Additional Temporal Logic Operators

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_declaration ::=
property property_identifier [([tf_port_list])] ;
{ assertion_variable_declaration }
property_spec ;
endproperty [: property_identifier]
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** [const_or_range_expression] property_expr
| **s_next** property_expr

| **s_next** [const_or_range_expression] property_expr
| **always** property_expr
| **always** [range_expression] property_expr
| **s_always** [range_expression] property_expr
| **eventually** 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 ;

property_instance ::=
ps_property_identifier [([property_list_of_arguments])]

property_list_of_arguments ::=
[property_actual_arg] { , [property_actual_arg] } { , . identifier ([property_actual_arg]) }
| . identifier ([property_actual_arg]) { , . identifier ([property_actual_arg]) }

property_actual_arg ::=
property_instance
| sequence_actual_arg

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

Formatted Table

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, boolean operators (negation, disjunction, conjunction, **if...else**, implication and iff) and temporal operators (**next**, **always**, **until**, **until_with**, **eventually**, **accept_on** and **reject_on**) 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~~

Formatted: 2.DRAFT
Formatted: Font color: Blue
Formatted: Keyword, Font color: Blue
Formatted: 2.DRAFT
Formatted: Font color: Blue
Formatted: 2.DRAFT
Formatted: 2.DRAFT
Formatted: Keyword, Font color: Blue
Formatted: 2.DRAFT
Formatted: Keyword, Font color: Blue
Formatted: 2.DRAFT
Formatted: Keyword, Font color: Blue
Formatted: 2.DRAFT, Check spelling and grammar
Formatted: 2.DRAFT
Formatted: Keyword, Font color: Blue
Formatted: 2.DRAFT
Formatted: Keyword, Font color: Blue
Formatted: 2.DRAFT
Formatted: Keyword, Font color: Blue
Formatted: 2.DRAFT
Formatted: 1.DELETE
Formatted: 2.DRAFT
Formatted: 1.DELETE, (Asian) Japanese, (Complex) Arabic Saudi Arabia, Check spelling and grammar

~~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
	`strong, weak`	—
	`not`	—
`and`	**`and`**	Left
`or`	**`or`**	Left
	`if-else`	Right
	`accept_on, reject_on, next, s_next, always, s_always, eventually, until, s_until, until_with, s_until_with`	Left
	`|->, |=>`	Right
	`implies`	Right
	`iff`	Left

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.

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)`. **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, it can be extended to a nonempty match of the sequence. A sequence that admits an empty match is not allowed as a property. In other words it is evaluated false if, and only if there is a finite prefix of the computation, which witness that there is no nonempty match of the sequence.

Since only one match is needed, properties of the form `sequence_expr`, `(`**weak**`(`*`sequence_expr`*`))`, `(`**strong**`(`*`sequence_expr`*`))` are evaluated to true if and only if the properties **first_match**`(`*`sequence_expr`*`),` `(`**weak**`(`**first_match**`(`*`sequence_expr`*`)))` and `(`**strong**`(`**first_match**`(`*`sequence_expr`*`)))` are evaluated to true respectively. 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.

Without the **strong, weak** operators, the evaluation of `sequence_expr` depends on the statements in which it is being used. In case that the statement is **assert property** or **assume property**, it is evaluated as **weak**`(sequence_expr),` otherwise it is beng evaluated as **strong**`(`*`sequence_expr`*`)`.

The following examples illustrate the sequence property forms:

```
property p1;
      strong(##[0:$] a);
endproperty
property p2;
   weak(##[0:$] a);
endproperty
property p3;
   @(posedge clk) a;
endproperty
c1: cover property(p3);
a1: assert property (p3);
```

Property `p1` says that the sequence `##[0:$] a` must match. This sequence matches if, and only if `a` eventually becomes true. Property `p2` is meaningless: it is true for any possible valuations of `a`. Indeed there is no evidence at any time that `##[0:$] a` has no match in the future. Assertion `c1` says that `a` is true when `clk` rises for the first time, and that `clk` must rise at least once. Cover `a1` says that if `clk` rises at least once then `a` must be true when `clk` rises for the first time,

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_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.

Note to editor: Insert the subclause “16.12.2 Implication” 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. When `property_expr1` and `property_expr2` are boolean, then `property_expr1` **`implies`** `property_expr2` is similar to `property_expr1 -> property_expr2`.

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 `property_expr1` evaluates to false and `property_expr2` evaluates to false or `property_expr1` evaluates to true and `property_expr2` evaluates to true. When `property_expr1` and `property_expr2` are boolean, then `property_expr1` **`iff`** `property_expr2` is similar to `property_expr1 <-> property_expr2`.

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*, 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.

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 either the form

```
sequence_expr #-# property_expr
```

in which case it is an overlapping *followed-by* or the form

```
sequence_expr #=# property_expr
```

in which case it is a non-overlapping *followed-by*.

The *followed_by* operator is used to specify a property precondition. The following points should be noted for the overlapping *followed_by* operator:

— 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*.

— The end point of the match of the antecedent sequence_expr is the start point of the evaluation of the consequent *property_expr*.

For overlapped *followed_by*, the end point of the match of *sequence_expr* is the start point of the evaluation of *property_expr*. For non-overlapped *followed_by*, the start point of the evaluation of *property_expr* is the clock tick after the end point of the match.

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 first 6 clock ticks, and starting from one of the clock ticks when `done` was asserted `rst` shall always be low. Property `p2` says that `done` shall be asserted at some clock tick during first 6 cycles, and starting from one of the clock tick following the clock tick when `done` was asserted `rst` shall always be low. Note that the *followed_by* operators are strong. In the examples above, the properties require that the simulation shall not terminate before `done` is true.

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

The *followed_by* operators are especially convenient for specifying cover properties over sequences 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:

- **next** `property_expr` (weak **next** operator)
- **next** `[const_or_range_expression] property_expr` (weak form with range)
- **s_next** `property_expr` (strong **next** operator)
- **s_next** `[const_or_range_expression] property_expr` (strong form with range)

Weak **next** property evaluates to true if `property_expr` holds at the next clock tick or if there are no further clock ticks. To require the clock tick to occur, use the strong form **s_next** `property_expr`. Weak next property with range evaluates to true if `property_expr` holds somewhere within the range of future clock ticks indicated by the constants, or if there are not enough clock ticks for the property to complete the evaluation. To require the necessary clock ticks to occur use the strong form **s_next** `[const_or_range_expression] property_expr`. The `const_or_range_expression` may be unbounded for the strong form, but shall be bounded for the weak form.

Examples.

```
property p1;
   next a;
endproperty
```

```
property p2;
   s_next a;
endproperty
property p3;
   next always a;
endproperty
property p4;
   s_next always a;
endproperty
property p5;
   next eventually a;
endproperty
property p6;
   s_next eventually a;
endproperty
property p7;
   next [2:5] a;
endproperty
property p8;
   s_next [2:5] a;
endproperty
property p9;
   next [2:$] a; // Illegal
endproperty
property p10;
   s_next [2:$] a;
endproperty
```

The property `p1` says that if the property clock ticks once more, then `a` shall be true at the next clock tick. The property `p2` says that the property clock shall tick once more and `a` shall be true at the next clock tick. The property `p3` says that while its clock ticks, `a` shall be true at each future clock tick starting from the next clock tick. Property `p4` says that the property clock shall tick at least one more time and while it ticks, `a` shall be true at each future clock starting from the next clock tick. The property `p5` says that if the property clock ticks at least once, it shall tick enough times for `a` to be true at some point in the future starting from the next clock tick. The property `p6` says that should be true sometime in the strict future.

The properties `p7` and `p8` say that `a` shall be true at some tick between the second and the fifth clock ticks; `p7` does not require the clock to tick, while `p8` does. `p7` is equivalent to **weak**`(##[2:5] a)`, and `p8` is equivalent to **strong**`(##[2:5] a)`. The property `p9` is illegal since an unbounded range cannot be used with the weak **next** form. The property `p10` says that the clock shall tick enough times and a shall happen in the future,

```
property p9;
```

```
    next [2:$] a;  // Illegal
endproperty
property p10;
    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:

- **always** `property_expr`
- **always** `[const_or_range_expression] property_expr` (weak form with range)
- **s_always** `[const_or_range_expression] property_expr` (strong form with range)

Note that there is no strong form of **always** operator without range.

A property **always** `property_expr` evaluates to true if `property_expr` holds at every clock tick. A property **always** `[const_or_range_expression] property_expr` evaluates to true if `property_expr` holds at every clock tick within the range of future clock ticks, or if there are not enough clock ticks for the property to complete the evaluation. To require the necessary clock ticks to occur use the strong form **s_always** `[const_or_range_expression] property_expr`. The `const_or_range_expression` may be unbounded for the weak form, but shall be bounded for the strong form.

There is also the implicit **always** that is associated with the verification statements (see 16.13.4). In that case the verification statement will evaluate the underlying `property_expr` starting at every occurrence of its leading clocking event, unless the verification statement is placed inside an **initial** block. The implicit **always** in the following example will succeed in every attempt, if and only if the explicit always will succeed in its single attempt:

Implicit form:

```
assert property (p);
```

Explicit form:

```
initial 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` shell be 1 for the first 5 clock ticks and then remain 0 for the rest of the computation. The assertion is being evaluated once starting at the first clock tick The property `p1` says that if there is `a` followed by `b` then starting from the next clock tick after b c shall be forever true. The properties `p2` and `p3` say that during the clock ticks 2, 3, 4, and 5 `a` shall be true. The difference between them is that `p3` requires the property clock to tick at least 5 more times. The property `p4` is true if a remains true starting from the second clock tick (the clock is not required to tick). 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:

- `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 until at least one tick before a clock tick where `property_expr2` evaluates to true. The overlapping form evaluates to true if `property_expr1` evaluates to true at every clock tick until and including a clock tick at which `property_expr2` evaluates to true. The strong forms of an **until** property require that `property_expr2` eventually happens in the future (this also include the requirement from the property clock to tick enough times), while the weak forms evaluate to true even if `property_expr2` never happens provided that `property_expr1` is always true at each clock tick.

Examples.

```
property p1;
   a until b;
endproperty
property p2;
   a s_until b;
endproperty
property p1;
```

```
    a until_with b;
endproperty
property p1;
    a s_until_with b;
endproperty
```

The property p1 says that a remains true until (not including) b becomes true. If b never becomes true a shall remain true forever. The property p1 is equivalent to a[*0:$] ##1 b. The property p2 says that a remains true until (not including) b becomes true, and that b shall eventually happen. The property p2 is equivalent to **strong**(a[*0:$] ##1 b). The property p3 says that a remains true until (including) b becomes true. If b never becomes true a shall remain true forever. The property p3 is equivalent to a[*0:$] ##0 b. The property p4 says that a remains true until (including) b becomes true, and that b shall eventually happen. The property p2 is equivalent to **strong**(a[*0:$] ##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 form

```
eventually property_expr
```

The **eventuality** property evaluates to true if property_expr evaluates to true in a finite number of clock ticks, and the clock shall tick enough times for property_expr to happen.

Examples.

```
property p1;
    eventually a;
endproperty
property p2;
    eventually always a;
endproperty
property p3;
    always eventually a;
endproperty
```

The property p1 says that a shall happen some time in the future. It is equivalent to ##[*0:$] a. The property p2 says that starting from some point on a should be always true. The property p3 says that for infinite computations a shall becomes true infinitely many times, and for finite words, a should hold at the last clock tick.

16.12.14 accept_on and reject_on property

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

Insert description from Mantis 1757

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**, **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 at some (finite) time `a` becomes false. To the contrary, a failure of the property `eventually a` cannot be identified in a finite time: if it is violated, the value of `a` must be always false.

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.

16.15.1

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 s_next *q* is {*inherited*}.

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

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**, **vpiIfOp, vpiIfElseOp, vpiOverlapFollowedByOp, vpiNonOverlapFollowedByOp, vpiNextOp, vpiStrongNextOp, vpiAlwaysOp, vpiStrongAlwaysOp, vpiEventuallyOp, 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** [const_or_range_expression] property_expr
| **s_next** property_expr
| **s_next** [const_or_range_expression] property_expr
| **always** property_expr
| **always** [range_expression] property_expr
| **s_always** [range_expression] property_expr
| **eventually** 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

strong

until

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 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 */
```