

## 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
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 assertion invocations in loops. Once this proposal and the checkers proposal (Mantis 1900) are accepted, a future proposal will also allow checkers in loops.

Note that the existing BNF actually already allows this construct (though the explicit text of clause 16.14.5 prohibits it), so no BNF changes are needed in this proposal.

### 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:

- There must not be a preceding statement with a timing control.
- A preceding statement shall not invoke a task call that contains a timing control on any statement.
- 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:

- There must not be a preceding statement with a timing control.
- A preceding statement shall not invoke a task call that contains a timing control on any statement.
- ~~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 shall not change any other variables or have side effects. 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 and have constant bounds.
- The loop may not contain an early exit using **break** or **continue** constructs.

The instance naming of a concurrent assertion in a procedural loop shall be the same as in an ordinary labeled statement that would appear at the same line. It is still considered a single assertion, not a set of assertions for each iteration, so no loop index is used. Thus the fact that a concurrent assertion is in a procedural loop does not modify hierarchical or VPI references to it.

A concurrent assertion in a procedural loop evaluates its property expression for each possible valid set of loop iterators. The action blocks are then executed with the same set of iterator values that caused the assertion to pass or fail. Thus, unlike a concurrent assertion outside a loop, a combination of its pass and fail action blocks may be executed multiple times (with different iterator values) during a single simulation time step.

For example, in the following code, the expression in assertion a1 will be evaluated separately for  $i=0$  and  $i=1$ :

```
integer [1:0] my_ints = '{123, 456};
always @(posedge clk) begin : b1
  for (i=0; i<=1; i++) begin : b2
    foo[i] = (my_ints[i] == 123);
    a1: assume property (foo[i])
      $display("Good foo vector: %d", foo[i]);
      else $display("Bad foo vector: %d",foo[i]);
  end
end
end
```

For iterator value 0, the assertion will fail, and the fail action block will be executed. For iterator value 1, the assertion will pass, and the pass action block will be generated. Hence the following display will result:

```
Bad foo vector: 456

Good foo vector: 123
```

Enabling conditions are applied to concurrent assertions as described earlier in this clause, even if they are in a procedural loop. 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<3; i=i+1) begin
      if (bar) begin
        for (j=0; j<=i; j=j+1) begin
          a1: assert property ( table[i][j] != `BAD_VAL);
          else report_failure(i,j);
        end
      end
    end
  end
end
```

```

        end
    end
end

```

Following the enable rewrite rules earlier in this clause, the assertion above is logically equivalent to assertion a1 below:

```

always @(posedge clk) begin
    for (i=0; i<3; i=i+1) begin
        for (j=0; j<=i; j=j+1) begin :
            a1: assert property (foo |-> (bar |->
                table[i][j] != `BAD_VAL));
            else report_failure(i, j);
        end
    end
end
end
end

```

The property (foo |-> (bar |-> (table[i][j] != `BAD\_VAL:))) will be evaluated at the posedge of clk for each legal set of (i,j) values: (0,0), (1,0), (1,1), (2,0), (2,1), (2,2). The function report\_failure will be called for each of these value sets for which the property is false.

When using concurrent assertions within loops, users must be careful about what variables they use in the property expressions being checked. Since these are concurrent assertions, all variables other than the loop iterators use the sampled values during the assertion checks. This is very different from immediate assertions, which in general check a possibly transient value that occurred during loop operation. The following example illustrates this issue:

```

bit [4:0] my_bits = '{0, 1, 0, 0};
always @(posedge clk) begin : b1
    for (i=0; i<=4; i++) begin : b2
        ok = (my_bits[i] == 0);
        iterator_copy = i;
        // Concurrent assertions: operate on sampled values for non-
        // iterator variables, so only ac3 actually checks all four
        // possible 'i' values.
        ac1: assert property (ok);
        ac2: assert property (my_bits[iterator_copy] == 0);
        ac3: assert property (my_bits[i] == 0);
        // Immediate assertions: ai1, ai2, and ai3 behave identically
        ai1: assert (ok);
        ai2: assert (my_bits[iterator_copy] == 0);
        ai3: assert (my_bits[i] == 0);
    end
end
end

```

In the example above, ac1 will always be checking the sampled value of variable ok. Since this will be equal to (my\_bits[3] == 0) by the end of any time step, it will always pass, and not be checking each bit as the user probably intended. Similarly, ac2 will always be using the sampled value of iterator\_copy, which will be 3 by the end of the time step, and again is effectively only checking bit 3 of the array. On the other hand, ac3 uses the loop iterator, so its expression will be checked for all four values, and a failure due to the value of my\_bits[2] will be reported. The behavior of these three assertions contrasts with the immediate assertions—since they are directly checked while the procedural code is being executed, ai1, ai2, and ai3 will all report failures during iteration 2.

Thus, to achieve intuitive behavior, concurrent assertions in procedural loops should usually be written so that all variables in their property expression are either loop iterators, indexed by loop iterators, or refer to values that are constant for all iterations of the loop.

The following examples show illegal uses of concurrent assertions in procedural 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
```