

## 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_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 [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 ;
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 l-> property_expr`

or the form

`sequence_expr l=> 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
And	not	—
Or	and	Left
	or	Left
	if-else	Right
	->,  =>	Right

A **disable iff** clause can be attached to a *property\_expr* to yield a *property\_spec*. **disable iff** (*expression\_or\_dist*) *property\_expr* The expression of the **disable iff** is called the *reset expression*. The **disable iff** clause allows 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 described in the following subclasses.](#) ~~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_expr1 -> property_expr`

or the form

`sequence_expr1 ==> 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 strong and weak sequences is not defined because these sequences use parenthesis.

Sequence operators	Property operators	Associativity
[*], [=], [->]		—
##		Left
throughout		Right
within		Left
intersect		Left
	Not, next, s_next	—
and	and	Left
or	or	Left
	iff	
	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*). **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 a non-match of the sequence. A sequence that admits an empty match is not allowed as a property.

Without the **strong**, **weak** operators, the evaluation of *sequence\_expr* depends on the verification 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 being evaluated as **strong**(*sequence\_expr*).

Since only one match of a *sequence\_expr* needed for **strong**(*sequence\_expr*) to hold, a property of the form **strong**(*sequence\_expr*) is evaluated to *true* if and only if the property **strong**(**first\_match**(*sequence\_expr*)) is evaluated to *true*.

Note that a property of the form **weak**(*sequence\_expr*) is evaluated to *true* if and only if the property **weak**(**first\_match**(*sequence\_expr*)) is evaluated to *true*. However, the explanation for that is more complex.

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

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 at the last cycle of a simulation the sequential property `a ##1 b` always holds and thus, the assertion a1 always fails. 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 “Implication sub clause” 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 `property_expr1` evaluates to false and `property_expr2` evaluates to false or `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

The **followed\_by** construct specifies that there should be a successful evaluation of a property at the end of a match of a sequential antecedent

---

```
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.

.Note that `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).

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\_expression] property\_expr (indexed form of weak next)
- **s\_next** property\_expr (strong **next**)
- **s\_next** [const\_expression] property\_expr (indexed form of strong next)

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 *const\_expression* evaluates to true if *property\_expr* holds in the specified of future clock ticks indicated by the **const\_expression**, 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\_expression*] *property\_expr*.

Note that **next** *property\_expr* is semantically equivalent to  $1 \mid \Rightarrow$  *property\_expr*, and that **s\_next** *property\_expr* is semantically equivalent to  $1 \# \#$  *property\_expr*.

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 s_eventually a;
endproperty
property p6;
    s_next s_eventually a;
endproperty
```

```

property p7;
    next [2] a;
endproperty
property p8;
    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 the second clock ticks; `p7` does not require the clock to tick, while `p8` does. `p7` is equivalent to `weak` (`##2 a`), and `p8` is equivalent to `strong` (`##2 a`).

### 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** [`cycle_delay_const_range_expression`] `property_expr` (ranged form of always)
- **s\_always** [`constant_range`] `property_expr` (ranged strong form of `s_always`)

Note that the only strong form of **always** operator is with a bounded range.

A property **always** `property_expr` evaluates to true if `property_expr` holds at every clock tick. A property **always** [`cycle_delay_const_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** [`constant_range`] `property_expr`.

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` shall be true 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 includes 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 **weak**(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 **weak**(a[\*1:\$] ##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[\*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 form

```

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)

```

Note that the only weak form of **eventually** operator is with bounded ranged.

The **s\_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. Weak eventually 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\_eventually** [constant\_range] `property_expr`. The `constant_range` may be unbounded for the strong form, but shall be bounded for the weak form.

Examples.

```

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

```

```

property p3;
    always 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 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.

The properties p4 and p5 say that a shall be true at some tick between the second and the fifth clock ticks; p4 does not require the clock to tick, while p5 does. P4 is equivalent to `weak(##[2:5] a)`, and p5 is equivalent to `strong(##[2:5] a)`. The property p6 is illegal since an unbounded range cannot be used with the weak `eventually` form. The property p7 says that the clock shall tick enough times and a shall happen in the future,

#### 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`, `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 at some (finite) time a becomes false. To the contrary, a failure of the property `s_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.

### 16.13.2 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**) 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  $\mid\rightarrow$  and **if/if-else**.

Because  $\mid\rightarrow$  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 temporal operators `next`, `always`, `until`, and `eventually`.

Because `|->` and `##` 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.

The temporal operators `next`, `always`, `until`, and `eventually` require their operands to occur in future cycles. The exact timing where they should hold is determined according to the leading clock of the

**Formatted:** Keyword, Font: (Default) Courier New, Font color: Blue, Complex Script Font: Courier New

**Formatted:** 2.DRAFT, Font: (Default) Courier New, Bold, Complex Script Font: Courier New

**Formatted:** Keyword, Font: (Default) Courier New, Font color: Blue, Complex Script Font: Courier New

**Formatted:** 2.DRAFT, Font: (Default) Courier New, Bold, Complex Script Font: Courier New

**Formatted:** Keyword, Font: (Default) Courier New, Font color: Blue, Complex Script Font: Courier New

**Formatted:** 2.DRAFT, Font: (Default) Courier New, Bold, Complex Script Font: Courier New, Check spelling and grammar

**Formatted:** 2.DRAFT, Font: (Default) Courier New, Bold, Complex Script Font: Courier New

underlying property. 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 `(@(posedge clk1) s1` operand property begins on a different clock from the **clock of the underlying property**.

### 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  $m \mid\rightarrow q$  is equivalent to the clocking of  $@(c) m \mid\rightarrow q$

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  $\mid\rightarrow$ . The multiclocked overlapped implication  $m \mid\rightarrow q$  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  $\mid\rightarrow$ .

For example:

$$@ (c) s \mid\rightarrow p_1 \text{ or } @(c_2) p_2$$

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,

$$@ (c) s \#\#1 (@(c_1) s_1) \mid\rightarrow p$$

is not legal because the set of semantic leading clocks of  $p$  is *{inherited}*, the ending clock of the antecedent is  $c$ , and the clock that flows across  $\mid\rightarrow$  and is inherited by  $p$  is  $c$ .

On the other hand,

$$@ (c) s \mid\rightarrow p_1 \text{ or } @(c) p_2$$

And

$$@ (c) s \#\#1 @(c_1) s_1 \mid\rightarrow p_1 \text{ or } @(c_1) p_2$$

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

$$@ (c) \text{ if } (b) q_1 [ \text{ else } q_2 ]$$

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:

$$@ (c) \text{ if } (b) p_1 \text{ else } @(c) p_2$$

is legal, but

$$@ (c) \text{ if } (b) @(c) (p_1 \text{ and } @(c_2) p_2)$$

is not.

## With

The rules for using multiclocked overlapping implication/followed\_by, **and if/if-else**, and the temporal operators **next**, **always**, **until**, and **eventually** 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  $m \mid\rightarrow q$  ( $m \#-\# q$ ) is equivalent to the clocking of  $@(c) m \mid\rightarrow q$  ( $@(c) m \#-\# q$ ). 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  $\mid\rightarrow$  ( $\#-\#$ ). The multiclocked overlapped implication  $m \mid\rightarrow q$  ( $m \#-\# q$ ) 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  $\mid\rightarrow$  ( $\#-\#$ ).

For example:

$$@ (c) s \mid\rightarrow p1 \text{ or } @(c2) p2$$

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

Also,

$$@ (c) s \# \# 1 (@(c1) s1) \rightsquigarrow \#-\# p$$

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  $\rightsquigarrow \#-\#$  and is inherited by  $p$  is  $c$ .

On the other hand,

$$@ (c) s \mid\rightarrow p1 \text{ or } @(c) p2$$

And

$$@ (c) s \# \# 1 @(c1) s1 \rightsquigarrow \#-\# p1 \text{ or } @(c1) p2$$

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

$$@ (c) \text{ if } (b) q1 [ \text{ else } q2 ]$$

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:

$$@ (c) \text{ if } (b) p1 \text{ else } @(c) p2$$

is legal, but

$@(c) \text{ if } (b) \text{ } @(c) \text{ } (p1 \text{ and } @(c2) \text{ } p2)$

is not.

c) Multiclocked **until**

Let  $c$  be the incoming outer clock. Then the clocking of  $q1 \text{ until } q2$  is equivalent to the clocking of

$@(c) \text{ } (q1 \text{ until } q2)$

The cycles where the evaluation of  $q1$  and  $q2$  should start are being determined by  $c$ ; therefore, the multiclocked  $q1 \text{ until } q2$  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:

$@(c) \text{ } (@(c)p1 \text{ until } @(c) \text{ } p2)$

is legal, but

$@(c) \text{ } (@(c) \text{ } p1 \text{ until } @(c2) \text{ } p2)$

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  $\text{next } q$  is equivalent to the clocking of

$@(c) \text{ next } q$

The cycle where the evaluation of  $q$  should start are being determined by  $c$ ; therefore, the multiclocked  $\text{next } q$  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  $\text{always } q$  is equivalent to the clocking of

$@(c) \text{ always } q$

The cycles where the evaluation of  $q$  should start are being determined by  $c$ ; therefore, the multiclocked  $\text{always } q$  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  $\text{eventually } q$  is equivalent to the clocking of

@ (c) **eventually**  $q$

The cycles where the evaluation of  $q$  should start are being determined by  $c$ ; 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**, **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 l-> property_expr
```

| sequence\_expr **l=>** 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 **l->** property\_expr  
 | sequence\_expr **l=>** 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  
 | **reject\_on** (expression\_or\_dist ) property\_expr  
 | property\_instance  
 | clocking\_event property\_expr

Note: from Mantis #1757

Note: from Mantis #1757

**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 vpiStronglyEventuallyOp 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 */
```