

## Resets (Based on P1800-2008-draft1)

Two new property operators `accept_on` and `reject_on` are introduced.

### Modify Syntax 16-14

```

...
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
    | accept_on (expression_or_dist) property_expr
    | reject_on (expression_or_dist) property_expr
...

```

### 16.11, Table 16-25

#### Replace

Sequence operators	Property operators	Associativity
[*], [=], [->]		—
##		Left
throughout		Right
within		Left
intersect		Left
	<b>not</b>	—
<b>and</b>	<b>and</b>	Left
<b>or</b>	<b>or</b>	Left
	->, <->	Right
	<b>if...else</b>	Right
	->,  =>	Right

#### With

Sequence operators	Property operators	Associativity
[*], [=], [->]		—
##		Left
throughout		Right
within		Left
intersect		Left
	<b>not</b>	—
<b>and</b>	<b>and</b>	Left

<b>or</b>	<b>or</b>	Left
	->, <->	Right
	<b>if...else</b>	Right
	->,  =>	Right
	<b>reject_on, accept_on</b>	—

## pp 346, add after g)

h) Property resets are

```

    accept_on(expression) property_expression
and
    reject_on(expression) property_expression

```

For an evaluation of **accept\_on**(expression) property\_expression, there is an evaluation of the underlying property\_expression. If prior to the completion of that evaluation the reset expression becomes true, then the overall evaluation of the property results in *true*. Otherwise, the overall evaluation of the property is equal to the evaluation of the property\_expression.

For an evaluation of **reject\_on**(expression) property\_expression, there is an evaluation of the underlying property\_expression. If prior to the completion of that evaluation the reset expression becomes true, then the overall evaluation of the property results in *false*. Otherwise, the overall evaluation of the property is equal to the evaluation of the property\_expression.

## Insert 16.12.3

(Note to the editor: shift clause numbering)

The operators **accept\_on** and **reject\_on** are evaluated at the granularity of the simulation time step like **disable iff** but they use the sampled value of their argument (i.e., **accept\_on**(b) and **reject\_on**(b) use the value \$sampled(b)). They represent asynchronous resets.

The semantics of **accept\_on** are similar to **disable iff**, except that it operates at the property level rather than the verification statement level, it uses sampled values and it affects the truth of a property in its scope. The semantics of **reject\_on**(expression) property are the same as **not**(**accept\_on**(expression) **not**(property)).

Any nesting of **accept\_on** and **reject\_on** operators is allowed.

For example, a sequence beginning with **go** being true, followed by two occurrences of **get** being true during which **kill** is false, must be followed by a sequence of two occurrences of **put** (not necessarily consecutive) before **stop** may be asserted.

```

assert property (go ##1 (get && !kill)[*2] |-> reject_on(stop) put[->2]);

```

**reject\_on** ( or **accept\_on**) that appear in nested properties are applied to the inner most property first and the result is propagated up to the next level property.

For example,

```

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

```

If a becomes true before the evaluation of p1 is completed and the second term of the **and** operation completed evaluation, the truth of p1 is ignored in deciding the truth of p. On the other hand, if b becomes true before the evaluation of p2 is completed then p evaluates to false.

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

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

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

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

Nested **reject\_on** and **accept\_on** operators are evaluated in the lexical order (left to right). Therefore, if two nested operators conditions become true in the same time step before the completion of the argument property, then the outermost operator takes precedence. For example,

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

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

Like **disable iff**, **reject\_on** and **accept\_on** expressions can contain the sequence boolean method **triggered** and sampled value functions (see 16.13.6). In the latter case, the clock argument can be explicitly specified or inferred from the clock flow. The expressions must not contain any reference to local variables and the sequence methods ended and matched.

## Insert on pp 355 before "recursive properties can represent complicated requirements..."

The operators **accept\_on** and **reject\_on** may be used inside a recursive property. For example, the following uses of **accept\_on** and **reject\_on** property are legal:

```
property p1(p, bit b, abort);
    accept_on(b) (p and (1'b1 | => reject_on(abort) p1(p, b, abort)));
endproperty
property p2(s, p, bit b, abort);
    s | -> p1(p, b, abort);
endproperty
```

The following use of a recursive property also contains a legal deployment of the operators.

```
property p3(p, bit b, abort);
    (p and (1'b1 | => p4(p, b, abort)));
endproperty

property p4(p, bit b, abort);
    accept_on(b) reject_on(abort) p3(p, b, abort);
endproperty
```

### In A.2.10

## 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
  | ( 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
  | accept_on (expression_or_dist) property_expr
  | reject_on (expression_or_dist) property_expr
```

## Annex B

### add the keywords

[accept\\_on](#)  
[reject\\_on](#)

## Annex F.2.1

### Replace

The abstract grammar for unlocked properties is

```
P ::= R // "sequence" form
  | ( P ) // "parenthesis" form
  | not P // "negation" form
  | ( P or P ) // "or" form
  | ( P and P ) // "and" form
  | ( R |> P ) // "implication" form
  | disable iff ( b ) P // "reset" form
```

## With

The abstract grammar for unlocked properties is

```
P ::= R // "sequence" form
    | ( P ) // "parenthesis" form
    | not P // "negation" form
    | ( P or P ) // "or" form
    | ( P and P ) // "and" form
    | ( R |-> P ) // "implication" form
    | disable iff ( b ) P // "global reset" form
    | accept_on ( b ) P // "property accept" form
    | reject_on ( b ) P // "property reject" form
```

## Replace

The abstract grammar for clocked properties is

```
Q ::= @( b ) P // "clock" form
    | S // "sequence" form
    | ( Q ) // "parenthesis" form
    | not Q // "negation" form
    | ( Q or Q ) // "or" form
    | ( Q and Q ) // "and" form
    | ( S |-> Q ) // "implication" form
    | disable iff ( b ) Q // "reset" form
```

## With

The abstract grammar for clocked properties is

```
Q ::= @( b ) P // "clock" form
    | S // "sequence" form
    | ( Q ) // "parenthesis" form
    | not Q // "negation" form
    | ( Q or Q ) // "or" form
    | ( Q and Q ) // "and" form
    | ( S |-> Q ) // "implication" form
    | disable iff ( b ) Q // "reset" form
    | accept_on ( b ) Q // "property accept" form
    | reject_on ( b ) Q // "property reject" form
```

## Annex F.3.1

### Replace

```
...
— @(c) disable iff ( b ) P  ⇨  disable iff ( b ) @(c) P .
— @(c) not P  ⇨  not @(c) P .
— @(c) ( R |-> P )  ⇨  ( @(c) R |-> @(c) P ) .
— @(c) ( P1 or P2 )  ⇨  ( @(c) P1 or @(c) P2 ) .
```

—  $@(c) ( P_1 \text{ and } P_2 ) \mapsto ( @(c) P_1 \text{ and } @(c) P_2 )$ .

## With

...

—  $@(c) \text{ disable iff } ( b ) P \mapsto \text{disable iff } ( b ) @(c) P$ .

—  $@(c) \text{ accept\_on } ( b ) P \mapsto \text{accept\_on } ( b ) @(c) P$ .

—  $@(c) \text{ reject\_on } ( b ) P \mapsto \text{reject\_on } ( b ) @(c) P$ .

—  $@(c) \text{ not } P \mapsto \text{not } @(c) P$ .

—  $@(c) ( R \mid \rightarrow P ) \mapsto ( @(c) R \mid \rightarrow @(c) P )$ .

—  $@(c) ( P_1 \text{ or } P_2 ) \mapsto ( @(c) P_1 \text{ or } @(c) P_2 )$ .

—  $@(c) ( P_1 \text{ and } P_2 ) \mapsto ( @(c) P_1 \text{ and } @(c) P_2 )$ .

## Annex F.3.3.1

### Replace

...

—  $w \models ( P_1 \text{ and } P_2 )$  iff  $w \models P_1$  and  $w \models P_2$ .

Remark: Because  $w$  is nonempty, it can be proved that  $w \models \text{not } b$  iff  $w \models !b$ .

## With

—  $w \models ( P_1 \text{ and } P_2 )$  iff  $w \models P_1$  and  $w \models P_2$ .

—  $w \models \text{accept\_on}(b) P$  iff either  $w \models P$ , or some letter  $i$  of  $w$  satisfies  $b$  and  $w^{0,i-1} \top^\omega \models P$   
for  $i$  the least index such that  $w^i \models b$ ,  $0 < i < |w|$ .

—  $w \models \text{reject\_on}(b) P$  iff  $w \models P$  and if there is a  $k$ ,  $0 \leq k < |w|$ , such that  $w^k \models b$  then the following must hold  $w^{0,k-1} \perp^\omega \models P$ .

$w^{0,-1}$  denotes the empty word.

A word satisfies property  $\text{accept\_on}(b) \mathbb{P}$  if and only if  $\mathbb{P}$  succeeds or prior to the completion of the evaluation of  $\mathbb{P}$  the expression  $b$  evaluates to true.

The operator  $\text{reject\_on}$  has the dual semantics. A word  $w$  satisfies property  $\text{reject\_on}(b) \mathbb{P}$  if and only if  $w$  satisfies  $\mathbb{P}$  and if  $b$  happens first time at the position  $i$  then  $\mathbb{P}$  must have evaluated to true before that  $b$  is encountered.

Remark: Because  $w$  is nonempty, it can be proved that  $w \models \text{not } b$  iff  $w \models !b$ .

## Annex F.3.6.1

### Replace

...

—  $w, L_0 \models (P_1 \text{ and } P_2)$  iff  $w, L_0 \models P_1$  and  $w, L_0 \models P_2$ .

## With

...

—  $w, L_0 \models (P_1 \text{ and } P_2)$  iff  $w, L_0 \models P_1$  and  $w, L_0 \models P_2$ .

—  $w, L_0 \models \text{reject\_on}(b) P$  iff  $w, L_0 \models P$  and if there is a  $k, 0 \leq k < |w|$ , such that  $w^k, L_0 \models b$  then the following must hold  $w^{0,k-1} \perp^\omega, L_0 \models P$ .

$w^{0,-1}$  denotes the empty word.

—  $w, L_0 \models \text{accept\_on}(b) P$  iff either  $w, L_0 \models P$ , or some letter  $i$  of  $w$  satisfies  $b$  and  $w^{0,i-1}, L_0 \top^\omega \models P$  for  $i$  the least index such that  $w^i \models b, 0 < i < |w|$ .