Additional Temporal Logic Operators

Aligned with p1800-2008-draft 4

Objectives:

Currently SVA supports properties formed from sequences, suffix implications, property operators and, or, not, if-else, and recursion. Although recursion provides much expressive power, the verification community that has used assertions in the past is also accustomed to Linear Temporal Logic. Introducing such operators to SVA would provide a significant advantage by enabling flexible forms for expressing properties, and reducing the need for indirect property definitions created through negation. Such operators are also part of the PSL standard. SVA would have the advantage of combining the proposed operators with recursion and local variables.

Furthermore, the dual forms (so called *followed_by*) of the suffix implications are also included in the proposal, using the syntax #-# and #=# for the overlapping and non-overlapping versions, respectively. These operators are particularly useful when creating coverage properties in which regular concatenation (##) cannot be used.

Some examples:

- SVA: !g[*1:$] |-> f // weak until
  LTL: f **until** g

- SVA: **property** prop_always(p); p **and** (1'b1 |=> prop_always(p)); **endproperty**
  LTL: **property** prop_always(p); **always** p; **endproperty**;

- SVA: **not** (##[1:$] (e) |-> ##[1:$] !(e));
  LTL: **eventually always** e;

Another important application of the proposed operators is their ability to make property libraries more generic, since their plug-in capabilities are not restricted. As an example consider a property saying that some condition must hold between the start event and the end event. In the sequence-based implementation

```
property between(start_ev, end_ev, cond);
    start_ev ##0 !(end_ev && cond)[*1:$] |-> cond;
endproperty : between
```

`start_ev` may be any sequence, but the `end_ev` must be a boolean, since it is negated in the formula `!(end_ev && cond)`. In the implementation using the proposed operators no limitations are imposed on `end_ev`:

```
property between(start_ev, end_ev, cond);
    start_ev |-> cond until_with end_ev;
endproperty : between
```

(Note also that the new form is more intuitive than the original one.)

In addition to introducing the new operators in this proposal, the document also defines the concept of strong and weak sequences. It proposes to change the interpretation of sequence properties from the default "strong"

to "weak". The latter is the more usual interpretation in simulation and also the default in PSL. The keyword **strong** is introduced to this end and it can be applied to sequence expressions.

The existing definition in SVA is not intuitive: even innocently looking assertions turn out to be liveness. Consider the following example:

```
a1: assert property (@(posedge clk) a);
```

The assertion a1 is liveness, since it requires the clock to tick infinitely often. To make this assertion safety one should rewrite it as

```
a1: assert property (@(posedge clk) 1'b1 |-> not(!a));
```

which is awkward. It is likely that the existing tools simply ignore the liveness part of this assertion, but it is not compliant to the definition of the formal semantics in Annex F. Making a sequence property weak by default addresses this problem.

16.12 Declaring properties

Syntax 16-14—Property construct syntax (excerpt from Annex A)

CHANGE TO

concurrent_assertion_item_declaration ::=
property_declaration
...
property_spec ::=
[clocking_event] [**disable iff** (expression_or_dist)] property_expr
property_expr ::=
sequence_expr
| **strong** (sequence_expr)
| **weak** (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]
| sequence_expr **#-#** property_expr
| sequence_expr **#=#** property_expr
| **next** property_expr
| **next** [constant _expression] property_expr
| **s_next** property_expr
| **s_next** [constant_expression] property_expr
| **always** property_expr
| **always** [cycle_delay_const_range_expression] property_expr

| **s_always** [constant_range] property_expr
| s_**eventually** property_expr
| **eventually** [constant_range] property_expr
| **s_eventually** [cycle_delay_const_range_expression] property_expr
| property_expr **until** property_expr
| property_expr **s_until** property_expr
| property_expr **until_with** property_expr
| property_expr **s_until_with** property_expr
| property_expr **implies** property_expr
| property_expr **iff** property_expr
| **accept_on** (expression_or_dist) property_expr Note: from Mantis #1757
| **reject_on** (expression_or_dist) property_expr Note: from Mantis #1757
| property_instance
| clocking_event property_expr

assertion_variable_declaration ::=
var_data_type list_of_variable_identifiers ;

…

REPLACE

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.

a) A property that is a sequence evaluates to true if, and only if, there is a nonempty match of the sequence. A sequence that admits an empty match is not allowed as a property. Because 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.

b) 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.

c) 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.

d) 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.

e) 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.

f) 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 16.12.2.

g) 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 multiclock support when the property is instantiated in a *property_expr* or *property_spec* that also involves other clock events.

Table 16-25 lists the sequence and property operators from highest to lowest precedence and shows the associativity of the non-unary operators.

Sequence operators	Property operators	Associativity
`[*], [=], [->]`		—
`##`		Left
`throughout`		Right
`within`		Left
`intersect`		Left
	`not`	—
`And`	**`and`**	Left
`Or`	**`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 preemptive 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 results in disabled. A property has disabled evaluation if it was preempted due to a **`disable iff`** condition. A disabled evaluation of a property does not result in success or failure. 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*. The values of variables used in the reset expression are those in the current simulation cycle, i.e., not sampled. The expression may contain a reference to an end point of a sequence by using the method *triggered* of that sequence. *Matched* and *ended* of a sequence and local variables cannot be used in the reset expression. If a sampled value function is used in the reset expression, the sampling clock must be explicitly specified in its actual argument list as described in 16.8.3. Nesting of **`disable iff`** clauses, explicitly or through property instantiations, is not allowed.

WITH

The result of property evaluation is either true or false. Properties may be built from other properties or sequences using instantiation, and the operators described in the following subclauses. ~~There are seven kinds of property: sequence, negation, disjunction, conjunction, if...else, implication, and instantiation~~.

~~a) A property that is a sequence evaluates to true if, and only if, there is a nonempty match of the sequence. A sequence that admits an empty match is not allowed as a property. Because 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.~~

~~b) 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.~~

~~c) 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.~~

~~d) 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.~~

~~e) 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.~~

~~f) 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 16.12.2.~~

~~g) 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 multiclock support when the property is instantiated in a property_expr or property_spec that also involves other clock events.~~

Table 16-25 lists the sequence and property operators from highest to lowest precedence and shows the associativity of the non-unary operators. The precedence for the strong and weak sequence operators is not defined because these operators require parenthesis.

Sequence operators	Property operators	Associativity
`[*], [=], [->]`		—
`##`		Left
`throughout`		Right
`within`		Left
`intersect`		Left
	`not, next, s_next`	—
`and`	**`and`**	Left
`or`	**`or`**	Left
	`iff`	Right
	`until, s_until, until_with, s_until_with, implies`	Right
	`|->, |=>, #-#, #=#`	Right
	`always, s_always, eventually, s_eventually, if-else, accept_on, reject_on`	—

A **`disable iff`** clause ~~can~~ may 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 preemptive 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 results in disabled. A property has disabled evaluation if it was preempted due to a **`disable iff`** condition. A disabled evaluation of a property does not result in success or failure. 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*. The values of variables used in the reset expression are those in the current simulation cycle, i.e., not sampled. The expression may contain a reference to an end point of a sequence by using the method *triggered* of that sequence. *Matched* and *ended* of a sequence and local variables cannot be used in the reset expression. If a sampled value function is used in the reset expression, the sampling clock must be explicitly specified in its

actual argument list as described in 16.8.3. Nesting of **disable iff** clauses, explicitly or through property instantiations, is not allowed.

ADD before 16.12.1 Typed formal arguments in property declarations

16.12.1 Sequence property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

Sequence properties have three forms: *sequence_expr,* **weak**(*sequence_expr*) and **strong**(*sequence_expr*). The **strong** and **weak** operators are called *sequence operators*. **strong**(*sequence_expr*) evaluates to true if, and only if, there is a nonempty match of the sequence. **weak**(*sequence_expr*) evaluates to true if, and only if, there is no finite prefix that witnesses inability to match the sequence. The sequence_expr of a sequential property shall not admit an empty match.

If the **strong** or **weak** operator is omitted, then the evaluation of the *sequence_expr* depends on the verification statement in which it is used. If the verification statement is **assert property** or **assume property**, then the *sequence_expr* is evaluated as **weak**(*sequence_expr*). Otherwise, the *sequence_expr* is evaluated as **strong**(*sequence_expr*).

Since only one match of a sequence_expr needed for **strong(sequence_expr)** to hold, a poperty of the form **strong**(*sequence_expr*) evaluates to *true* if and only if the property **strong**(**first_match**(*sequence_expr*)) evaluates to *true*.

Similarly, a property of the form **weak**(*sequence_expr*) evaluates to true if and only if the property **weak**(**first_match**(*sequence_expr*)) evaluates to true. This is because a prefix witnesses inability to match *sequence_expr* if, and only if, it witnesses inability to match **first_match**(*sequence_expr*).

The following examples illustrate the sequential property forms:

```
property p3;
   b ##1 c;
endproperty

c1:  cover property  (@(posedge clk) a #-# p3);
a1:  assert property (@(posedge clk) a |-> p3);
```

The sequential property `p3` is interpreted as **strong** in the **cover property** `c1`. An evaluation attempt of `c1` returns true if, and only if, `a` is true at the tick of **posedge** `clk` at which the attempt begins and both of the following conditions are satisfied:

- `b` is true at the tick of **posedge** `clk` at which the attempt begins.
- There exists a subsequent tick of **posedge** `clk` and **c** is true at the first such tick.

The sequential property `p3` is interpreted as **weak** in the **assert property** `a1`. An evaluation attempt of `a1` returns true if, and only if, either `a` is false at the tick of **posedge** `clk` at which the attempt begins or both of the following conditions are satisfied:

- `b` is true at the tick of **posedge** `clk` at which the attempt begins.
- If there exists a subsequent tick of **posedge** `clk`, then `c` is true at the first such tick.

The **not** operator switches the strength of a property. In particular one should be careful when negating a sequence. For example consider the following assertion:

```
a1: assert property (not a ##1 b);
```

Since the sequential property `a ##1 b` is used in an assertion, it is weak. This means that if `a` holds at the last tick of the assertion's clock in a simulation, the weak sequential property `a ##1 b` will also hold beginning at that tick, and so the assertion `a1` will fail. In this case it is more reasonable to use:

```
a2: assert property (not strong(a ##1 b));
```

16.12.2 Negation property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

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.

16.12.3 Disjunction property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

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.

16.12.4 Conjunction property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

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.

16.12.5 If-else property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

A property is an **if–else** if it has either the form

```
if (expression_or_dist) property_expr
```

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_expr` 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.

Note to editor: Insert the subclause entitled "Implication" (16.12.2in Draft 4) here as 16.12.6

16.12.7 Implies and iff properties

Note to editor: Shift the numeration of the subsequent subclauses accordingly

A property is an **`implies`** if it has the form `property_expr1` **`implies`** `property_expr2`

A property of this form evaluates to true if and only if either `property_expr1` evaluates to false or `property_expr2` evaluates to true.

A property is an **`iff`** if it has the form `property_expr1` **`iff`** `property_expr2`

A property of this form evaluates to true if and only if either both `property_expr1` evaluates to false and `property_expr2` evaluates to false or both `property_expr1` evaluates to true and `property_expr2` evaluates to true.

16.12.8 Property instantiation

Note to editor: Shift the numeration of the subsequent subclauses accordingly

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*. 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 multiclock support when the property is instantiated in a *property_expr* or *property_spec* that also involves other clock events.

16.12.9 Followed_by property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

A property is a **`followed_by`** if it has one of the following forms, which use the *followed_by* operators:

---

property_expr ::= *// from A.2.10*
...
| sequence_expr **#-#**property_expr
| sequence_expr **#=#** property_expr

Syntax [Note to editor put right numbering]—followed_by 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 followed_by is either true or false. The left-hand operand *sequence_expr* is called the *antecedent*, while the right-hand operand *property_expr* is called the *consequent*. The following points should be noted for `#-#` followed_by:

- From a given start point *sequence_expr* must have at least one successful match.
- *`property_expr`* shall be successfully evaluated starting from the end point of some successful match of *`sequence_expr`*.
- From a given start point, evaluation of the followed_by succeeds and returns true if, and only if, there exists a match of the antecedent *sequence_expr* beginning at the start point, and the evaluation of the consequent *property_expr* beginning at the end point of the match succeeds and returns true.

Two forms of followed_by are provided: Overlapped using operator `#-#` and nonoverlapped using operator `#=#`. For overlapped **`followed_by`**, there shall be a match for the antecedent *sequence_expr*, where the end point of this match is the start point of the evaluation of the consequent *property_expr*. For nonoverlapped **`followed_by`**, 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 the following:

```
sequence_expr ##1 1 #-# property_expr
```

The followed_by operators are the duals of the implication operators. Therefore, `sequence_expr #-# property_expr` is equivalent to the following:

```
not (sequence_expr |-> not property_expr)
```

and `sequence_expr #=# property_expr` is equivalent to the following:

```
not (sequence_expr |=> not property_expr)
```

Examples:

```
property p1;
      ##[0:5] done #-# always !rst;
endproperty
property p2;
      ##[0:5] done #=# always !rst;
endproperty
```

Property `p1` says that `done` shall be asserted at some clock tick during the first 6 clock ticks, and starting from one of the clock ticks when `done` is asserted, `rst` shall always be low. Property `p2` says that `done` shall be asserted at some clock tick during the first 6 clock ticks, and starting the clock tick after one of the clock ticks when `done` is asserted, `rst` shall always be low.

`sequence_expr` **`#-#`** **`strong`**`(sequence_expr1)` is semantically equivalent to **`strong`**`(sequence_expr ##0 sequence_expr1)`, and `sequence_expr #=#` **`strong`**`(sequence_expr1)` is semantically equivalent to **`strong`**`(sequence_expr ##1 sequence_expr1)`.

A **`followed_by`** operator is especially convenient for specifying a **`cover property`** directive over a sequence followed by a property.

16.12.10 Next property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

A property is a **`next`** if it has one of the following forms, which use the *next operators*:

- **`next`** `property_expr` (weak next operator)

  The weak next property **`next`** *property_expr* evaluates to true if, and only if, either the *property_expr* evaluates to true beginning at the next clock tick or there is no further clock tick.

- **next** `[const_expression] property_expr` (indexed form of weak next)

  The indexed weak next property **next** [*const_expression*] *property_expr* evaluates to true if, and only if, either there are not *const_expression* clock ticks or *property_expr* evaluates true beginning at the last of the next *const_expression* clock ticks.

- **s_next** `property_expr` (strong next)

  The strong next property **s_next** *property_expr* evaluates to true if, and only if, there exists a next clock tick and *property_expr* evaluates to true beginning at that clock tick.

- **s_next** `[const_expression] property_expr` (indexed form of strong next)

  The indexed strong next property **s_next** [*const_expression*] *property_expr* evaluates to true if, and only if, there exist *const_expression* clock ticks and *property_expr* evaluates true beginning at the last of the next *const_expression* clock ticks.

Examples.

```
// if the clock ticks once more, then a shall be true at the next clock tick
property p1;
      next a;
endproperty


// the clock shall tick once more and a shall be true at the next clock tick.
property p2;
      s_next a;
endproperty


// as long as the clock ticks, a shall be true at each future clock tick starting
// from the next clock tick.
property p3;
      next always a;
endproperty
// the clock shall tick at least once more and as long as it ticks, a shall be
// true at every clock tick starting from the next one.
property p4;
      s_next always a;
endproperty


// if the clock ticks at least once more, it shall tick enough times for a to be
// true at some point in the future starting from the next clock tick.
property p5;
      next s_eventually a;
endproperty
```

```
// a shall be true sometime in the strict future.
property p6;
       s_next s_eventually a;
endproperty


// if there are at least two more clock ticks, a shall be true at the second
// future clock tick
property p7;
       next [2] a;
endproperty


// there shall be at least two more clock ticks, and a shall be true at the second
// future clock tick
property p8;
       s_next [2] a;
endproperty
```

16.12.11 Always property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

A property is an **always** if it has one of the following forms, which use the *always operators*:

- **always** property_expr

  A property **always** *property_expr* evaluates to true if, and only if, *property_expr* holds at every current or future clock tick.

- **always** [cycle_delay_const_range_expression] property_expr (ranged form of always)

  A property **always** [*cycle_delay_const_range_expression*] *property_expr* evaluates to true if, and only if, *property_expr* holds at every current or future clock tick that is within the range of clock ticks specified by *cycle_delay_const_range_expression*. It is not required that all clock ticks within this range exist.

- **s_always** [constant_range] property_expr (ranged strong form of s_always)

  A property **s_always** [*constant_range*] *property_expr* evaluates to true if, and only if, all current or future clock ticks specified by *constant_range* exist and *property_expr* holds at each of these clock ticks.

The strong form of always operator is allowed only with a bounded range.

There is also the implicit **always** that is associated with the verification statements (see 16.13.4). A verification statement that is not placed inside an initial block specifies that an evaluation attempt of its top-level property shall begin at each occurrence of its leading clocking event. In the following two examples, there is a one-to-one correspondence between the evaluation attempts of `p` specified by the implicit always from the verification statement `implicit_always` and the evaluation attempts of `p` specified by the explicit always operator in `excplicit_always`:

Implicit form:

```
Implicit_always: assert property(p);
```

Explicit form:

```
initial explicit_always: assert property(always p);
```

This is not shown as a practical example, but only for illustration of the meaning of **always**.

Examples.

```
initial a1: assume property( @(posedge clk) reset[*5] #=# always !reset);
property p1;
       a ##1 b |=> always c;
endproperty
property p2;
       always [2:5] a;
endproperty
property p3;
       s_always [2:5] a;
endproperty
property p4;
       always [2:$] a;
endproperty
property p5;
       s_always [2:$] a; // Illegal
endproperty
```

The assertion `a1` says that `reset` shall be true for the first 5 clock ticks and then remain 0 for the rest of the computation. The assumption is being evaluated once starting at the first clock tick. The property `p1` says that if `a` is true at the first clock tick and `b` is true at the second clock tick, then `c` shall be true at every clock tick that follows the second. The properties `p2` and `p3` say that `a` shall be true at each of the second through fifth clock ticks after the starting clock tick of the evaluation attempt. Property `p3` requires that these clock ticks exist, while property `p2` does not. The property `p4` evaluates to true if, and only if, `a` is true at every clock tick that is at least two clock ticks after the starting clock tick of the evaluation attempt. These clock ticks are not required to exist. The property `p5` is illegal since specifying an unbounded range is not permitted with the strong form of an **always** property.

16.12.12 Until property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

A property is an **until** if it has one of the following forms, which use the *until operators*:

- `property_expr1 until property_expr2` (weak non-overlapping form)
- `property_expr1 until_with property_expr2` (weak overlapping form)
- `property_expr1 s_until property_expr2` (strong non-overlapping form)

- `property_expr1` **`s_until_with`** `property_expr2` (strong overlapping form)

An **until** property of the non-overlapping form evaluates to true if `property_expr1` evaluates to true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until at least one tick before a clock tick where `property_expr2` evaluates to true. An until property of one of the overlapping forms evaluates to true if `property_expr1` evaluates to true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until and including a clock tick at which `property_expr2` evaluates to true. An until property of one of the strong forms requires a current or future clock tick exist at which *property_expr2* evaluates to true, while an until property of one of the weak forms does not make this requirement. An until property of one of the weak forms evaluates to true if *property_expr1* evaluates to true at each clock tick, even if *property_expr2* never holds.

Examples.

```
property p1;
       a until b;
endproperty
property p2;
       a s_until b;
endproperty
property p3;
       a until_with b;
endproperty
property p4;
       a s_until_with b;
endproperty
```

The property `p1` says that `a` shall be true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until, but not necessarily including, a clock tick at which `b` is true. If there is no current or future clock tick at which `b` is true, than `a` shall be true at every current or future clock tick. If `b` is true at the starting clock tick of the evaluation attempt, then `a` need not be true at that clock tick. The property `p2` says that there shall exist a current or future clock tick at which `b` is true and that a shall be true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until, but not necessarily including, the clock tick at which `b` is true. If `b` is true at the starting clock tick of the evaluation attempt, then `a` need not be true at that clock tick. The property `p3` says that a shall be true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until and including a clock tick at which `b` is true. If there is no current or future clock tick at which b is true, than `a` shall be true at every current or future clock tick. The property `p4` says that there shall exist a current or future clock tick at which `b` is true and that `a` shall be true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until and including the clock tick at which `b` is true. The property `p4` is equivalent to **`strong`**`(a[*1:$] ##0 b)`.

16.12.13 Eventually property

Note to editor: Shift the numeration of the subsequent subclauses accordingly

A property is an **eventually** if it has one of the following forms, which use the *eventually operators:*

```
s_eventually property_expr
eventually [constant_range] property_expr
                                (ranged weak form of eventually)
```

s_eventually [cycle_delay_const_range_expression] property_expr

(ranged strong form of eventually)

The weak form of eventually operator is allowed only with a bounded range.

The **s_eventually** *property_expr* evaluates to true if, and only if, there exists a current or future clock tick at which *property_expr* evaluates to true. The ranged weak eventually property **eventually** [*constant_range*] *property_expr* evaluates to true if, and only if, either there exists a current or future clock tick within the range specified by *constant_range* at which *property_expr* evaluates to true or not all the current or future clock ticks within the range specified by *constant_range* exist. The ranged strong eventually property **s_eventually** [*cycle_delay_const_range_expression*] *property_expr* evaluates to true if, and only if, there exists a current or future clock tick within the range specified by *cycle_delay_const_range_expression* at which *property_expr* evaluates to true. The range for a strong eventually may be unbounded, but the range for a weak eventually shall be bounded.

Examples.

```
property p1;
       s_eventually a;
endproperty
property p2;
       s_eventually always a;
endproperty
property p3;
       always s_eventually a;
endproperty
property p4;
       eventually [2:5] a;
endproperty
property p5;
       s_eventually [2:5] a;
endproperty
property p6;
       eventually [2:$] a; // Illegal
endproperty
property p7;
       s_eventually [2:$] a;
endproperty
```

The property `p1` says that there shall exist a current or future clock tick at which `a` is true. It is equivalent to **strong**`(##[*0:$] a)`. The property `p2` says that there shall exist a current or future clock tick such that `a` is true both at that clock tick and also at every subsequent clock tick. For a computation with infinitely many clock ticks, the property `p3` says that `a` shall be true at infinitely many of those clock ticks. For a computation with finitely many clock ticks, the property `p3` says that if there is at least one clock tick, then `a` shall hold at the last clock tick. The property `p4` says that if the second through fifth clock ticks from the starting clock tick of the evaluation attempt all exist, then a shall be true at one of these clock ticks. `p4` is equivalent to **weak**`(##[2:5] a)`. The property `p5` says that there shall exist a clock tick at which `a` is true and that is

between the second and fifth clock ticks, inclusive, from the starting clock tick of the evaluation attempt. `p5` is equivalent to **`strong`**`(##[2:5] a)`. The property `p7` says that there shall exist a clock tick at which `a` is true and that is no earlier than the second clock tick after the starting clock tick of the evaluation attempt.

If 1757 pass, the following adjustment to 1757 should be done and the modified text should be placed here:

16.12.14 Abort Properties

Note to editor: Shift the numeration of the subsequent subclauses accordingly.

~~pp 342, add after g)~~

h) A property is an abort if it has either the form:

— **`accept_on`**`(`*expression_or_dist*`)` *property_expr*

or the form

— **`reject_on`**`(`*expression_or_dist*`)` *property_expr*

where the *expression_or_dist* is called the abort condition.

For an evaluation of **`accept_on`**`(`*expression_or_dist*`)` *property_expr*, there is an evaluation of the underlying *property_expr*. If during the evaluation, the abort condition becomes true, then the overall evaluation of the property results in *true*. Otherwise, the overall evaluation of the property is equal to the evaluation of the *property_expr*.

For an evaluation of **`reject_on`**`(`*expression_or_dist*`)` *property_expr*, there is an evaluation of the underlying *property_expr*. If during the evaluation, the abort condition becomes true, then the overall evaluation of the property results in *false*. Otherwise, the overall evaluation of the property is equal to the evaluation of the *property_expr*.

~~The meaning of **`accept_on`** and **`reject_on`** is further discussed in 16.12.3.~~

~~Insert 16.12.3 Abort properties~~

~~(Note to the editor: shift clause numbering)~~

The operators **`accept_on`** and **`reject_on`** are evaluated at the granularity of the simulation time step like **`disable iff`** but their abort condition is evaluated using sampled value as a regular boolean expression in assertions. The operators **`accept_on`** and **`reject_on`** represent asynchronous resets.

The semantics of **`accept_on`** is similar to **`disable iff`**, except for the following three differences:

— **`accept_on`** operates at the property level rather than the verification statement level.
— **`accept_on`** uses sampled values.
— While a disable condition of a **`disable iff`** in a *property_spec* may cause an evaluation of the *property_spec* to be disabled, an abort condition of **`accept_on`** in a *property_expr* may cause the evaluation of the *property_expr* to be true.

The semantics of **`reject_on`**`(`*expression_or_dist*`)` *property_expr* is the same as **`not`**`(`**`accept_on`**`(`*expression_or_dist*`)` **`not`**`(`*property_expr*`))`.

Any nesting of **`accept_on`** and **`reject_on`** operators is allowed.
For example, whenever `go` is high, followed by two occurrences of `get` being high, then `stop` cannot be high until after `put` is asserted twice (not necessarily consecutive).

```
assert property (@(clk) go ##1 get[*2] |-> reject_on(stop) put[->2]);
```

When the abort condition occurs at the same time step where the evaluation of the *property_expr* ends, the abort condition takes precedence.

For example,

```
property p; (accept_on(a) p1) and (reject_on(b) p2); endproperty
```

If a becomes true during the evaluation of p1, the first term is ignored in deciding the truth of p. On the other hand, if b becomes true during the evaluation of p2 then p evaluates to false.

```
property p; (accept_on(a) p1) or (reject_on(b) p2); endproperty
```

If a becomes true during the evaluation of p1 then p evaluates to true. On the other hand, if b becomes true during the evaluation of p2, then the second term is ignored in deciding the truth of p.

```
property p; not (accept_on(a) p1); endproperty
```

not inverts the effect of the abort operator. Therefore, if a becomes true while evaluating p1, property p evaluates to false.

Nested **reject_on** and **accept_on** operators are evaluated in the lexical order (left to right). Therefore, if two nested operator conditions become true in the same time step during the evaluation of the argument property, then the outermost operator takes precedence. For example,

```
property p; accept_on(a) reject_on(b) p1; endproperty
```

if a becomes true in the same time step as b and during the evaluation of p1, then p succeeds in that time step. If b becomes true before a and during the evaluation of p1, then p fails.

reject_on and **accept_on** abort expressions may contain sampled value functions (see 16.8.3). When sampled value functions other than **$sampled** are used in the abort condition, the clock argument shall be explicitly specified. Abort expressions shall not contain any reference to local variables and the sequence methods ended, triggered and matched.

16.12.15 Weak and strong operators

Note to editor: Shift the numeration of the subsequent subclauses accordingly

The property operators **s_next, s_always, s_eventually, s_until, s_until_with** and strong sequence are strong: they require that some terminating condition happen in the future, and this includes the requirement that the property clock tick enough time to enable the condition to happen. The property operators **next, always, until, eventually, until_with** and weak sequence are weak, they don't impose any requirement on the terminating condition, and don't require the clock to tick.

The concept of weak and strong operators is closely related to an important notion of safety properties. Safety properties have the characteristic that all their failures happen at a finite time. E.g., the property **always** a is safety since it is violated only if after finitely many clock ticks there is a clock tick at which a is false, even if there are infinitely many clock ticks in the computation. To the contrary, a failure of the property **s_eventually** a on a computation with infinitely many clock ticks cannot be identified at a finite time: if it is violated, the value of a must be false at each of the infinitely many clock ticks.

16.12.16 Recursive properties

Note to editor: Shift the numeration accordingly

REPLACE

— 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.

WITH

— RESTRICTION 1: The negation operator **not** and strong `operators` **`s_next, s_eventually, s_always, s_until,`** and **`s_until_with`** 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.

16.13.2 Multiclocked properties

Note to the editor: if 1683 does not pass then

Replace

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

With

The boolean property operators (**`not`**, **`and`**, **`or`**) as well as the abort operators **`(accept_on, reject_on)`** may ~~can~~ be used freely to combine singly clocked and multiclocked properties. The meanings of ~~the boolean~~ these property operators are the usual ones, just as in the case of singly clocked properties. For example:

Note to the editor: if 1683 does not pass then

Replace

Because 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`.
Because **`|->`** 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`, and `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`, and `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 from the **`if`** condition**.**

With

Because 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 **`|->`**, the overlapping followed by `#-#`,~~and~~ `if/if-else`, and the next, always, until, and eventually operators. Because **`|->`** and `#-#` overlap~~s~~ the end of ~~it's~~ the antecedent with the beginning of ~~it's~~ the 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`, and `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`, and `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 from the **`if`** condition**.**

Each of the next, always, until, and eventually operators specifies evaluation of its underlying operand properties at current and future ticks of clock incoming to the temporal operator. Therefore the leading clock of each underlying operand property shall be the same as the clock incoming to the temporal operator.

Formatted: Keyword, Font: Times New Roman, Font color: Blue

Formatted: 2.DRAFT, Font: Times New Roman

Formatted: Keyword, Font: Times New Roman, Font color: Blue

Formatted: 2.DRAFT, Font: Times New Roman

Formatted: Keyword, Font: Times New Roman, Font color: Blue

Formatted: 2.DRAFT, Font: Times New Roman, Check spelling and grammar

Formatted: 2.DRAFT, Font: Times New Roman

Therefore, the operands properties shall begin on the same clock as the clock of the underlying property. For example, if `clk0` and `clk1` are not identical and `s1`, and `s2` are sequences with no clocking events, then

```
@(posedge clk0) (@(posedge clk0)s1 until @(posedge clk0) s2)
```

is legal, but

```
@(posedge clk0) ( (@(posedge clk1) s1) until @(posedge clk0) s2)
```

is illegal because the leading clock of the operand property `@(posedge clk1) s1` is different from the clock incoming to the **`until`** operator.

16.13.3 Clock flow

Replace

Intuitively, clock flow provides that in a multiclocked 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.

With

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

16.15.1 Clock resolution in multiclocked properties

Replace

— The set of semantic leading clocks of `if` (*b*) *q1* else *q2* is {*inherited*}.

with

— The set of semantic leading clocks of `if` (*b*) *q1* else *q2* is {*inherited*}.

— The set of semantic leading clocks of **`next`** `q` is {*inherited*}.

— The set of semantic leading clocks of **`always`** `q` is {*inherited*}.

— The set of semantic leading clocks of **`eventually`** `q` is {*inherited*}.

— The set of semantic leading clocks of `q1 until q2` is {*inherited*}.

— The set of semantic leading clocks of `accept_on(b) q` is the set of semantic leading clocks of `q`.

— The set of semantic leading clocks of `reject_on(b) q` is the set of semantic leading clocks of `q`.

Note to the editor: if 1683 does not pass then

Replace

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

a) Multiclocked 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 multiclocked overlapped implication `m |-> q` is legal for incoming clock *c* if, and only if, the following two conditions are met:

1) Every explicit semantic leading clock of *q* is identical to the ending clock of *m*.
2) 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 ##1 (@(c1) s1) |-> p
```

is not legal because the set of semantic leading clocks of *p* is {*inherited*}, the ending clock of the antecedent is *c1*, 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 ##1 @(c1) s1 |-> p1 or @(c1) p2
```

are both legal.

b) Multiclocked **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*; therefore, the multiclocked 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.

With

The rules for using multiclocked overlapping implication/followed_by, ~~and~~ **if/if-else**, and the next, always, until, and eventually operators in the presence of an incoming outer clock can now be stated more precisely.

a) Multiclocked overlapping implication and followed_by.

Let c be the incoming outer clock. Then the clocking of *m* |-> *q* (respectively m #-# q) is equivalent to the clocking of @(c) *m* |-> *q* (respectively @(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 **|->** (respectively #-#). The multiclocked overlapped implication *m* **|->** *q* (respectively m #-# q) is legal for incoming clock *c* if, and only if, the following two conditions are met:

1) Every explicit semantic leading clock of *q* is identical to the ending clock of *m*.
2) If *inherited* is a semantic leading clock of *q*, then the ending clock of *m* is equal to the clock that flows across |-> (respectively #-#).

For example:

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

is not legal because the ending clock of the antecedent is *c*, while the consequent has c2 as an explicit semantic leading clock.
Also,

```
@(c) s ##1 (@(c1) s1) |->#-# p
```

is not legal because the set of semantic leading clocks of *p* is {*inherited*}, the ending clock of the antecedent is *c1*, 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 ##1 @(c1) s1 |->#-# p1 or @(c1) p2
```

are both legal.

b) Multiclocked **`if/if-else`**
Let `c` be the incoming outer clock. Then the clocking of **`if`** `(b) q1 [`**`else`** `q2]` is equivalent to the clocking of

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

The boolean condition *b* is clocked by *c*; therefore, the multiclocked `if/if-else` **`if`** `(b) q1 [` **`else`** `q2 ]` is legal for incoming clock *c* if, and only if, the following condition is met:

— Every explicit semantic leading clock of `q1 [` **`or`** `q2 ]` 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.

c) Multiclocked **`until`**
Let `c` be the incoming outer clock. Then the clocking of `q1` **`until`** `q2` is equivalent to the clocking of

```
@(c) (q1 until q2)
```

The cycles where the evaluation of `q1` and `q2` should start are being determined by `c`; therefore, the multiclocked `q1` **`until`** `q2` is legal for incoming clock `c` if, and only if, the following condition is met:

— Every explicit semantic leading clock of `q1` or `q2` is identical to `c`.

For example:

```
@(c) (@(c)p1 until @(c) p2)
```

is legal, but

```
@(c) (@(c) p1 until @(c2) p2)
```

is not**.** The rules for multiclocked **`s_until`**, **`until_with`**, and **`s_until_with`** are the same as for multiclocked **`until`**.

d) Multiclocked **`next`**
Let `c` be the incoming outer clock. Then the clocking of **`next`** `q` is equivalent to the clocking of

```
@(c) next q
```

The cycle in which the evaluation of `q` starts is a clock tick of `c`; therefore, the multiclocked **`next`** `q` is legal for incoming clock `c` if, and only if, the following condition is met:

— Every explicit semantic leading clock of `q` is identical to `c`.

The rules for multiclocked **s_next** are the same as for multiclocked **next**.

e) Multiclocked **always**
Let c be the incoming outer clock. Then the clocking of **always** q is equivalent to the clocking of

```
@(c) always q
```

The cycles at which evaluations of q start are clock ticks of c; therefore, the multiclocked **always** q is legal for incoming clock c if, and only if, the following condition is met:

— Every explicit semantic leading clock of q is identical to c.

The rules for multiclocked **s_always** are the same as for multiclocked **always**.

f) Multiclocked **eventually**
Let c be the incoming outer clock. Then the clocking of **eventually** q is equivalent to the clocking of

```
@(c) eventually q
```

The cycle at which evaluation of q starts is a clock tick of c; therefore, the multiclocked **next** q is legal for incoming clock c if, and only if, the following condition is met:

— Every explicit semantic leading clock of q is identical to c.

The rules for multiclocked **s_eventually** are the same as for multiclocked **eventually**.

36.45 Property specification

REPLACE

Details:

1) Variables are declarations of property variables. The value of these variables cannot be accessed.

2) Within the context of a property expr, **vpiOpType** can be any one of **vpiNotOp**, **vpiOverlapImplyOp**, **vpiNonOverlapImplyOp**, **vpiCompAndOp**, **vpiCompOrOp**, **vpiIfOp** or **vpiIfElseOp**. Operands to these operations shall be provided in the same order as shown in the BNF.

WITH

Details:

3) Variables are declarations of property variables. The value of these variables cannot be accessed.

4) Within the context of a property expr, **vpiOpType** can be any one of **vpiNotOp**, **vpiOverlapImplyOp**, **vpiNonOverlapImplyOp**, **vpiCompAndOp**, **vpiCompOrOp**, ~~or~~ **vpiIfOp, vpiIfElseOp, vpiOverlapFollowedByOp, vpiNonOverlapFollowedByOp, vpiNextOp, vpiStrongNextOp, vpiAlwaysOp, vpiStrongAlwaysOp, vpiEventuallyOp, vpiStrongEventuallyOp, vpiUntilOp, vpiStrongUntilOp, vpiUntilWithOp, vpiStrongUntilWithOp** and **vpiStrongOp**. Operands to these operations shall be provided in the same order as shown in the BNF.

A.2.10 Assertion declarations

REPLACE

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

WITH

property_expr ::=

sequence_expr
| **strong** (sequence_expr)
| **weak** (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]
| sequence_expr **#-#** property_expr
| sequence_expr **#=#** property_expr
| **next** property_expr
| **next** [constant_expression] property_expr
| **s_next** property_expr
| **s_next** [constant_expression] property_expr
| **always** property_expr
| **always** [cycle_delay_const_range_expression] property_expr
| **s_always** [constant_range] property_expr
| **eventually** property_expr
| **eventually** [constant_range] property_expr

| **s_eventually**[cycle_delay_const_range_expression] property_expr
| property_expr **until** property_expr
| property_expr **s_until** property_expr
| property_expr **until_with** property_expr
| property_expr **s_until_with** property_expr
| property_expr **implies** property_expr
| property_expr **iff** property_expr
| **accept_on** (expression_or_dist) property_expr Note: from Mantis #1757
| **reject_on** (expression_or_dist) property_expr Note: from Mantis #1757
| property_instance
| clocking_event property_expr

Table B1—Reserved keywords

Note to editor: add the following keywords to Table B1

eventually

implies

next

s_always

s_eventually

s_next

strong

s_until

s_until_with

until

until_with

weak

M.2 Source code

REPLACE

```
#define vpiImplyOp               50 /*  implication operator */
#define vpiNonOverlapImplyOp     51 /* |=> nonoverlapped implication */
#define vpiOverlapImplyOp        52 /* |-> overlapped implication operator */
```

WITH

```
#define vpiImplyOp               50 /* implication operator */
#define vpiNonOverlapImplyOp     51 /* |=> nonoverlapped implication */
#define vpiOverlapImplyOp        52 /* |-> overlapped implication operator */
```

```
#define vpiOverlapImplyOp  editor to fill /* #-# overlapped followed_by operator
*/
#define vpiNonOverlapFollowedByOp       editor to fill /* #=# overlapped
followed_by operator */
#define vpiNextOp                editor to fill /* next operator */
#define vpiStrongNextOp          editor to fill /* s_next operator */
#define vpiAlwaysOp              editor to fill /* always operator */
#define vpiStrongAlwaysOp        editor to fill /* s_always operator */
#define vpiEventuallyOp          editor to fill /* eventually operator */
#define vpStrongiEventuallyOp    editor to fill /* s_eventually operator */
#define vpiUntilOp               editor to fill /* until operator */
#define vpiStrongUntilOp         editor to fill /* s_until operator */
#define vpiUntilWithOp           editor to fill /* until_with operator */
#define vpiStrongUntilWithOp     editor to fill /* s_until_with operator */
#define vpiStrongOp              editor to fill /* strong operator */
```