

Objectives

Mantis 1900 adds a new language construct called "checkers", to encapsulate sets of assertions and related modeling code. 1995 allows assertions to be placed in procedural loops. Assuming those proposals pass, for consistency we should also allow checkers in procedural loops—hence this proposal.

It is often desired to invoke checkers as close as possible to the code they are checking, to simplify later code maintenance. In the current definition from 1900, this is sometimes not possible. 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);
        // instantiation of a checker-- illegal in current standard
        mycheck c1(table[i][j]);
        ...
    end
end
end
```

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

Note that since the BNF in 1900 allows a checker_instantiation to be the expansion of a procedural_assertion_statement, there is no BNF change required here.

Note to editor: This proposal assumes 1900 and 1995 have already passed.

ADD after clause 16.18.2 from proposal 1900 (Editor: correct numbering here if clauses from 1900 are renumbered)

16.18.2.1 Embedding checkers in procedural loops

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

- In each **for** loop, there shall be a single loop control variable. The loop iteration assignment shall modify the control variable by a constant nonzero value, and this assignment shall not change any other variables or have side effects. The loop control variable 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 fixed-size bounds.

—The loop may not contain an early exit using **break**, **continue**, or **disable** constructs.

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

A checker in a procedural loop evaluates its inputs and executes its assertion statements for each possible valid set of loop control variables. Checker elements other than assertion statements, such as `always_check` blocks and `assign` statements, are not repeated for multiple loop iterations: it is still a single checker, and thus a single instance of its contents. To make this behavior safe, a checker in a procedural loop shall obey the following restriction:

—If a checker input port is connected to or dependent on a loop control variable, that port shall only be used within assertion statement instantiations, and shall not be used otherwise in the body of the checker.

Unlike a checker outside a loop, a combination of pass and fail action blocks for contained assertion statements may be executed multiple times (with different control variable values) during a single simulation time step. For example, in the following code, the expression in assertion `a1` will be evaluated separately for `checkval==0` and `checkval==1`, due to the instantiation within the loop:

```
checker mycheck(integer checkval);
  a1: assume property (checkval != 456)
      $display("Good value: %d", checkval);
      else $display("Bad value: %d", checkval);
endchecker

integer my_ints[1:0] = '{123, 456};
always @(posedge clk) begin : b1
  for (i=0; i<=1; i++) begin : b2
    mycheck c1(my_ints[i]);
  end
end
```

For control variable value 0, assertion `a1` will fail, and the fail action block will be executed. For control variable value 1, assertion `a1` will pass, and the pass action block will be executed. Hence, assuming no code elsewhere modifies `my_ints`, the following display will result:

```
Bad value: 456
Good value: 123
```

When using checkers within loops, users must be careful about what variables they use in the checker arguments. The overall behavior is very different from procedural constructs like immediate assertions, which in general check a possibly transient value that occurred during loop operation. The following example illustrates these issues:

```
checker mycheck2(bit checkval);
  a2: assume property (checkval);
endchecker

bit [3:0] my_bits = '{0, 1, 0, 0};
bit ok = 1'b1;
integer control_variable_copy = 3;
always @(posedge clk) begin
  for (i=0; i<4; i++) begin
    ok = (my_bits[i] == 0);
    control_variable_copy = i;
```

```

    // Checker instances: operate on sampled values for non-
    // control variables, so only ci3 actually checks all four
    // possible 'i' values.
    mycheck2 ci1(ok);
    mycheck2 ci2(my_bits[control_variable_copy] == 0);
    mycheck2 ci3(my_bits[i] == 0);
    // Immediate assertions: ai1, ai2, and ai3 behave identically
    ai1: assert (ok);
    ai2: assert (my_bits[control_variable_copy] == 0);
    ai3: assert (my_bits[i] == 0);
end
end

```

In the example above, `ci1` 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 (assuming no other code modifies any of the variables assigned in this example), it will always pass, and not be checking each bit as the user probably intended. Similarly, `ci2` will always be using the sampled value of `control_variable_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, `ci3` uses the loop control variable, 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 checkers 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, checkers in procedural loops should usually be written so that all variables passed in are either loop control variables, indexed by loop control variables, or refer to values that are guaranteed not to change (for the remainder of the current time step) after the loop is entered.

The following examples show illegal uses of checkers in procedural loops:

```

checker mycheck3(bit checkval);
    a3: assume property (checkval);
endchecker

input wire [8:0] loopsize;
...
for (i=0; i<loopsize; i++) begin : l1
    ...
    // ILLEGAL: checker in loop with runtime bounds
    mycheck3 cbad1 (foo[i] & bar[i]);
end

while (!stopcond) begin : l1
    ...
    // ILLEGAL: checker in while loop
    mycheck3 cbad2 (foo & bar);
end

for (i=0; i<8; i++) begin : l1
    ...
    if (foo[i] != bar[i]) break;
    ...
    // ILLEGAL: checker in loop with break
    mycheck3 cbad3 (foo[i] & bar[i]);
end

```

In the following example, the checker itself is legal, but the usage within a loop is illegal, since an input port affected by a loop iterator is used in procedural code:

```

checker mycheck4(input bit checkval);
    checkvar v1;
    always_check begin

```

```

        // General use of input OK in checker, IF no loop control dependency
        v1 <= checkval;
    end
    ...
    a3: assume property (checkval);
endchecker

for (i=0; i<4; i++) begin : l1
    ...
    // ILLEGAL: miscellaneous use of input depending on loop control variable
    mycheck4 cbad1 (foo[i] & bar[i]);
end

```

16.18.2.1.1 Checker items that are constant in procedural loops

All items other than assertion statements that appear in a checker shall behave identically for every loop iteration, as they are not evaluated multiple times for the multiple iterations. This behavior is enabled by the restriction in 16.18.2.1 (editor correct reference), requiring that any inputs whose values are not identical for all loop iterations be used only in assertion statements within the checker.

The following example shows a checker that verifies some global model properties, with the use of modelling code and some per-iteration properties.

```

checker mycheck2(bit clk1, integer checkval, globalbus);
    checkvar bit bus_ok = 1'b1;
    ...
    // Since globalbus is used in modelling code, it must have the same
    // value for all loop iterations
    always_check @(clk1) begin
        bus_ok <= myfunction(globalbus);
    end
    a1: assert property (bus_ok != 1'b0);

    // Since checkval is used only in an assertion statement, the
    // input may depend on a loop iterator
    a2: assert property (checkval != 456);
endchecker
    ...
    integer my_ints[1:0] = '{123, 456};
    integer my_bus;
    always @(posedge clk) begin
        for (i=0; i<=1; i++) begin
            // LEGAL use: the 'checkval' input depends on loop control variable
            mycheck2 c1(clk1, my_ints[i], my_bus);
            // ILLEGAL use: the 'globalbus' input depends on loop control variable
            mycheck2 c1(clk1, my_ints[i], my_ints[i]);
        end
    end
end

```

MODIFY Clause 16.18.3 from proposal 1900

CHANGE

Checker actual arguments cannot reference automatic variables.

TO

Checker actual arguments cannot reference automatic variables, [except in the special case of loop control variables as described in 16.18.2.1. \(editor correct reference\)](#)