

Resets (Based on P1800-2008-draft4)

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 l-> property_expr
    | sequence_expr l=> 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
	not	—
and	and	Left
or	or	Left
	->, <->	Right
	if...else	Right
	->, =>	Right

With

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

	<code>reject_on, accept_on</code>	—
	<code>if...else</code>	Right
	<code> ->, =></code>	Right

At 16.12 just before the itemized list.

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.

with

The result of property evaluation is either true or false. There are several kinds of property: sequence, negation, disjunction, conjunction, if-else, implication, reset, and instantiation.

pp 342, add after g)

h) A property is a reset if it is of the following forms :

- `accept_on`(expression_or_dist) property_expr
- `reject_on`(expression_or_dist) property_expr

where the `expression_or_dist` is called the *disable condition*.

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

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

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

Insert 16.12.3 Reset properties

(Note to the editor: shift clause numbering)

The operators `accept_on` and `reject_on` are evaluated at the granularity of the simulation time step like `disable iff` but 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` is similar to `disable iff`, except for the following three differences:

- `accept_on` operates at the property level rather than the verification statement level

— `accept_on` uses sampled values

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, whenever `go` is high, followed by two occurrences of `get` being high, then `stop` cannot be high until after `put` is asserted twice (not necessarily consecutive).

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

When the disable condition occurs at the same time step where the evaluation of the `property_expr` ends, the disable condition takes precedence. In particular, when `reject_on` (or `accept_on`) appears in nested properties, the outermost disable condition takes precedence over inner disable conditions.

For example,

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

If `a` becomes true during the evaluation of `p1` 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 during the evaluation of then `p` evaluates to false.

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

If `a` becomes true during the evaluation of `p1` then `p` evaluates to true. On the other hand, if `b` becomes true during the evaluation of `p2` 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 operator conditions become true in the same time step during the evaluation of the argument property, then the outermost operator takes precedence. For example,

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

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

Like `disable iff`, `reject_on` and `accept_on` expressions may contain sampled value functions (see 16.8.3). The clock argument shall be explicitly specified. The expressions shall not contain any reference to local variables and the sequence methods `ended`, `triggered` and `matched`.

Insert on pp 351 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 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 36.45

Replace

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

with

2) Within the context of a property expr, **vpiOpType** can be any one of **vpiNotOp**, **vpiOverlapImPLYOp**, **vpiNon-OverlapImPLYOp**, **vpiCompAndOp**, **vpiCompOrOp**, **vpiIfOp**, ~~**vpiIfElseOp**~~, **vpiAcceptOp**, or **vpiRejectOp**. Operands to these operations shall be provided in the same order as shown in the BNF.

In 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 vpiAcceptOp editor to fill /* accept_on operator */
#define vpiRejectOp editor to fill /* reject_on operator */

```

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 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
  | ( 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
  | 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 l-> P ) // "implication" 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 l-> P ) // "implication" form
  | accept_on ( b ) P // "reset" form
```

Replace

The abstract grammar for clocked properties is

```
 $Q ::= @(b) P // \text{"clock" form}$   
|  $S // \text{"sequence" form}$   
|  $(Q) // \text{"parenthesis" form}$   
| not  $Q // \text{"negation" form}$   
|  $(Q \text{ or } Q) // \text{"or" form}$   
|  $(Q \text{ and } Q) // \text{"and" form}$ 
```

With

The abstract grammar for clocked properties is

```
 $Q ::= @(b) P // \text{"clock" form}$   
|  $S // \text{"sequence" form}$   
|  $(Q) // \text{"parenthesis" form}$   
| not  $Q // \text{"negation" form}$   
|  $(Q \text{ or } Q) // \text{"or" form}$   
|  $(Q \text{ and } Q) // \text{"and" form}$   
|  $(S \mid\text{->} Q) // \text{"implication" form}$   
| accept_on  $(b) Q // \text{"reset" form}$ 
```

F.2.3.5

Replace

— $(\text{if}(b) P1 \text{ else } P2) \equiv ((b \mid\text{->} P1) \text{ and } (!b \mid\text{->} P2))$

With

— $(\text{if}(b) P1 \text{ else } P2) \equiv ((b \mid\text{->} P1) \text{ and } (!b \mid\text{->} P2))$

— $(\text{reject_on}(b) P) \equiv (\text{not } \text{accept_on}(b) \text{ not } P)$

Annex F.3.1

Replace

```
...  
—  $@(c) \text{ disable iff } (b) P \rightarrow \text{disable iff } (b) @(c) P .$   
—  $@(c) \text{ not } P \rightarrow \text{not } @(c) P .$   
—  $@(c) (R \mid\text{->} P) \rightarrow (@(c) R \mid\text{->} @(c) P) .$   
—  $@(c) (P1 \text{ or } P2) \rightarrow (@(c) P1 \text{ or } @(c) P2) .$   
—  $@(c) (P1 \text{ and } P2) \rightarrow (@(c) P1 \text{ and } @(c) P2) .$ 
```

With

- ...
- $@(c) \text{ disable iff } (b) P \rightarrow \text{ disable iff } (b) @(c) P .$
- $@(c) \text{ accept_on } (b) P \rightarrow \text{ accept_on } (b) @(c) P .$
- $@(c) \text{ not } P \rightarrow \text{ not } @(c) P .$
- $@(c) (R \mid\rightarrow P) \rightarrow (@(c) R \mid\rightarrow @(c) P) .$
- $@(c) (P_1 \text{ or } P_2) \rightarrow (@(c) P_1 \text{ or } @(c) P_2) .$
- $@(c) (P_1 \text{ and } P_2) \rightarrow (@(c) P_1 \text{ and } @(c) P_2) .$

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 for some $0 < i < |w|$, $w^i \models b$, and $w^{0,i-1} \top^\omega \models P$.
- A word satisfies property $\text{accept_on } (b) P$ if and only if P succeeds or if during the evaluation of P the expression b evaluates to *true*.

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 $w, L_0 \models P_2$.
- $w, L_0 \models \text{accept_on } (b) P$ iff either $w, L_0 \models P$, or for some $0 < i < |w|$, $w^i \models b$ and $w^{0,i-1}, L_0 \top^\omega \models P$.