Resets (Based on P1800-2008-draft1)

Two new property operators **`accepton`** and **`rejecton`** are introduced.

Modify Syntax 17-14

...
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
| **accepton (**expression_or_dist **)** property_expr
| **rejecton (**expression_or_dist **)** property_expr
...

17.11, Table 17-2

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

	`->, <->`	Right
	if...**else**	Right
	`|->, |=>`	Right
	`rejecton, accepton`	—

pp 264, add after g)

h) Property resets are

```
accepton(expression) property_expression
```

and

```
rejecton(expression) property_expression
```

when the sampled value of expression becomes true in a time step, then in the case of **`accepton,`** `property_expression` becomes true, while in the case of **`rejecton`**, `property_expression` becomes false in the time step.

The semantics of **`accepton`** are similar to **`disable iff`**, except that it operates at the property level rather than the verification statement level and it uses sampled values.
The semantics of **`rejecton`**`(expression)` property are the same as
`not``(`**`accepton`**`(expression)` **`not`**`(property))`.

Insert 17.11.3 (note to editor: shift clause numbering)

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

The semantics of accepton are similar to disable iff, except that it operates at the property level rather than the verification statement level and it uses sampled values.
The semantics of **`rejecton`**`(expression)` property are the same as
`not``(`**`accepton`**`(expression)` **`not`**`(property))`.

Any nesting of **`accepton`** and **`rejecton`** 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] |-> rejecton(stop) put[->2]);
```

r**ejecton** (or **accepton)** that appear in nested properties are applied to the inner most property first and the the result is propagated up to the next level property. **not** applied to the top property inverts the result of these operators.

For example,

```
property p; (accepton(a) p1) and (rejecton(b) p2); endproperty
```

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

```
property p; (accepton(a) p1) or (rejecton(b) p2); endproperty
```

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

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

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

Nested **rejecton** and **accepton** 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; accepton(a) rejecton(b) p1; endproperty
```

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

Like **disable iff**, **rejecton** and **accepton** expressions can contain the sequence boolean method `triggered` and sampled value functions (see 17.7.3). 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 pp273 before "recursive properties can represent complicated requirements..."

The operators accepton and rejecton may not be used inside a recursive property. For example, the following uses of accepton and rejecton property are illegal:

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

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

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

property p4(s, p, bit b, abort);
       accepton(b) rejecton(abort) p3(p);
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
| **accepton (**expression_or_dist **)** property_expr
| **rejecton (**expression_or_dist **)** property_expr

Annex E.2.1

Replace

The abstract grammar for unclocked 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 unclocked 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
   | accepton ( b ) P // "property accept" form
   | rejecton ( 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
   | accepton ( b ) Q // "property accept" form
   | rejecton ( b ) Q // "property reject" form
```

Annex E.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) (P1 **and** P2) ↦ (@(c) P1 **and** @(c) P2).

With

...

— @(c) **disable iff** (b) P ↦ **disable iff** (b) @(c) P .

— @(*c*) **accepton** (*b*) *P* ↦ **accepton** (*b*) @(*c*) *P* .

— @(*c*) **rejecton** (*b*) *P* ↦ **rejecton** (*b*) @(*c*) *P* .

— @(*c*) **not** *P* ↦ **not** @(*c*) *P*.

— @(*c*) (*R* |-> *P*) ↦ (@(*c*) *R* |-> @(*c*) *P*) .

— @(*c*) (*P*1 **or** *P*2) ↦ (@(*c*) *P*1 **or** @(*c*) *P*2) .

— @(*c*) (*P*1 **and** *P*2) ↦ (@(*c*) *P*1 **and** @(*c*) *P*2) .

Annex E.3.3.1

Replace

...
— $w \models$ (P_1 **and** P_2) iff $w \models P_1$ and $w \models P_2$.

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

With

— $w \models$ ($P1$ **and** $P2$) iff $w \models P1$ and $w \models P2$.

— $w \models$ **rejecton**(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.

— $w \models$ **accepton** (b) P iff some letter of w satisfies b and both $w^{0,i-1} \top^{\omega} \models P$ and
$w^{0,i-1} \perp^{\omega} \not\models P$ for i the least index such that $w^i \models b$, $0 < i < |w|$.

The operator **`rejecton`** has the dual semantics. A word `w` satisfies property **`rejecton`**`(b) P` if and only if `w` satisfies `P` and if `b` happens first time at the position `i` then `w[0:i-1]` concatenated with any infinite word must satisfy the property (in other words the property `P` should be completely satisfied before `b` is encountered).

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

Annex E.3.6.1

Replace

...
— $w, L_0 \models$ (P_1 **and** P_2) iff $w, L_0 \models P_1$ and $w, L_0 \models P_2$.

With

...
— $w, L_0 \models$ (P_1 **and** P_2) iff $w, L_0 \models P_1$ and $w, L_0 \models P_2$.

— $w, L_0 \models$ **rejecton**(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.

— $w, L_0 \vDash$ **accepton** $(b)\ P$ iff some letter of w, L_0 satisfies b and both $w^{0,\,i-1}\ \top^{\omega}, L_0 \vDash P$ and $w^{0,\,i-1}, L_0\ \bot^{\omega} \nvDash P$ for i the least index such that $w^{i}, L_0 \vDash b$, $0 < i < |w|$.