Overlapping operators in multiclock environment

Motivation

Currently, the operators `##0`, `|->` and **`if`** … **`else`**, may only be specified when the same clocking event is used on all the operands. This proposal aims to relax this restriction to allow these operators to be used in multiclock environment provided that global clocking is defined.

Changing clock `clk1` to clock `clk2` with zero delay has the following meaning: wait for the tick of `clk1` and then wait for the nearest tick `@clk2`, i.e., if the tick of `clk2` happens at the same time as the tick of `clk1`, there is no waiting after `clk1` tick, if the ticks of `clk1` and `clk2` do not happen simultaneously, the next tick of `clk2` after `clk1` is awaited.

Allowing clock change with zero delay simplifies multiclock property definition in case of general LTL operators (see Mantis 1932). Currently 1932 contains the limitation that the operand properties shall begin on the same clock as the clock of the underlined property. E.g.,

```
@(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 property clock.

The current proposal lifts this limitation.

16.12 Declaring properties

REPLACE

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.

WITH

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.~~

16.13.1 Multiclocked sequences

REPLACE

Multiclocked sequences are built by concatenating singly clocked subsequences using the single-delay concatenation operator `##1`. This operator is nonoverlapping and synchronizes between the clocks of the two sequences. The single delay indicated by `##1` is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins.

For example, consider

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

A match of this sequence starts with a match of `sig0` at **`posedge`** `clk0`. Then `##1` moves the time to the nearest strictly subsequent **`posedge`** `clk1`, and the match of the sequence ends at that point with a match of `sig1`. If `clk0` and `clk1` are not identical, then the clocking event for the sequence changes after `##1`. If `clk0` and `clk1` are identical, then the clocking event does not change after `##1`, and the above sequence is equivalent to the singly clocked sequence

```
@(posedge clk0) sig0 ##1 sig1
```

When concatenating differently clocked sequences, the maximal singly clocked subsequences are required to admit only nonempty matches. Thus, if `s1`, `s2` are sequence expressions with no clocking events, then the
multiclocked sequence

```
@(posedge clk1) s1 ##1 @(posedge clk2) s2
```

is legal only if neither `s1` nor `s2` can match the empty word. The clocking event `@(`**`posedge`** `clk1)` applies throughout the match of `s1`, while the clocking event `@(posedge clk2)` applies throughout the match of `s2`. Because the match of `s1` is nonempty, there is an end point of this match at **`posedge`** `clk1`. The ##1 synchronizes between this end point and the first occurrence of **`posedge`** `clk2` strictly after it. That occurrence of **`posedge`** `clk2` is the start point of the match of `s2`.

The restriction that maximal singly clocked subsequences not match the empty word ensures that any multiclocked sequence has well-defined starting and ending clocking events and well-defined clock changes. If `clk1` and `clk2` are not identical, then the sequence

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1[*0:1]
```

is illegal because of the possibility of an empty match of `sig1[*0:1]`, which would make ambiguous whether the ending clocking event is `@(posedge clk0)` or `@(posedge clk1)`.

Differently clocked or multiclocked sequence operands cannot be combined with any sequence operators other than ##1. For example, if `clk1` and `clk2` are not identical, then the following are illegal:

```
@(posedge clk1) s1 ##0 @(posedge clk2) s2

@(posedge clk1) s1 ##2 @(posedge clk2) s2

@(posedge clk1) s1 intersect @(posedge clk2) s2
```

WITH

Multiclocked sequences are built by concatenating singly clocked subsequences using the single-delay concatenation operator `##1` or the zero-delay concatenation operator `##0`. ~~This operator is nonoverlapping and synchronizes between the clocks of the two sequences.~~ The single delay indicated by `##1` is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins. The zero delay indicated by `##0` is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest possibly overlapping tick of the second clock, where the second sequence begins.

~~For example, consider~~

Example 1.

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

A match of this sequence starts with a match of `sig0` at **`posedge`** `clk0`. Then `##1` moves the time to the nearest strictly subsequent **`posedge`** `clk1`, and the match of the sequence ends at that point with a match of `sig1`. If `clk0` and `clk1` are not identical, then the clocking event for the sequence changes after `##1`. If `clk0` and `clk1` are identical, then the clocking event does not change after `##1`, and the above sequence is equivalent to the singly clocked sequence

```
@(posedge clk0) sig0 ##1 sig1
```

Example 2.

```
@(posedge clk0) sig0 ##0 @(posedge clk1) sig1
```

A match of this sequence starts with a match of `sig0` at **`posedge`** `clk0`. Then `##0` moves the time to the nearest possibly overlapping **`posedge`** `clk1`, and the match of the sequence ends at that point with a match of `sig1`: if **`posedge`** `clk0` and **`posedge`** `clk1` happen simultaneously then the time does not move at `##0`, otherwise, it behaves as `##1`. If `clk0` and `clk1` are not identical, then the clocking event for the sequence changes after `##0`. If `clk0` and `clk1` are identical, then the clocking event does not change after `##0`, and the above sequence is equivalent to the singly clocked sequence

```
@(posedge clk0) sig0 ##0 sig1
```

which is equivalent to

```
@(posedge clk0) sig0 && sig1
```

When concatenating differently clocked sequences, the maximal singly clocked subsequences are required to admit only nonempty matches. Thus, if `s1`, `s2` are sequence expressions with no clocking events, then the
multiclocked sequence

```
@(posedge clk1) s1 ##1 @(posedge clk2) s2
```

is legal only if neither `s1` nor `s2` can match the empty word. The clocking event `@(posedge clk1)` applies throughout the match of `s1`, while the clocking event `@(posedge clk2)` applies throughout the match of `s2`. Because the match of `s1` is nonempty, there is an end point of this match at **`posedge`** `clk1`. The ##1 synchronizes between this end point and the first occurrence of **`posedge`** `clk2` strictly after it. That occurrence of **`posedge`** `clk2` is the start point of the match of `s2`.

The restriction that maximal singly clocked subsequences not match the empty word ensures that any multiclocked sequence has well-defined starting and ending clocking events and well-defined clock changes. If `clk1` and `clk2` are not identical, then the sequence

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1[*0:1]
```

is illegal because of the possibility of an empty match of `sig1[*0:1]`, which would make ambiguous whether the ending clocking event is `@(posedge clk0)` or `@(posedge clk1)`.

Differently clocked or multiclocked sequence operands cannot be combined with any sequence operators other than ##1 and `##0`. For example, if `clk1` and `clk2` are not identical, then the following are illegal:

```
@(posedge clk1) s1 ##0 @(posedge clk2) s2
```

```
@(posedge clk1) s1 ##2 @(posedge clk2) s2
```

```
@(posedge clk1) s1 intersect @(posedge clk2) s2
```

16.13.2 Multiclocked properties

REPLACE

As in the case of singly clocked properties, the result of evaluating a multiclocked property is either true or false. Multiclocked properties can be formed in a number of ways. Multiclocked sequences are themselves multiclocked properties. For example:

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

is a multiclocked property. If a multiclocked sequence is evaluated as a property starting at some point, the evaluation returns true if, and only if, there is a match of the multiclocked sequence beginning at that point.

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:

```
(@(posedge clk0) sig0) and (@(posedge clk1) sig1)
```

is a multiclocked property, but it is not a multiclocked sequence. This property evaluates to true at a point if, and only if, the two sequences

```
@(posedge clk0) sig0
```

and

```
@(posedge clk1) sig1
```

both have matches beginning at the point.

The nonoverlapping implication operator `|=>` can be used freely to create a multiclocked property from an antecedent sequence and a consequent property that are differently clocked or multiclocked. The meaning of multiclocked nonoverlapping implication is similar to that of singly clocked nonoverlapping implication. For example, if `s0` and `s1` are sequences with no clocking event, then in

```
@(posedge clk0) s0 |=> @(posedge clk1) s1
```

`|=>` synchronizes between **posedge** `clk0` and **posedge** `clk1`. Starting at the point at which the implication is being evaluated, for each match of `s0` clocked by `clk0`, time is advanced from the end point of the match to the nearest strictly future occurrence of **posedge** `clk1`, and from that point there must exist a match of `s1` clocked by `clk1`.

The nonoverlapping implication operator `|=>` can synchronize between the ending clock event of its antecedent and several leading clock events for subproperties of its consequent. For example, in

```
@(posedge clk0) s0 |=> (@(posedge clk1) s1) and (@(posedge clk2) s2)
```

`|=>` synchronizes between **`posedge`** `clk0` and both **`posedge`** `clk1` and **`posedge`** `clk2`.

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 `|->` and **`if`**/**`if-else`**.

Because `|->` 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

A clock may be explicitly specified with any property. The property is multiclocked if some of its subproperties have a clock different from the property clock, or some of its subproperties are multiclocked sequences.

As in the case of singly clocked properties, the result of evaluating a multiclocked property is either true or false. ~~Multiclocked properties can be formed in a number of ways.~~ Multiclocked sequences are themselves multiclocked properties. For example:

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

is a multiclocked property. If a multiclocked sequence is evaluated as a property starting at some point, the evaluation returns true if, and only if, there is a match of the multiclocked sequence beginning at that point.

~~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:~~

The following example shows how to form a multiclocked property using boolean property operators:

```
(@(posedge clk0) sig0) and (@(posedge clk1) sig1)
```

This is a multiclocked property, but it is not a multiclocked sequence. This property evaluates to true at a point if, and only if, the two sequences

```
@(posedge clk0) sig0
```

and

```
@(posedge clk1) sig1
```

both have matches beginning at the point.

~~The nonoverlapping implication operator |=> can be used freely to create a multiclocked property from an antecedent sequence and a consequent property that are differently clocked or multiclocked.~~ The meaning of multiclocked nonoverlapping implication is similar to that of singly clocked nonoverlapping implication. For example, if `s0` and `s1` are sequences with no clocking event, then in

```
@(posedge clk0) s0 |=> @(posedge clk1) s1
```

`|=>` synchronizes between **`posedge`** `clk0` and **`posedge`** `clk1`. Starting at the point at which the implication is being evaluated, for each match of `s0` clocked by `clk0`, time is advanced from the end point of the match to the nearest strictly future occurrence of **`posedge`** `clk1`, and from that point there must exist a match of `s1` clocked by `clk1`.

~~The nonoverlapping implication operator |=> can synchronize between the ending clock event of its antecedent and several leading clock events for subproperties of its consequent. For example, in~~

The following example shows a combination of differently clocked properties using both implication and boolean property operators:

```
@(posedge clk0) s0 |=> (@(posedge clk1) s1) and (@(posedge clk2) s2)
```

~~|=> synchronizes between posedge clk0 and both posedge clk1 and posedge clk2.~~

~~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 |> and if/if-else.~~

The multiclocked overlapping implication `|->` has the following meaning: at the end of the antecedent the nearest tick of the consequent clock is awaited. If the consequent clock happens at the end of the antecedent, the consequent is started checking immediately. Otherwise, the meaning of the multiclocked overlapping implication is the same as the meaning of the multiclock nonoverlapping implication.

~~Because |> 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`~~,~~ and `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.~~

means the following: at each match of `s0` the nearest **`posedge`** `clk1` is awaited. If it happens immediately then `s1` is checked without delay, otherwise its check starts at the next **`posedge`** `clk1` as in case with `|=>`. In both cases the evaluation of `s1` is controlled by **`posedge`** `clk1`.

~~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 semantics of multiclocked **`if/if-else`** operators is similar to the semantics of the overlapping implication. For example, if `s1` and `s2` are sequences with no clocking events, then

```
@(posedge clk0) if (b) @(posedge clk1) s1 else @(posedge clk2) s2
```

has the following meaning: the condition `b` is checked at posedge `clk0`. If `b` is true then `s1` is checked at the nearest, possibly overlapping **`posedge`** `clk1`, else `s2` is checked at the nearest non-strictly subsequent **`posedge`** `clk2`.

REPLACE

16.15.1 Clock resolution in multiclocked properties

Due to clock flow, juxtaposition of two clocks nullifies the first. This and the nesting of clocking events within other property building operators mean that there are subtleties in the general interpretation of the restrictions about where the clock can change in multiclocked properties. For example:

```
@(c) s |-> @(c) (p and @(c1) p1)
```

appears legal because the antecedent is clocked by c and the consequent begins syntactically with the clocking event `@(c)`. However, the consequent sequence is equivalent to

```
(@(c) p) and (@(c1) p1)
```

and `|->` cannot synchronize between clock c from the antecedent and clock c_1 from the second conjunct of the consequent. Similarly,

```
@(c) s |-> @(c1) (@(c) p)
```

appears illegal due to the apparent clock change from c to c_1 across `|->`. However, it is legal, although arguably misleading in style, because the consequent property is equivalent to `@(c) p`.

WITH

~~16.15.1 Clock resolution in multiclocked properties~~

16.15.1 Semantic leading clocks for multiclocked sequences and properties

~~Due to clock flow, juxtaposition of two clocks nullifies the first. This and the nesting of clocking events within other property building operators mean that there are subtleties in the general interpretation of the restrictions about where the clock can change in multiclocked properties. For example:~~

~~@(c) s |> @(c) (p and @(c1) p1)~~

~~appears legal because the antecedent is clocked by c and the consequent begins syntactically with the clocking event @(c). However, the consequent sequence is equivalent to~~

~~(@(c) p) and (@(c1) p1)~~

~~and |> cannot synchronize between clock c from the antecedent and clock c1 from the second conjunct of the consequent. Similarly,~~

~~@(c) s |> @(c1) (@(c) p)~~

~~appears illegal due to the apparent clock change from c to c1 across |>. However, it is legal, although arguably misleading in style, because the consequent property is equivalent to @(c) p.~~

REPLACE

~~This subclause gives a more precise treatment of the restrictions on multiclocked use of |> and if/if_else than the intuitive discussion in 16.13. The present treatment depends on the notion of the set of semantic leading clocks for a multiclocked sequence or property.~~

WITH

This subclause defines a notion of the set of semantic leading clocks for a multiclocked sequence or property.

REPLACE

A multiclocked sequence has a unique semantic leading clock, defined inductively as follows:

— The semantic leading clock of *s* is inherited.

— The semantic leading clock of `@(c)` *s* is *c*.

— If inherited is the semantic leading clock of *m*, then the semantic leading clock of `@(c)` *m* is *c*. Otherwise, the semantic leading clock of `@(c)` *m* is equal to the semantic leading clock of *m*.

— The semantic leading clock of (*m*) is equal to the semantic leading clock of *m*.

— The semantic leading clock of m_1 `##1` m_2 is equal to the semantic leading clock of m_1.

WITH

A multiclocked sequence has a unique semantic leading clock, defined inductively as follows:

— The semantic leading clock of *s* is inherited.

— The semantic leading clock of `@(c)` *s* is *c*.

— If ~~inherited~~ *inherited* (Note to the editor: font change) is the semantic leading clock of *m*, then the semantic leading clock of `@(c)` *m* is *c*. Otherwise, the semantic leading clock of `@(c)` *m* is equal to the semantic leading clock of *m*.

— The semantic leading clock of (*m*) is equal to the semantic leading clock of *m*.

— The semantic leading clock of m_1 `##1` m_2 is equal to the semantic leading clock of m_1.

— The semantic leading clock of m_1 `##0` m_2 is equal to the semantic leading clock of m_1.

REPLACE

— If inherited is an element of the set of semantic leading clocks of q, then the set of semantic leading clocks of `@(c)` q is obtained from the set of semantic leading clocks of q by replacing *inherited* by c. Otherwise, the set of semantic leading clocks of `@(c)` q is equal to the set of semantic leading clocks of q.

WITH

— If ~~inherited~~ *inherited* (Note to the editor: font change) is an element of the set of semantic leading clocks of q, then the set of semantic leading clocks of `@(c)` q is obtained from the set of semantic leading clocks of q by replacing *inherited* by c. Otherwise, the set of semantic leading clocks of `@(c)` q is equal to the set of semantic leading clocks of q.

REPLACE

— The set of semantic leading clocks of q_1 and q_2 is the union of the set of semantic leading clocks of q_1 with the set of semantic leading clocks of q_2.

— The set of semantic leading clocks of q_1 or q_2 is the union of the set of semantic leading clocks of q_1 with the set of semantic leading clocks of q_2.

WITH

— The set of semantic leading clocks of q_1 ~~and~~ **and** (Note to the editor: font change) q_2 is the union of the set of semantic leading clocks of q_1 with the set of semantic leading clocks of q_2.

— The set of semantic leading clocks of q_1 ~~or~~ **or** (Note to the editor: font change) q_2 is the union of the set of semantic leading clocks of q_1 with the set of semantic leading clocks of q_2.

DELETE from "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." until the end of the subclause.

INSERT at the end of the subclause.

A multiclocked property has a unique semantic leading clock in case when all its leading clocks are identical. Consider the following example:

```
wire clk1, clk2;
logic a, b;
…
assign clk2 = clk1;
a1: assert property (@(clk1) a and @(clk2) b); // Illegal
a2: assert property (@(clk1) a and @(clk1) b); // OK
always @(posedge clk1) begin
   a3: assert property(a and @(posedge clk2)); //Illegal
   a4: assert property(a and @(posedge clk1)); // OK
end
```

The assertions `a2` and `a4` are legal, while the assertions `a1` and `a3` are not. Though both clocks of `a1` have the same value, they are not identical, therefore `a1` does not have a unique semantic leading clock. The assertions `a3` and `a4` have `@(posedge clk1)` as their inferred clock. This clock is not identical to `@(posedge clk2)` therefore `a3` does not have a unique semantic leading clock.