Motivation

It is often the case that all the concurrent assertions that are placed in a design unit share the same clock and disable iff condition. While it is possible to define a default clocking for the assertions, it is not possible to do so for the disabling condition. It is then necessary to repeat the same code in all the assertions. This proposal remedies the situation by introducing a default disable declaration which applies to all assertions in the scope where the default disable is declared. (Note that it is not applied to properties or sequences, only to assert, assume and cover property statements.) Note that its scoping rules are different from the current default clocking scoping rules in that default disable can be redefined in nested module declarations. However, there is a Mantis ticket #1799 which asks to modify the default clocking scoping rules in the same way.

A byproduct of this proposal is the unification of the terminology: the argument of disable iff is sometimes called *reset expression*, sometimes *reset condition*, and sometimes unwieldy phrases like "expression in the **disable iff** clause". The current proposal suggests using *disable condition* in all cases, since it is more consistent with other assertion terminology, e.g., *disabled execution*, *disabled attempt*, etc.

16.5 Boolean expressions

REPLACE

There are two places where Boolean expressions occur in concurrent properties:

- In the sequences used to build properties
- In the top-level **disable iff** clause (see 16.12)

The expressions used in defining sequences are evaluated over the sampled values of all variables (other than local variables as described in 16.9) and the current values of local variables and the sequence boolean methods `ended` and `matched` (see 16.13.6). The expression in the **disable iff** clause is evaluated using the current values of variables (not sampled) and can contain the sequence boolean method `triggered`. It must not contain any reference to local variables and the sequence methods `ended` and `matched`. If a sampled value function (see 16.8.3) is used in the expression, the sampling clock must be explicitly specified in the actual argument list. For example:

```
assert property ( @(posedge clk)
  disable iff (a && $rose(b, @(posedge clk))) trigger |=> test_expr );
```

The **disable iff** expression will preempt the evaluation of the assertion in a time step where `a` is `1` and the sampled value function returns a `1` as determined by the rules of evaluation for use outside sequences described in 16.8.3.

WITH

There are two places where ~~Boolean~~ boolean expressions occur in concurrent ~~properties~~ assertions:

- In the sequences used to build properties
- In the disable condition inferred for an assertion, specified either in a top-level **disable iff** clause (see 16.12) or in a default disable declaration (see 16.15)

The expressions used in defining sequences are evaluated over the sampled values of all variables (other than local variables as described in 16.9) and the current values of local variables and the sequence boolean methods `ended` and `matched` (see 16.13.6). The ~~expression~~ expressions in ~~the **disable iff** clause is~~ a disable condition are evaluated using the current values of variables (not sampled) and ~~can~~ may contain the sequence boolean method `triggered`. ~~It must~~ They shall not contain any reference to local variables ~~and~~ or

to the sequence methods `ended` ~~and~~ or `matched`. If a sampled value function (see 16.8.3) is used in ~~the expression~~ an expression in a disable condition, the sampling clock ~~must~~ shall be explicitly specified in the actual argument list. For example:

```
assert property ( @(posedge clk)
  disable iff (a && $rose(b, @(posedge clk))) trigger |=> test_expr );
```

The disable condition specified in the **`disable iff`** ~~expression~~ clause will preempt the evaluation of the assertion in a time step where `a` is `1` and the sampled value function returns a `1` as determined by the rules of evaluation for use outside sequences described in 16.8.3.

16.12 Declaring properties

REPLACE

The expression of the **`disable iff`** is called the *reset expression*. The **`disable iff`** clause allows preemptive resets to be specified. For an evaluation of the *property_spec*, there is an evaluation of the underlying *property_expr*. If prior to the completion of that evaluation the reset expression becomes true, then the overall evaluation of the property results in disabled. A property has disabled evaluation if it was preempted due to a **`disable iff`** condition. A disabled evaluation of a property does not result in success or failure. Otherwise, the evaluation of the *property_spec* is the same as that of the *property_expr*. The reset expression is tested independently for different evaluation attempts of the *property_spec*. The values of variables used in the reset expression are those in the current simulation cycle, i.e., not sampled. The expression may contain a reference to an end point of a sequence by using the method *triggered* of that sequence. *Matched* and *ended* of a sequence and local variables cannot be used in the reset expression. If a sampled value function is used in the reset expression, the sampling clock must be explicitly specified in its actual argument list as described in 16.8.3. Nesting of **`disable iff`** clauses, explicitly or through property instantiations, is not allowed.

WITH

The expression of the **`disable iff`** is called the ~~*reset expression*~~ *disable condition*. The **`disable iff`** clause allows preemptive resets to be specified. For an evaluation of the *property_spec*, there is an evaluation of the underlying *property_expr*. If prior to the completion of that evaluation the ~~reset expression~~ disable condition becomes true, then the overall evaluation of the property results in disabled. A property has disabled evaluation if it was preempted due to a **`disable iff`** condition. A disabled evaluation of a property does not result in success or failure. Otherwise, the evaluation of the *property_spec* is the same as that of the *property_expr*. The ~~reset expression~~ disable condition is tested independently for different evaluation attempts of the *property_spec*. The values of variables used in the ~~reset expression~~ disable condition are those in the current simulation cycle, i.e., not sampled. The expression may contain a reference to an end point of a sequence by using the method *triggered* of that sequence. *Matched* and *ended* of a sequence and local variables cannot be used in the ~~reset expression~~ disable condition. If a sampled value function is used in the ~~reset expression~~ disable condition, the sampling clock must be explicitly specified in its actual argument list as described in 16.8.3. Nesting of **`disable iff`** clauses, explicitly or through property instantiations, is not allowed.

16.13.3 Clock flow

REPLACE

The scope of a clocking event does not flow into the reset condition of **`disable iff`**.

WITH

The scope of a clocking event does not flow into the ~~reset~~ disable condition of **`disable iff`**.

16.13.6 Sequence methods

REPLACE

The value of method `ended` evaluates to true if the given sequence has reached its end point at that particular point in time and false otherwise. The ended status of the sequence is set in the Observe region and persists through the Observe region. This method shall only be used to detect the end point of a sequence used in another sequence. It shall be considered an error if this method is used in **`disable iff`** boolean expression for properties. There shall be no circular dependencies between sequences induced by the use of `ended`.

The value of method `triggered` evaluates to true if the given sequence has reached its end point at that particular point in time and false otherwise. The triggered status of the sequence is set in the Observe region and persists through the remainder of the time step. This method shall only be used in **`wait`** statements or boolean expressions (see 9.4.4) outside of sequence context or in the **`disable iff`** boolean expression for properties. It shall be considered an error to invoke this method on sequences that treat their formal arguments as local variables. A sequence treats its formal argument as a local variable if the formal argument is used as an lvalue in operator_assignment or inc_or_dec_expression in sequence_match_item.

WITH

The value of method `ended` evaluates to true if the given sequence has reached its end point at that particular point in time and false otherwise. The ended status of the sequence is set in the Observe region and persists through the Observe region. This method shall only be used to detect the end point of a sequence used in another sequence. It shall be considered an error if this method is used in ~~the **`disable iff`** boolean expression for properties~~ a disable condition. There shall be no circular dependencies between sequences induced by the use of `ended`.

The value of method `triggered` evaluates to true if the given sequence has reached its end point at that particular point in time and false otherwise. The triggered status of the sequence is set in the Observe region and persists through the remainder of the time step. This method shall only be used in **`wait`** statements or boolean expressions (see 9.4.4) outside of sequence context or in disable conditions ~~the **`disable iff`** boolean expression for properties~~. It shall be considered an error to invoke this method on sequences that treat their formal arguments as local variables. A sequence treats its formal argument as a local variable if the formal argument is used as an lvalue in ~~operator_assignment~~ *operator_assignment* or ~~inc_or_dec_expression~~ *inc_or_dec_expression* in ~~sequence_match_item~~ *sequence_match_item*.

16.15 Disable resolution

Note to editor: Shift the numeration of the following subsections accordingly.

Note to the editor: Add a Syntax Box containing the following text:

---

module_or_generate_item_declaration ::= *// from A.1.4*

...

| **default clocking** clocking_identifier **;**

| **default disable** expression_or_dist **;**

A default disable may be declared as an item within a module, interface, or program. It provides a default disable condition to all concurrent assertions in the scope of the default disable declaration. The scope can be the module, interface, or program in which the default disable is declared, but its effect is independent of the position of the declaration within that scope. Declaring more than one default disable item within the same

module, interface, or program shall be an error. Furthermore, the scope also includes any nested module, interface, or program declaration. However, if a nested module, interface, or program declaration itself has a default disable declaration, then that default disable applies within the nested declaration and overrides any default disable from without. The scope does not extend into any instances of modules, interfaces or programs.

In the following example, module `m1` declares `rst1` to be the default disable condition, and there is no default disable declaration in the nested module `m2`. The default disable condition `rst1` applies throughout the declaration of `m1` and the nested declaration of `m2`. Therefore, the inferred disable condition of both assertions `a1` and `a2` is `rst1`.

```
module m1;
   bit clk, rst1;
   default disable rst1;
   a1: assert property (@(posedge clk) p1); // property p1 is defined elsewhere
   ...
   module m2;
      bit rst2;
      ...
      a2: assert property (@(posedge clk) p2); // property p2 is defined
elsewhere
   endmodule
   ...
endmodule
```

If there is a default disable declaration in the nested module `m2`, then within `m2` this default disable condition overrides the default disable condition declared in `m1`. Therefore, in the following example the inferred disable condition of `a1` is `rst1`, but the inferred disable condition of `a2` is `rst2`.

```
module m1;
   bit clk, rst1;
   default disable rst1;
   a1: assert property (@(posedge clk) p1); // property p1 is defined elsewhere
   ...
   module m2;
      bit rst2;
      default disable rst2;
      ...
      a2: assert property (@(posedge clk) p2); // property p2 is defined
elsewhere
   endmodule
   ...
endmodule
```

The following rules apply for resolution of the disable condition:

a) If an assertion has a **`disable iff`** clause, then the disable condition specified in this clause shall be used and any **`default disable`** declaration ignored for this assertion.

b) If an assertion does not contain a **`disable iff`** clause, but the assertion is within the scope of a **`default disable`** declaration, then the disable condition for the assertion is inferred from the **`default disable`** declaration.

c) Otherwise, no inference is performed (this is equivalent to the inference of a `1'b0` disable condition).

Below are two example modules illustrating the application of these rules.

```
module examples_with_default (input logic a, b, clk, rst, rst1);
default disable rst;
```

```
property p1;
   disable iff (rst1) a |=> b;
endproperty

// Disable condition is rst1 - explicitly specified within a1
a1 : assert property (@(posedge clk) disable iff (rst1) a |=> b);

// Disable condition is rst1 - explicitly specified within p1
a2 : assert property (@(posedge clk) p1);

// Disable condition is rst - no explicit specification, inferred from
// default disable statement
a3 : assert property (@(posedge clk) a |=> b);

// Disable condition is 1'b0 . This is the only way to
// cancel the effect of default disable.
a4 : assert property (@(posedge clk) disable iff (1'b0) a |=> b);

endmodule

module examples_without_default (input logic a, b, clk, rst);

property p2;
   disable iff (rst) a |=> b;
endproperty

// Disable condition is rst - explicitly specified within a5
a5 : assert property (@(posedge clk) disable iff (rst) a |=> b);

// Disable condition is rst - explicitly specified within p2
a6 : assert property (@ (posedge clk) p2);

// No Disable condition
a7 : assert property (@ (posedge clk) a |=> b);

// Only enable condition and clocking event are inferred from an always block
// Assertion a8 is equivalent to
// assert property (@(posedge clk) !bit'(rst!='b0) |-> (a |=> b));

always @(posedge clk or posedge rst)
if (rst)
   ...
else begin
   a8 : assert property (a |=> b);
   ...
end

endmodule
```

In assertion `a8` the inferred enabling condition is from the **`else`** clause of the **`if-else`** statement, and thus it has to represent the complementary interpretation of the four-valued expression in the **`if`** condition. One such form is as indicated in the comment above `a8`. Other equivalent forms may be used, such as `((rst != 'b0) !== 1'b1)`.

14-12 Default clocking

Change in Syntax 14-3 from

module_or_generate_item_declaration ::= *// from A.1.4*

...

| **default clocking** clocking_identifier **;**

to

module_or_generate_item_declaration ::= *// from A.1.4*

...

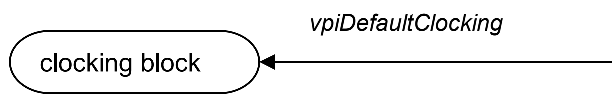
| **default clocking** clocking_identifier **;**

...

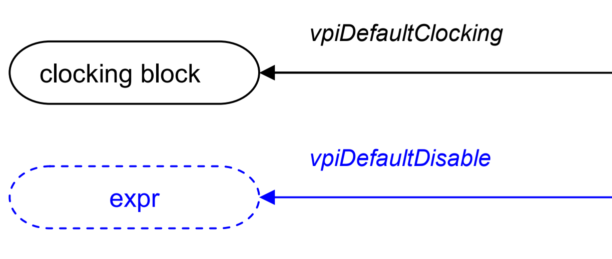
36.4 Module

Note to editor: in the diagram

REPLACE



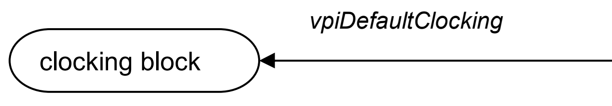
WITH



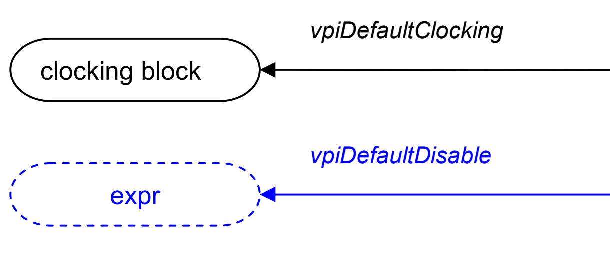
36.5 Interface

Note to editor: in the diagram

REPLACE



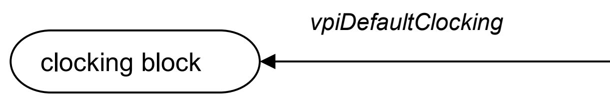
WITH



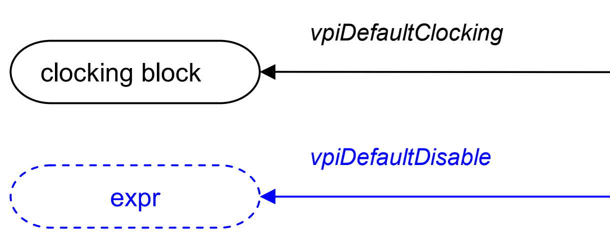
36.8 Program

Note to editor: in the diagram

REPLACE



WITH



A.1.4 Module items

REPLACE

module_or_generate_item_declaration ::=
    package_or_generate_item_declaration
    | genvar_declaration
    | clocking_declaration
    | **default clocking** clocking_identifier **;**

WITH

module_or_generate_item_declaration ::=
    package_or_generate_item_declaration
    | genvar_declaration

| clocking_declaration
| **default clocking** clocking_identifier **;**
| **default disable** expression_or_dist **;**

M.2 Source code

REPLACE

```
#define vpiDefaultClocking 709
```

REPLACE

```
#define vpiDefaultClocking 709
#define vpiDefaultDisable   Editor to fill
```