2

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 ~~treated as a condition as in an~~ **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 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.

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 evaluation of the expression follows the same semantic as that of the conditional context of the **if** statement. As with the **if** statement, if the conditional expression evaluates to `X`, `Z` or `0`, then the assertion fails.~~ The pass statement can, for example, record the number of successes for a coverage log, but it 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 ~~assertion fails~~. ~~That is, the expression does not evaluate to a known, non-zero value.~~ The **else** statement can also be omitted. The action block is executed immediately after the evaluation of the assert expression.

1.3 Page 200, Section 17.3

A *clock tick* is an atomic moment in time that itself spans no duration of time. ~~and implies that there is no duration of time in a clock tick.~~ 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. ~~It is also given that a clock shall tick only once at any simulation time, and the sampled values for that simulation time are used for evaluation.~~ 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, at clock tick 9, the simulation value transitions to~~

~~high. However, the sampled value is low.~~ Notice that the simulation value transitions to high at clock tick 9. However, the sampled value at clock tick 9 is low.

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 sub-expressions that are required to determine a match of a sequence~~with respect to a sequence expression~~.

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

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. An expression such as `(clk && gating_signal)` **and** `(clk` **iff** `gating_signal)` could be used to represent a gated clock~~s~~. Other more complex expressions are possible. 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.

1.4 Page 201, Section 17.4

The expressions used in sequences are evaluated over sampled values of the variables that appear in the expression. The outcome of the evaluation of an expression~~s~~ 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.

1.5 Page 201, Section 17.4.1

The following types are not allowed:

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

1.6 Page 201, Section 17.4.1

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

1.7 Page 202, Section 17.4.3

All operators that are valid for the types described in Section 17.4.1 are allowed with the exception of assignment operators ~~or~~ 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.

1.8 Page 204, Section 17.5

~~Properties are often constructed out of sequential behavior. The **sequence** feature provides the capability to build and manipulate sequential behavior. A sequence is a list of SystemVerilog boolean expressions in a linear order of increasing time. The boolean expressions must be true at those specific clock ticks for the sequence to be true over time. A boolean expression at a point in time is a simple case of a sequence with time length of~~

~~one clock cycle. To determine a match of a sequence, the boolean expressions are evaluated at each successive clock tick in an attempt to satisfy the sequence. If all expressions are true, then a match of the sequence occurs.~~

~~A sequence expression describes one or more sequences by using *regular expressions*. Such a *regular expression* can concisely specify a set of zero, finitely many, or infinitely many sequences that satisfy the sequence expression.~~

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 ~~and sequence expressions~~ 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.

1.9 Page 204, Section 17.5

~~A ## followed by an optional number or range specifies that the *sequence_expr* should occur later than the current cycle. A number of 1 indicates that the next element should occur a single cycle later than the current cycle. The number 0 specifies that the next expression should occur in parallel with the current clock tick.~~

A ## followed by a number or range specifies the delay from the current cycle 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 cycle. The delay `##0` indicates that the beginning of the sequence that follows is at the same clock tick as the current cycle.

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.

1.10 Page 204, Section 17.5

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:

1.11 Page 205, Section 17.5

In the above example, `c` ~~is~~ must be true at the endpoint of sequence `seq1`, and `d` ~~is~~ must be true at the start of sequence `seq2`. When concatenated with 0 clock tick delay, `c` and `d` must be true ~~occur~~ at the same time,

resulting in a concatenated sequence equivalent to:

1.12 Page 205, Section 17.5

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

1.13 Page 205, Section 17.5

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 4th and 32nd after the current clock tick.

1.14 Page 208, Section 17.6

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

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

1.15 Page 208, Section 17.6

~~To use **sequence** as a sub-expression or a part of the expression, simply reference its name. The evaluation of a sequence expression that references a sequence is performed the same way as if the sequence expression contained in the **sequence** was a lexical part of the expression, with the formal arguments substituted by the actual ones and the remaining variables that were not arguments substituted from the scope of declaration. An example is shown below:~~

To use a named sequence as a sub-sequence 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 were 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:

1.16 Page 208, Section 17.7.1

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

1.17 Page 209, Section 17.7.1, Table 17-1

The operators in Table 17-1 must be in bold font.

1.18 Page 209, Section 17.7.2

~~The repetition counts are specified as a range and the minimum and maximum range expressions must be literals or constant expressions.~~

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* (`[*..]`) ~~, where a sequence is consecutively repeated with one cycle delay between the repetitions~~ :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* (`[->..]`) ~~, where a boolean expression is repeated with one or more cycle delays between the repetitions and the resulting sequence terminates at the last boolean expression~~ :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* (`[=..]`) ~~, where a boolean expression is repeated with one or more cycle delays between the repetitions and the resulting sequence can proceed beyond the last boolean expression, but before the occurrence of the boolean expression~~ :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.

~~To specify the *consecutive repetition* of an expression within a sequence, the expression can simply be repeated, as:~~The effect of consecutive repetition of a sub-sequence within a sequence can be achieved by explicitly iterating the sub-sequence, as:

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

~~Or the number of repetitions can be specified with `[*N]`, as:~~ 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 item or expression must occur a specified number of times.~~ A consecutive repetition specifies that the operand sequence must match a specified number of times. ~~Each repeated item is concatenated (with a delay of 1 clock tick) to the next repeated item.~~ A repeat of N specifies that the sequence must occur N times in succession. The consecutive repetition operator `[*N]` specifies that the operand sequence must match `[*N]` times in succession. For example:

1.19 Page 210, Section 17.7.2

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

```
a [*0]
```

~~An empty sequence shall be illegal.~~

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. The 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)
```

1.20 Page 210, Section 17.7.2

~~Which~~ This is the same as:

1.21 Page 210, Section 17.7.2

~~A repetition with a range of maximum and minimum number of times can be expressed with [* min:max]. As an example, the following two expressions are equivalent.~~A repetition with a range of `min` minimum and `max` maximum number of iterations can be expressed with the consecutive repetition operator {\courier `[* min:max]`.

As an example,

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

is equivalent to

1.22 Page 210, Section 17.7.2

~~The following two expressions are also equivalent.~~ 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 ##~~1c~~1 c)
```

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

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

~~means a is true on the current sample, then b shall be true on every subsequent sample until c is true. On the sample in which c is true, b does not have to be true.~~ 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.

1.23 Page 211, in Section 17.7.2

~~The rules for specifying repeat counts are summarized as:~~

- ~~Each form of repeat count specifies a minimum and maximum number of occurrences~~
- ~~expression [*n:m], where n is the minimum, m is the maximum [*m:n], where m is the minimum, n is the maximum, and m >= 0, m<= n or n is $~~
- ~~expression [*n] is the same as expression [*n:n]~~

- ~~The sequence as a whole cannot be empty~~
- ~~If n is 0, then there must be either a prefix, or a suffix concatenation term (i.e., not the only term in the expression) to the repeated sequence~~
- ~~The match shall not be empty~~

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 [*N] notation indicates *consecutive repetition* of an expression.~~

~~The *goto repetition* (non-consecutive exact repetition) specifies the repetition of a boolean expression, such as:~~

```
a ##1 b [* >min:max] ##1 c
```

~~This is equivalent to:~~

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max]) ##1 c
```

~~Adding the range specification to this allows the construction of useful sequences containing a boolean expression that is true for at most N occurrences:~~

```
a ##1 b[*->1:N] ##1 c //a followed by at most N occurrences of b, followed by c
```

~~The *non-consecutive repetition* extends the *goto repetition* by extra clock ticks where the boolean expression is not true.~~

```
a ##1 b [*=min:max] ##1 c
```

~~This is equivalent to:~~

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max]) ##1 !b[*0:$] ##1 c
```

~~The above expression would pass the following sequence, assuming that 3 is within the min:max range.~~

```
a c c c c b c c b c b d d d c
```

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 repetition 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
```

1.24 Page 212, in Section 17.7.3

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

~~$fell (expression [, clocking_event])~~
~~$stable (expression [, clocking_event])~~
~~$past (expression1 [, number_of_ticks] [, expression2] [, clocking_event])~~

$sampled(expression [**,** clocking_event] **)**
$rose (expression [**,** clocking_event] **)**
$fell (expression [**,** clocking_event] **)**
$stable (expression [**,** clocking_event] **)**
$past (expression1 [**,** number_of_ticks] [**,** expression2] [**,** clocking_event] **)**

1.25 Page 212, in Section 17.7.3

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

1.26 Page 212, in Section 17.7.3

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

1.27 Page 214, in Section 17.7.3

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

1.28 Page 214, Section 17.7.4

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

```
te1 and te2
```

— ~~Succeeds~~ Match if `te1` and `te2` ~~succeed~~ match.

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

The following example is an expression with the **and** operator, where the two operands are ~~single sequence evaluations~~ simple linear sequences.

1.29 Page 215, Section 17.7.4

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 ~~entire expression~~ composite sequence is the ~~last~~ later of the two end times, so a match is recognized for the ~~expression~~ composite sequence at clock tick 12.

In the following example, ~~an~~ the first operand sequence has a concatenation operator with range from 1 to 5 ~~is associated with a range of time specification, such as~~:

```
(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:

~~The first operand sequence consists of an expression with a time range from 1 to 5 and implies that when `te1`~~

~~evaluates to true, `te2` must follow 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, the following steps are taken:~~

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

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

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 ~~Figure 17-5 shows the attempt to examine at clock tick 8 when both operand sequences start and succeed. All five sequences for the first operand sequence match, as shown in a time window, 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. ~~To compute the result for the composite expression, each successful sequence from the first operand sequence is matched against the second operand sequence according to the rules of the and operation to determine the end time for each match.~~

~~The result of this computation is five successes, four of them ending at clock tick 12, and the fifth ends at clock tick 13. Figure 17-5 shows the two unique successes at clock ticks 12 and 13.~~

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.

1.30 Page 216, Section 17.7.4

If `te1` and `te2` are sampled booleans (not sequences), the expression (`te1` **and** `te2`) matches ~~succeeds~~ if `te1` and `te2` are both evaluate to true ~~evaluated to be true~~.

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

1.31 Page 217, Section 17.7.5

The two operands of **intersect** are sequences ~~expressions~~. The requirements for the match ~~success~~ of the **intersect** operation are

1.32 Page 217, Section 17.7.5

~~For each attempted evaluation of *sequence_expr*, there could be multiple matches. When there are multiple matches for each operand sequence expression, the results are computed as follows.~~

- ~~A match from the first operand is paired with a match from the second operand with the same length.~~
- ~~If no such pair is found, the result of **intersect** is no match.~~
- ~~If such pairs are found, then the result consists of matched sequences, one for each pair. The end time of each match is determined by the length of the pair.~~

~~Figure 17-7 is similar to Figure 17-5, except that **and** is replaced by **intersect**. Compared with Figure 17-5,~~

~~there is only a single match in this case.~~

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** has been replaced by **intersect**. In this case, unlike in Figure 17-5, there is only a single match at clock tick 12.

1.33 Page 218, Section 17.7.6

The two operands of **or** are ~~sequence expressions~~ sequences.

1.34 Page 218, Section 17.7.6

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

~~For the expression:~~
~~`te1` **or** `te2`~~

~~when operands `te1` and `te2` are expressions, the sequence matches whenever at least one of two operands `te1` and `te2` is evaluated to true.~~

~~Figure 17-8 illustrates an **or** operation using `te1` and `te2` as simple values. The expression does not match at clock ticks 7 and 13 because `te1` and `te2` are both false at those times. At all other times, the expression matches, as at least one of the two operands is true.~~

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 18 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.

When `te1` and `te2` are sequences, then the ~~expression~~ sequence
`te1` **or** `te2`

~~matches if at least one of the two operand sequences te1 and te2 successfully matched sequences of each operand are calculated and two groups is computed. The result of the union provides the result is the end time of any sequence that matched. The following example shows an expression a sequence with operator **or** operator, where the two operands are sequences.~~

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.

1.35 Page 219, Section 17.7.7

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 ~~expression~~ are recognized.

~~In the next example, an operand sequence is associated with a time range specification, such as:~~

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 consists of an expression with a time range from 1 to 5 and specifies that when `te1` evaluates to true, `te2` must be true 1, 2, 3, 4, or 5 clock ticks later. The sequences from the second operand require that first `te3` must be true followed by `te4` being true two clock ticks later, followed by `te5` being true two clock ticks later. At any clock tick if an operand sequence succeeds, then the composite expressions succeeds.~~

~~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 match of the composite expression is computed as a union of the matches of the two operand sequences, which results in matches at clock ticks 9, 10, 11, 12, and 13.~~

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 sequence 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.

1.36 Page 220, Section 17.7.7

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

~~The operand expression can be a sequence expression. *sequence_expr* is evaluated to determine the match for the (**first_match** (`sequence_expr`)) expression. For a given evaluation attempt, the composite expression matches if *sequence_expr* results in at least one match of a sequence and fails to match if none of the sequences from the expression result in a match. Following the first successful match for the attempt, the **first_match** operator stops matching subsequent sequences for sequence_expr. For an attempt, if there are multiple matches with the same end time as the first detected match, then all those matches are considered as the result of the expression.~~

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*.

1.37 Page 220, Section 17.7.7

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

1.38 Page 221, Section 17.7.7

Each attempt of sequence `t1` can result in matches for up to four of the following sequences:

1.39 Page 221, Section 17.7.7

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

1.40 Page 221, Section 17.7.7

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` results in the earliest match. In this case, it is possible to have two matches ending at the same time.~~ 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
```

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

1.41 Page 221, Section 17.7.8

~~*expression* must evaluate true at every clock tick during the evaluation of *sequence_expr*. If an evaluation of *sequence_expr* starts at time `t1` and ends with a match at time `t2`, then for *sequence_expr* to match, expression must hold true from time `t1` to `t2`.~~

~~The **throughout** construct is an abbreviation for writing:~~

```
(expression) [*0:$] intersect sequence_expr
```

~~In the following example, illustrated in Figure 17-11, if a constraint were placed on the expression as shown below, then the checker `burst_rule1` would fail at clock tick 9.~~

The construct `expr` **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.

1.42 Page 222, Section 17.7.8

~~In the above expression, the value of signal `burst_mode` is required to be low during the sequence (from clock tick 2 to 10) and is checked at every clock tick during that period. At clock ticks from 2 to 8, signal `burst_mode` remains low and matches the expression at those clock ticks. At clock tick 9, signal `burst_mode` becomes high, thereby failing to match the expression for `burst_rule1`.~~

~~If signal `burst_mode` were to be maintained low until at least clock tick 10, the expression would result in a match as shown in Figure 17-12.~~

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 sub-sequence `(##2 ((trdy==0)&&(irdy==0)) [*7])` beginning at clock tick 2. This sub-sequence 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.

1.43 Page 222, Section 17.7.9

17.7.9 Sequence ~~occurrence~~ contained within another sequence

1.44 Page 222, Section 17.7.9

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

1.45 Page 223, Section 17.7.9

~~The within construct:~~

~~`sequence_expr1` **within** `sequence_expr2`~~

~~is an abbreviation for writing:~~

~~`(1[*0:$] ##1 sequence_expr1 ##1 1[*0:$])` **intersect** `sequence_expr2`~~

~~The sequence `sequence_expr1` must occur at least once entirely within the sequence `sequence_expr2`. That is, `sequence_expr1` must satisfy the following:~~

- ~~The start point of `sequence_expr1` must be between the start point and the end point (start and end point being inclusive) of `sequence_expr2`.~~
- ~~The end point of `sequence_expr1` must be between the start point and the end point (start and end point being inclusive) of `sequence_expr2`.~~

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

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

1.46 Page 223, Section 17.7.10

There are two ways in which a complex sequence can be decomposed into simpler sub-sequences ~~sub-expressions~~.

1.47 Page 223, Section 17.7.10

~~One is to reference the name of a sequence, thereby causing it to be started at the point where it is referenced, as shown below:~~

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 ~~cycle~~ tick after the ~~occurrence~~ evaluation of `start_trans` in the sequence `rule`.

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

The syntax of the ended method is:
```
sequence_identifier.ended
```

`ended` is a method on a sequence. The result of its operation is true or false. When method `ended` is ~~applied~~ evaluated in an expression, it tests whether its operand sequence ~~*seq_name*~~ has reached ~~the~~ 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 ~~*seq_name*~~. An example is shown below:

1.48 Page 224, Section 17.7.10

In this example, sequence ~~expression~~ e1 must ~~end successfully~~ 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 ~~definitions~~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~~ rule2 is equivalent to ~~rule2a~~ 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~~ ended is applied. See Section 17.8.

1.49 Page 224, Section 17.7.11

~~The implication construct allows a user to monitor properties based on satisfying some criteria. Most common uses are to attach a precondition to a sequence, where the evaluation of the sequence is based on the success of a condition.~~

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

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*~~ antecedent *sequence_expr* can have zero, one, or more than one successful match.

— If there is no match of the ~~*antecedent*~~ 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*~~ antecedent *sequence_expr*, the ~~*consequent*~~ consequent *property_expr* is separately evaluated. The end point of the match of the ~~*antecedent*~~ antecedent *sequence_expr* is the start point of the evaluation of the ~~*consequent*~~ 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*~~ antecedent *sequence_expr* beginning at the start point, the evaluation of the ~~*consequent*~~ 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*~~ 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:

1.50 Page 226, Section 17.7.11

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 ~~expression~~ is evaluated. The sequence ~~expression~~:

1.51 Page 227, Section 17.7.11

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

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

1.52 Page 228, Section 17.8

it is desired to sample `x = e` at the match of b, the sequence ~~expression~~ can be rewritten as

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

1.53 Page 228, Section 17.8

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

1.54 Page 228, Section 17.8

As an example ~~the~~ 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 ~~sequence expression~~ property verifies this behavior:

1.55 Page 229,230 Section 17.8

Local variables can be passed into an instance of a ~~defined~~ named sequence to which ~~ended~~ `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 ~~defined~~ named sequence to which ~~ended~~ `ended` is applied:

Page 230

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

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

If a local variable is ~~sampled~~ assigned before being passed into an instance of a ~~defined~~ named sequence to which ~~ended~~ `ended` is applied, then the restrictions prevent this ~~sampled~~ assigned value from being visible within the ~~defined~~ named sequence. The restrictions are important because the use of ~~ended~~ `ended` means that there is no guaranteed relationship between the point in time at which the local variable is ~~sampled~~ assigned outside the ~~defined~~ 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 ~~defined~~ named sequence to which ~~ended~~ `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 ~~defined~~ 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~~ `ended` cannot have negation or any other expression operator applied to it.

1.56 Page 228, Section 17.8

The use of static SystemVerilog variables implies that only one copy exists. ~~Therefore, if~~ 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 ~~a~~ 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. ~~In other words~~ 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.

1.57 Page 228, Section 17.8

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

1.58 Page 229, Section 17.8

5) When valid_in is true, `x` is assigned to the value of pipe_in. ~~Property e is true if~~ If five cycles later, x pipe_out1 is equal to (x+1), then property `e` is true. Otherwise, property `e` is false. ~~Property e is false if pipe_out1 is not equal to (x+1).~~

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

1.59 Page 230, Section 17.8

Both conditions are satisfied by `sub_seq2` and `seq2a`. Thus, in `seq2a` the value in ~~v1~~`lv` in the comparison `do1 == v1` is the value ~~sampled into~~ 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 undefined
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~~ `ended` to this instance, and so the reference to `v1` in `do1 == v1` is to an unsampled variable.

In a single cycle, there can be multiple matches of a sequence instance to which ~~ended~~ `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~~ `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 ~~definition~~, 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:$] b[*0:$] ##1 b && (data_out == lv);
endsequence
```

~~There are special considerations on using local variables in parallel branches using operators **or**, **and**, and **intersect**.~~

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**, it is the intersection of the variables (names) that pass on past **or** operations. More precisely, a local variable passes the **or** if, and only if, it passes through both branches of **or** operations.~~ 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) ~~All succeeding threads out of **or** branches continue as separate threads, carrying with them their own lates samplings of the local variables.~~ 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**, the symmetric difference of the local variables that are sampled in the two joining threads passes on past the join. More precisely, a local variable that passes through at least one branch of the join shall be passed on past the join unless it is blocked. A local variable is blocked from passing on past the join if either:~~ 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 ~~sampled~~ assigned in and ~~passes through each branch of the join~~ flows out of each operand of the composite sequence. Or,

b) ~~The local variable is blocked from passing through at least one of the branches of the join~~ The local variable is blocked from flowing out of at least one of the operand sequences. ~~The value passed on is the latest sampled value. The two joining threads are merged into one thread at the join.~~

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
```

~~5) The intersection and difference of the sets of names should be computed statically at compile time.~~

1.60 Page 232, Section 17.10

~~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.~~

1.61 Page 233, Section 17.11

Title should be

~~**The property definition**~~ **Declaring properties**

1.62 Page 233, Section 17.11

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.

1.63 Page 234, Section 17.11

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. ~~The semantic~~ 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 ~~properties~~ property: sequence, negation, disjunction, conjunction, **if**...**else**, implication, and instantiation.

1.64 Page 235, Section 17.11

1) .....

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~~ **not** *property_expr* evaluates to false, and if *property_expr* evaluates to false, then **not** *property_expr* evaluates to true.

3) ...

4) ....

5) ....

6) ....

7) An instance of a ~~defined~~ 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 ~~defined~~ 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 ~~defined~~ named property is used as a *property_expr* operand for any property-building operator, then the ~~defined~~ named property must not have a **disable iff** clause. Similarly, clock events in a ~~defined~~ 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.

1.65 Page 236, Section 17.11

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 ~~during~~ 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.

This allows for the following examples:

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

Property `rule2` negates the sequence `(b ##1 c ##1 d)` in the consequent of the implication ~~result of the implication (a |-> b ##1 c ##1 d) for every attempt~~. `clkev` specifies the clock for the property.

1.66 Page 237, Section 17.11

A named property can be instantiated by referencing its name. ~~referenced by instantiating 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.

1.67 Page 237, Section 17.11.1

SystemVerilog allows recursive properties ~~property definitions~~. A ~~defined~~ 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.

1.68 Page 238, Section 17.11.1

More generally, several properties ~~property definitions~~ 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 ~~definitions~~.

RESTRICTION 1: The negation operator ~~not~~ **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 ~~definitions~~ 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 ~~definition~~ 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 ~~definition~~ 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 ~~definition~~ 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 ~~definition~~ that violates Restriction 3:

1.69 Page 239, Section 17.11.1

Recursive properties can ~~deal with~~ 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.

1.70 Page 239, Section 17.11.1

— 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~~

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.

1.71 Page 239, Section 17.11.1

— The last data ~~valid~~ 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 ~~valid~~ beat, a write transaction can be forced to retry. Retry is indicated by the signal ~~retry~~ `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 ~~rerequest~~ 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 ~~valid~~ 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
```

1.72 Section 17.12

Note for the editor: In this section, the use of ## alone without a numeric operand has been changed to ##1. This change is included in the following edits.

1.73 Page 241, Section 17.12.1

~~Multiply-clocked sequences are built by concatenating singly-clocked subsequences using the multi-clock concatenation operator ##. This operator is non-overlapping and synchronizes between the ending clock tick of the left-hand sequence and the strictly subsequent starting clock tick of the right-hand sequence.~~

Multiply-clocked sequences are built by concatenating singly-clocked sequences 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 single-clocked sequence

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

When concatenating differently-clocked, the maximal singly-clocked subsequences ~~using the multi-clock concatenation operator ##, both operands~~ are required to admit only non-empty matches. Thus, if s1, s2 are sequences ~~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~~ **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 ~~to operands that do not match the empty word when using ##~~ 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~~the multi-clock concatenation operator ##~~. For example, if clk1 and clk2 are not identical, then the following all are illegal ~~the following is illegal~~:

```
(@(posedge clk1) x ##1 y) ##0 (@(posedge clk2) z[*1:$])
(@(posedge clk1) x ##1 y) ##2 (@(posedge clk2) z[*1:$])
(@(posedge clk1) x ##1 y) intersect intersect (@(posedge clk2) z[*1:$])
```

1.74 Page 242, Section 17.12.2

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

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

1.75 Page 242, Section 17.12.2

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 ~~expressions~~ with no clocking event, then in

1.76 Page 243, Section 17.12.2

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

is illegal, but

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

1.77 Page 243, Section 17.12.2

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 ~~expressions~~ with no clocking events, then

1.78 Page 243, Section 17.12.2

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 ~~expressions~~ with no clocking events, then

1.79 Page 243, Section 17.12.3

Throughout this subsection, `c`, `d` *c*, *d* denote clocking event expressions and `v`, `w`, `x`, `y`, `z` *v*, *w*, *x*, *y*, *z* denote

sequences ~~expressions~~ with no clocking events.

1.80 Page 243, Section 17.12.3

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
```

1.81 Page 244, Section 17.12.3

```
@(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
```

1.82 Page 244, Section 17.12.3

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 sub-sequence (*y* ##1 @(*d*) *z*), and across |=>. Clock *c* also flows into the parenthesized sub-sequence, 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 ~~defined~~ 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~~definition~~ of a sequence or property does not flow out of an instance of that sequence or property.

1.83 Page 245, Section 17.12.4

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
```

1.84 Page 245, Section 17.12.4

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.

1.85 Page 246, Section 17.12.4

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

1.86 Page 246, Section 17.12.5

Local variables can be passed into an instance of a named ~~defined~~ 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 ~~defined~~ sequence to which matched is applied will flow out under the same conditions as for ended. See Section 17.8.

1.87 Page 246, Section 17.12.5

Local variables can be passed into an instance of a defined 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 ~~defined~~ sequence to which matched is applied will flow out under the same conditions as for ended. See Section 17.8.

1.88 Page 254, Section 17.14

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

1.89 Page 254, Section 17.14

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

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

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

Table 17-4: Resolution of clock for a property declaration ~~definition~~

1.90 Page 256, Section 17.14.1

Throughout this subsection, `s`, `s1`, `s2` *s*, *s1*, *s2* denote sequences ~~expressions~~ without clocking events; `p`, `p1`, `p2` *p*, *p1*, *p2* denote property expressions without clocking events; `m`, `m1`, `m2` *m*, *m1*, *m2* denote multiply-clocked sequences, `q`, `q1`, `q2` *q*, *q1*, *q2* denote multiply-clocked properties ~~property expressions~~; and `c`, `c1`, `c2` *c*, *c1*, *c2* denote non-identical clocking event expressions.

1.91 Page 257, Section 17.14.1

— 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 ~~definition~~ by substituting in actual arguments.

1.92 Page 535, Section H.2.1

The abstract grammar for clocked sequences is

```
S ::= @(b) R // "clock" form
| ( S ) // "parenthesized" form
| ( S ##1 S ) // "concatenation" form
```

1.93 Page 536, Section H.2.3.1

- (R_1 |=> P) ((R_1 ##1 1) |-> P) .
- (S_1 |=> Q) ((S_1 ##1 @(1) 1) |-> Q)

1.94 Page 538, Section H.3.1

- (S_1 ##1 S_2)---> (S_1 ##1 S_2) .

