Objectives

It is often desired to invoke assertions as close as possible to the code they are checking, to simplify later code maintenance. In the current LRM, this is sometimes not possible due to the restriction that concurrent assertions are generally not permitted in procedural loops. For example:

```
...
always @(posedge clk)
for (i=0; i<MAXI; i=i+1) begin
    for (j=0; j<i; j=j+1) begin
        ...
        table[i][j] <= myfunction(i,j,other_variable);
        // concurrent assert on next line-- illegal in current standard
        assert property ( table[i][j] != `BAD_VAL);
        ...
    end
end
```

Here we have a currently illegal assertion that checks the calculation on the line above. While we could create a separate generate loop and move the property there, this could create a significant distance between the assertion and the code it is checking, hurting readability and maintainability.

This proposal is to enable concurrent property invocations in loops. Once this proposal and the checkers proposal (Mantis 1900) are accepted, a future proposal will also allow checkers in loops
.

16.14.5 Embedding concurrent assertions in procedural code

CHANGE

The enabling condition is inferred from procedural code inside an **always** or **initial** procedure, with the following restrictions:
a) There must not be a preceding statement with a timing control.
b) A preceding statement shall not invoke a task call that contains a timing control on any statement.
c) The concurrent assertion statement shall not be placed in a looping statement, immediately, or in any nested scope of the looping statement.

TO

The enabling condition is inferred from procedural code inside an **always** or **initial** procedure, with the following restrictions:
a) There must not be a preceding statement with a timing control.
b) A preceding statement shall not invoke a task call that contains a timing control on any statement.
~~c) The concurrent assertion statement shall not be placed in a looping statement, immediately, or in any nested scope of the looping statement.~~

Concurrent assertions are permitted within **for** or **foreach** loops. Any looping statement containing a concurrent assertion within its scope, either immediately or within any nested scope, shall be a **for** or **foreach** loop that obeys the following restrictions:

- In a **for** loop, the loop iteration assignment shall modify the iterator by a constant nonzero value, and the loop iterator may not be modified within the body of the loop.

- The outermost **for** loop shall have a constant bound. Any inner **for** loop shall have a bound that is a constant for each iteration of its enclosing loop.
- In a **foreach** loop, the body shall not modify any dimension of any array controlling the loop. All controlling arrays shall be non-associative.
- The body of the loop must consist of a labeled **begin-end** block.
- The loop may not contain an early exit using **break** or similar constructs.

The instance naming of an assertion in a loop shall be the same as in an ordinary labeled statement that would appear at the same line, except that loop iteration indices in square brackets shall be appended to each element of the hierarchical name that specifies a loop, in order to create a separate assertion instance for each loop iteration. In the case of **for** loops, this is basically the same naming method currently used within **generate** blocks

A concurrent assertion in a loop behaves as if it had been created outside the loop, in a **generate** block with the same set of iterations. If the loop is a **foreach** loop, the generate block will use a **for** loop that iterates over the indices used in the **foreach** statement. This may be a nested set of **for** loops if the **foreach** loop traverses a multidimensional structure. The inference of enabling conditions from enclosing **if** or **case** statements still occurs, just as in a normal non-loop concurrent assertion statement. Also, runtime modification of the loop iterator external to the loop shall not affect the set of assertions created.

For example:

```
integer my_ints[2] = '{123, 456};
always @(posedge clk) begin
    foreach (my_ints[i]) begin : b1
        foo[i] <= somefunction(my_ints[i]);
        a1: assume property (foo[i] != `BAD_VAL);
    end
end
```

The assumptions generated by this example are logically equivalent to the assumptions `b1[0].a1` and `b1[1].a1` in the example below:

```
integer my_ints[2] = '{123, 456};
always @(posedge clk) begin
    foreach (my_ints[i]) begin
        foo[i] <= somefunction(my_ints[i]);
    end
end

genvar i1;
generate for (i1=0; i1<=1; i1=i1+1) begin : b1
        a1: assume property (@(posedge clk) (foo[i1] != `BAD_VAL));
    end
endgenerate
```

A more complex example that shows the combination of assertions in loops with enabling conditions follows. Note that in the inner loop, the bound 'i' is not constant, but since it is a constant for each iteration of the enclosing loop, this code fragment is legal.

```
always @(posedge clk) begin
    if (foo) begin
        for (i=0; i<`MAXI; i=i+1) begin : l1
```

```
            if (bar) begin
                for (j=0; j<i; j=j+1) begin : l2
                    table[i][j] <= myfunction(i,j,other_variable);
                    a1: assert property ( table[i][j] != `BAD_VAL);
                end
            end
        end
    end
end
```

This generates assertions equivalent to those in

```
always @(posedge clk) begin
    if (foo) begin
        for (i=0; i<`MAXI; i=i+1) begin
            if (bar) begin
                for (j=0; j<i; j=j+1) begin
                    table[i][j] <= myfunction(i,j,other_variable);
                end
            end
        end
    end
end

genvar i1,j1;
generate
    for (i1=0; i1<`MAXI; i1=i1+1) begin : l1
        for (j1=0; j1<i1; j1=j1+1) begin : l2
            a1: assert property (@(posedge clk)
                  (for -> (bar |-> (table[i1][j1] != `BAD_VAL))));
        end
    end
endgenerate
```

The following examples show illegal uses of assertions in loops:

```
input wire [8:0] loopsize;
for (i=0; i<loopsize; i++) begin : l1
      ...
      // ILLEGAL:  concurrent assertion in loop with runtime bounds
      assert property (foo[i] |-> bar[i]);
end

while (!stopcond) begin : l1
      ...
      // ILLEGAL:  concurrent assertion in while loop
      assert property (foo |-> bar);
end

for (i=0; i<8; i++) begin : l1
      ...
      if (foo[i] != baz[i]) break;
      ...
      // ILLEGAL:  concurrent assertion in loop with break
      assert property (foo[i] |-> bar[i]);
end
```

```
for (i=0; i<8; i++) begin
      ...
      // ILLEGAL:  concurrent assertion in unlabeled loop
      assert property (foo[i] |-> bar[i]);
end
```