

This proposal corrects an incomplete fix for item 1381 regarding when an evaluation attempt takes place under inferred enabling condition from an always block.

LJP changes (in purple for review purposes – will be changed to blue for the final review):

1. Changed references to `if..else` to `if/if-else/if-else-if` and `case` to `case/casex/casexz`
2. fixed an error in the very last example where the case statement still had the “1:” instead of “2'b01:”
3. added color coding to indicated deleted text and unchanged text per JH’s feedback
4. changed "The enabling condition assumed from the context" to "The enabling condition inferred from the context" per JH’s feedback
5. changed font of “assert property” and “cover property” in text dashed lists to bold courier per JH’s feedback
6. changed `property_expr` from courier to normal italics per JH’s feedback
7. changed the font of "case" in "Similarly, the enabling condition is also inferred from case statements" should be bold Courier per JH’s feedback
8. Updated proposal to explain inferred enabling conditions for "cover sequence" per JH’s feedback. I think that `##0` should be used for this since the negation dual formulation for "cover property" will not be legal syntax. This will also require some reworking of various places that talk about `property_expr` and `property_spec` to be inclusive of the body of a "cover sequence", which may have a "disable iff" clause, but for which there is not an analogous non-terminal "sequence_spec". And the examples may want to illustrate the inference for "cover sequence".
9. Added a disable iff example per JH’s feedback: The proposal has been crafted carefully so that the transformation inserts the inferred enabling condition after any "disable iff" clause, but this may be a bit too subtle for readers. Consider mentioning this point, perhaps with an example .

P1800-2008-draft3a, 16.14.5, p. 366

Replace

Another inference made from the context is the enabling condition for a property. Such derivation takes place when a property is placed in an **if-else** block or a **case** block. A concurrent assertion embedded in procedural code specifies that a new evaluation attempt of the underlying `property_spec` begins at every occurrence of the inferred clocking event.

```
property r3;
    @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
    if (a) begin
        q <= d1;
        r3_p: assert property (r3);
    end
end
```

The above example is equivalent to the following:

```
property r3;
    @(posedge mclk)a |-> (q != d);
endproperty
r3_p: assert property (r3);
always @(posedge mclk) begin
    if (a) begin
        q <= d1;
    end
end
```

Similarly, the enabling condition is also inferred from **case** statements.

```
property r4;
    @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
    case (a)
        1: begin q <= d1;
            r4_p: assert property (r4);
        end
        default: q1 <= d1;
    endcase
end
```

The above example is equivalent to the following:

```
property r4;
    @(posedge mclk)(a==1) |-> (q != d);
endproperty
r4_p: assert property (r4);
always @(posedge mclk) begin
    case (a)
        1: begin q <= d1;
            end
        default: q1 <= d1;
    endcase
end
```

With

Another inference made from the context is the enabling condition for a **property** verification statement. Such derivation **takes place when a property** shall take place when a verification statement is placed in an **if/if-else/if-else-if** block or a **case/casex/casex** block. The enabling condition inferred from the context is used to modify the *property_expr* or *sequence_expr* that follows the **disable iff**, **if one exists**, of the verification statement. For example:

- If the verification statement is **assume property** or **assert property** then the enabling condition is used as the antecedent of an overlapping implication of which the consequent is the *property_expr* stated in the verification statement.
- If the verification statement is **cover property** then the enabling condition is used as the antecedent of a negated overlapping implication of which the consequent is the negated *property_expr* stated in the verification statement. The two negations used in the property

expression represent the dual property to the overlapping implication and it is often called "followed by".

- If the verification statement is **cover sequence** then the enabling condition is concatenated to the beginning of the *sequence_expr* using the **##0**.

The resulting new *property_expr* or *sequence_expr* then replaces the original *property_expr* or *sequence_expr*, respectively, in the verification statement.

A concurrent assertion embedded in procedural code in an **if/if-else/if-else-if** block or a **case/casex/casez** block specifies that a new evaluation attempt of the underlying *property_spec* or *sequence_expr* begins at every occurrence of the **inferred** (inferred) clocking event.

For example, in a simple case where p1, p2, p3, and p4 are *property_expr* with no disable iff clause, and s1 and s2 are *sequence_expr*:

```
bit a;
always @(posedge mclk) begin
    if (a) begin
        ap: assert property (p1);
        cp: cover property (p2);
        cs: cover sequence (s1);
    end
    else begin
        e_ap: assert property (p3);
        e_cp: cover property (p4);
        e_cs: cover sequence (s2);
    end
end
```

is equivalent to:

```
ap: assert property (@(posedge mclk) a |-> p1);
cp: cover property (@(posedge mclk) not (a |-> not(p2)));
cs: cover sequence(@(posedge mclk) (a ##0 s1));

e_ap: assert property (@(posedge mclk) !a |-> p3);
e_cp: cover property (@(posedge mclk) not (!a |-> not(p4)));
e_cs: cover sequence(@(posedge mclk) (!a ##0 s2));
```

As another example, consider the effect of the **disable iff** clause and of additional code in the conditional statement:

```
sequence s3;
    $rose(ack) ##1 ack;
endsequence
property r3;
    @(posedge mclk) disable iff (reset)((q != d) ##1 ack);
endproperty
always @(posedge mclk) begin
    if (a) begin
        q <= d1;
        ap: assert property (r3);
        cp: cover property (r3);
        cs: cover sequence (disable iff (reset)(s3));
    end
end
```

The above example is equivalent to the following, where the inferred enabling condition is inserted after any **disable iff** expression, and the non-assertion code of the conditional statement persists:

```

sequence s3;
  $rose(ack) ##1 ack;
endsequence
property r3;
  @(posedge mclk) disable iff (reset)((q != d) ##1 ack);
endproperty

r3_p: assert property (r3);
ap: assert property (@(posedge mclk) disable iff (reset)
  a |-> r3);
cp: cover property (@(posedge mclk) disable iff (reset)
  not (a |-> not r3));
cs: cover sequence(@(posedge mclk) disable iff (reset)
  (a ##0 s3));

always @(posedge mclk) begin
  if (a) begin
    q <= d1;
  end
end
end

```

Four-state variables in if/if-else/if-else-if/ or case/casex/casex conditions require special handling. If the bit variables in the previous examples are replaced with a logic variables, the assertions embedded in the else clause require different semantics. For example, for simplicity, assume that r1, r2, r3, and r4 are property_expr (e.g. there is no **disable iff** expression) and that s1 and s2 are sequence_expr, and the code is as follows:

```

logic a;
always @(posedge mclk) begin
  if (a) begin
    ap: assert property (r1);
    cp: cover property (r2);
    cs: cover sequence (s1);
  end
  else begin
    e_ap: assert property (r3);
    e_cp: cover property (r4);
    e_cs: cover sequence (s2);
  end
end
end

```

The equivalent code must account for 4-state semantics of “a”. The above example is equivalent to the following:

```

ap: assert property (@(posedge mclk) a |-> r1);
cp: cover property (@(posedge mclk) not (a |-> not r2));
cs: cover sequence (@(posedge mclk)(a ##0 s1));

e_ap: assert property (@(posedge mclk) !bit'(a!='b0) |-> r3);
e_cp: cover property (@(posedge mclk) not(!bit'(a!='b0)|-> not r4));
e_cs: cover sequence (@(posedge mclk) !bit'(a!='b0) ##0 s2));

```

The enabling condition for assertions in the **else** clause, the expression “!bit'(a!='b0)” properly handles 4-state multi-bit semantics in that ‘b0’ extends the bit length to match the length of “a” and bit'() converts the result to 2-state value.

Similarly, the enabling condition is also inferred from **case**, **casex**, or **casez** statements. For example, assume again that r1, r2, r3, and r4 are property_expr (e.g. there is no **disable iff** expression) and that s1 and s2 are sequence_expr, and the code is as follows:

```

logic [1:0] a;
property r4;
@(posedge mclk)((q != d) ##1 ack);
endproperty
always @(posedge mclk) begin
    case (a)
        1: begin q <= d1;
        2'b01: begin
            q3 <= d1;
            r4_p ap: assert property (r4);
        end
        2'b10: begin
            q2 <= d1;
            cp: cover property (r4);
        end
        default: begin
            q1 <= d1;
            cs: cover sequence (s1);
        end
    endcase
end

```

The above example is equivalent to the following:

```

logic [1:0] a;
property r4;
@(posedge mclk)((q != d) ##1 ack);
endproperty
ap: assert property (@(posedge mclk)(a==2'b01) |-> r4);
cp: cover property (@(posedge mclk) not((a==2'b10) |-> not r4));
cs: cover sequence !(a==2'b01 || a==2'b10) ##0 s1

always @(posedge mclk) begin
    case (a)
        1: begin q <= d1;
        end
        2'b01: q3 <= d1;
        2'b10: q2 <= d1;
        default: q1 <= d1;
    endcase
end

```