

Objectives

The goal of this proposal is to introduce synchronous aborts `accept_if` and `reject_if`. For example:

```
assert property (@(posedge clk) accept_if(rst) a | => b);
```

When `rst` value is high on rising clock, the evaluation attempt is accepted.

```
assert property (@(posedge clk) reject_if(rst) a | => b);
```

When `rst` value is high on rising clock, the active evaluation attempt is rejected.

The synchronous aborts `accept_if` and `reject_if` are similar to asynchronous aborts `accept_if` and `reject_if`, but they are checked only at the time when the clocking event happens, all other time they are ignored. The asynchronous aborts are preemptive: they are checked even between the clocking events.

This proposal is based on the following proposals:

- 1757 – Property resets: `accepton(b) P`, `rejecton(b) P`
- 1932 – Introduce LTL and other temporal operators

All the changes are relative to 1932.

16.12 Declaring properties

REPLACE in [Syntax 16-14](#)

property_expr ::=

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

WITH

property_expr ::=

```
...  
| accept_on ( expression_or_dist ) property_expr  
| reject_on ( expression_or_dist ) property_expr  
| accept_if ( expression_or_dist ) property_expr  
| reject_if ( expression_or_dist ) property_expr  
...
```

— An `always` procedure or `initial` procedure as a statement, wherever these `procedures may blocks can` appear

REPLACE in [Table 16-25](#)

`always`, `s_always`, `eventually`, `s_eventually`, `if-else`, `accept_on`, `reject_on`

WITH

`always`, `s_always`, `eventually`, `s_eventually`, `if-else`, `accept_on`, `reject_on`,
`accept_if`, `reject_if`

REPLACE

16.12.14 `accept_on` and `reject_on` property

Note to editor: The clause numbering is taken from 1932

A property is an abort if it has either the form:

— `accept_on`(*expression_or_dist*) *property_expr*

or the form:

— `reject_on`(*expression_or_dist*) *property_expr*

where the *expression_or_dist* is called the abort 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 abort 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 abort 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 operators `accept_on` and `reject_on` are evaluated at the granularity of the simulation time step like `disable iff` but their abort condition is evaluated using sampled value as a regular boolean expression in assertions. The operators `accept_on` and `reject_on` 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.
- While a disable condition of a `disable iff` in a *property_spec* may cause an evaluation of the *property_spec* to be disabled, an abort condition of `accept_on` in a *property_expr* may cause the evaluation of the *property_expr* to be true.

The semantics of `reject_on`(*expression_or_dist*) *property_expr* is the same as `not(accept_on(expression_or_dist) not(property_expr))`.

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 abort condition occurs at the same time step where the evaluation of the *property_expr* ends, the abort condition takes precedence.

For example,

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

If *a* becomes true during the evaluation of *p1*, the first term is ignored in deciding the truth of *p*. On the other hand, if *b* becomes true during the evaluation of *p2* 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*, 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 abort 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.

`reject_on` and `accept_on` abort expressions may contain sampled value functions (see 16.8.3). When sampled value functions other than `$sampled` are used in the abort condition, the clock argument shall be explicitly specified. Abort expressions shall not contain any reference to local variables and the sequence methods `ended`, `triggered` and `matched`.

WITH

16.12.14 ~~Accept_on and reject_on property~~

16.12.14 Abort property

A property is an abort if it has ~~either the form~~ one of the forms:

— `accept_on(expression_or_dist) property_expr`

~~or the form:~~

— `reject_on(expression_or_dist) property_expr`

— `accept_if(expression_or_dist) property_expr`

— `reject_if(expression_or_dist) property_expr`

where the *expression_or_dist* is called the abort condition. The properties `accept_on` and `reject_on` are called *asynchronous abort properties*, and the properties `accept_if` and `reject_if` are called *synchronous abort properties*.

For an evaluation of `accept_on(expression_or_dist) property_expr` and of `accept_if(expression_or_dist) property_expr` there is an evaluation of the underlying *property_expr*. If during the evaluation, the abort 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` and of `reject_if(expression_or_dist) property_expr` there is an evaluation of the underlying `property_expr`. If during the evaluation, the abort 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 operators `accept_on` and `reject_on` are evaluated at the granularity of the simulation time step like `disable iff`, but their abort condition is evaluated using sampled value as a regular boolean expression in assertions. The operators `accept_on` and `reject_on` represent asynchronous resets.

The operators `accept_if` and `reject_if` are evaluated at the simulation time step when the clocking event happens, unlike `disable iff`, `accept_on` and `reject_on`. Their abort condition is evaluated using sampled value as for `accept_on` and `reject_on`. The operators `accept_if` and `reject_if` represent synchronous 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.
- While a disable condition of a `disable iff` in a `property_spec` may cause an evaluation of the `property_spec` to be disabled, an abort condition of `accept_on` in a `property_expr` may cause the evaluation of the `property_expr` to be true.

The semantics of `reject_on(expression_or_dist) property_expr` is the same as `not(accept_on(expression_or_dist) not(property_expr))`.

The semantics of `accept_if` is similar to `accept_on`, except for it evaluates only at the time steps when the clocking event happens.

The semantics of `reject_if(expression_or_dist) property_expr` is the same as `not(accept_if(expression_or_dist) not(property_expr))`.

~~Any nesting of `accept_on` and `reject_on` operators is allowed.~~

Any nesting of abort operators `accept_on`, `reject_on`, `accept_if`, and `reject_if` 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]);
```

In this example the `stop` is an asynchronous abort, its value is checked even between ticks of `clk`. The following is the synchronous version of the same example:

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

Here `stop` is checked only at the `clk` ticks. The latter assertion can also be written as:

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

When the abort condition occurs at the same time step where the evaluation of the `property_expr` ends, the abort condition takes precedence. For example:

~~For example,~~

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

If `a` becomes true during the evaluation of `p1`, the first term is ignored in deciding the truth of `p`. On the other hand, if `b` becomes true during the evaluation of `p2` 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`, 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 abort operator. Therefore, if `a` becomes true while evaluating `p1` property `p` evaluates to false.

Nested ~~reject_on and accept_on~~ `accept_on`, `reject_on`, `accept_if` and `reject_if` operators are evaluated from 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.

~~reject_on and accept_on abort expressions~~ The abort conditions may contain sampled value functions (see 16.8.3). When sampled value functions other than `$sampled` are used in the abort condition, the clock argument shall be explicitly specified. Abort expressions conditions shall not contain any reference to local variables and the sequence methods `ended`, `triggered` and `matched`.

16.15.1 Clock resolution in multilocked properties

REPLACE

— 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`.

WITH

— 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`.

— The set of semantic leading clocks of `accept_if(b) q` is the set of semantic leading clocks of `q`.

— The set of semantic leading clocks of `reject_if(b) q` is the set of semantic leading clocks of `q`.

16.12.16 Recursive properties

Note to editor: The clause numbering is taken from 1932

REPLACE

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

```

WITH

The operators `accept_on`, ~~and~~ `reject_on`, `accept_if` and `reject_if` 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

```

16.13.2 Multiclocked properties

REPLACE

The boolean property operators (`not`, `and`, `or`) as well as the abort operators (`accept_on`, `reject_on`) may be used freely to combine singly clocked and multiclocked properties. The meanings of these 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`, `accept_if`, an `reject_if`) may be used freely to combine singly clocked and multiclocked properties. The meanings of these property operators are the usual ones, just as in the case of singly clocked properties. For example:

36.45 Property specification

REPLACE

- 2) 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.

WITH

- 2) Within the context of a property expr, vpiOpType can be any one of vpiNotOp, vpiOverlapImplyOp, vpiNonOverlapImplyOp, vpiCompAndOp, vpiCompOrOp, vpiIfOp, vpiIfElseOp, **vpiAcceptOnOp**, **vpiRejectOnOp**, **vpiAcceptIfOp**, **vpiRejectIfOp**, 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.

Note to the editor: In 1751 instead of `vpiAcceptOnOp` and `vpiRejectOnOp` `vpiAcceptOp` and `vpiRejectOp` are used. Should be `vpiAcceptOnOp` and `vpiRejectOnOp`.

A.2.10 Assertion declarations

REPLACE

property_expr ::=

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

WITH

property_expr ::=

```
...
| accept_on ( expression_or_dist ) property_expr
| reject_on ( expression_or_dist ) property_expr
| accept_if ( expression_or_dist ) property_expr
| reject_if ( expression_or_dist ) property_expr
...
```

Annex B

ADD the keywords

accept_on
reject_on

REPLACE

F.2.3.2.6 Derived reset operator

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

WITH

F.2.3.2.6 Derived reset operator

F.2.3.2.6 Derived abort operators

- $(\text{reject_on}(b) p) \equiv (\text{not}(\text{accept_on}(b) \text{not } p))$.
- $(\text{accept_if}(b) p) \equiv (\text{accept_on}(b) p)$ when the clock context is 1.
- $(\text{reject_if}(b) p) \equiv (\text{not}(\text{accept_if}(b) \text{not } p))$.

F.3.1.2 Rewrite rules for properties

REPLACE

— $\mathcal{T}(\text{accept_on}(b) p, c) = ((\text{accept_on}(b) \mathcal{T}(p, c)))$.

REPLACE

- $\mathcal{T}(\text{accept_on}(b) p, c) = ((\text{accept_on}(b) \mathcal{T}(p, c)))$.
- $\mathcal{T}(\text{accept_if}(b) p, c) = ((\text{accept_on}(b \ \&\& \ c) \mathcal{T}(p, c)))$.

M.2 Source code

REPLACE

```
#define vpiImPLYOp          50 /* implication operator */
#define vpiNonOverlapImPLYOp 51 /* |=> nonoverlapped implication */
#define vpiOverlapImPLYOp  52 /* |-> overlapped implication operator */
#define vpiAcceptOnOp editor to fill /* accept_on operator */
#define vpiRejectOnOp editor to fill /* reject_on operator */
```

WITH

```
#define vpiImPLYOp          50 /* implication operator */
#define vpiNonOverlapImPLYOp 51 /* |=> nonoverlapped implication */
#define vpiOverlapImPLYOp  52 /* |-> overlapped implication operator */
#define vpiAcceptOnOp editor to fill /* accept_on operator */
#define vpiRejectOnOp editor to fill /* reject_on operator */
#define vpiAcceptIfOp editor to fill /* accept_if operator */
#define vpiRejectIfOp editor to fill /* reject_if operator */
```

Note to the editor: In 1751 instead of vpiAcceptOnOp and vpiRejectOnOp vpiAcceptOp and vpiRejectOp are used. Should be vpiAcceptOnOp and vpiRejectOnOp.