

Problem: Assertions are not allowed in functions.

Context:

Logic designers typically use functions to describe logic that will be replicated but is smaller than logic placed into a module.

Description:

Allowing assertions in a function scope provides for protection for proper use of functions and correctness of their results. Without the ability, it is optional for a designer to write their own assertions. This allows for the possibility to misuse the function or to overlook an erroneous result.

Solution: Support assertions in functions following the same rules for extracting the enabling condition and clock for assertions (17.12.2)

17.12.3 Embedding concurrent assertions in functions.

A concurrent assertion can be embedded within functions. For example:

```
function [1:0] encode4;  
  input [3:0] decoded;  
  begin  
    assert property @(posedge clock)  
      ($onehot(decoded)  
        else $error("Input is not fully decoded, %0b.", decoded)); //fix syntax sampled.  
    casez (1'b1) //synopsys full_case parallel_case  
      decoded[0]: encode4 = 2'd0;  
      decoded[1]: encode4 = 2'd1;  
      decoded[2]: encode4 = 2'd2;  
      decoded[3]: encode4 = 2'd3;  
    endcase  
  end  
endfunction
```

```
...  
way = encode4(hit[3:0]); // Fixme - hit could be zero, violating the property.
```

The code does not allow for any other input value than 1, 2, 4, 8. Thus the property will detect an illegal value and report a failure.

Concurrent assertions shall not be placed in automatic functions or constant functions. Functions can be called from multiple places within a module, allowing reuse of the code. Functions that are called from multiple places will have the following operations done:

- 1) The internal function state is maintained for each call - so that extracted assertions that use internal function state will obtain the correct values.
- 2) The assertion has its enabling condition extracted from the logic to the function call point combined (and) with the enabling condition extracted from the function code.

For example.

```

always @(*)
  begin
    if (selected)
      begin
        selmask = mask_from_valid(valid, flush); // call #1.
        if (nextvalid == FULL && selectTwo)
          secondMask = mask_from_valid(validB, flush2); // call #2.
        ...
      end
  end

function [3:0] mask_from_valid;
  input [3:0] valid;
  input flush;
  reg [4:0] incr;
  begin
    if (!flush)
      begin
        incr = {valid[3:0], 1'b0};
        assert property @(posedge clock) // clock is in upper scope.
          ($onehot(incr))
        else $error("Incremented valid was not onehot, %0d.", incr);
        casez (incr) //synopsys full_case parallel_case
          5'b??010: mask_from_valid = 4'b0011;
          5'b?0110: mask_from_valid = 4'b0111;
          5'b01110: mask_from_valid = 4'b1111;
        endcase
      end
    else
      mask_from_valid = 4'b0;
  end
endfunction

```

The code fragment + function definition will produce assertions equivalent to the following code:

```

reg [4:0] incr_call1; // Extracted state assignments from first call.
reg flush_call1;
always @(*)
  begin : call_1_mask_from_valid
    reg [3:0] _valid;
    _valid = valid; // Assign input reg from call.
    flush_call1 = flush;

    // Compute state for assertion.
    incr_call1 = {_valid[3:0], 1'b0};
  end

reg [4:0] incr_call2; // Extracted state assignments from second call.
reg flush_call2;
always @(*)
  begin : call_2_mask_from_valid
    reg [3:0] _valid;
    _valid = validB; // Assign input reg from call.
    flush_call2 = flush2;

    // Compute state for assertion.
    incr_call2 = {_valid[3:0], 1'b0};
  end
end

```

```
always @(posedge clock)
begin
  assert property          // Assertion extracted from call #1.
    (selected              |-> // Enabling condition for first call.
     !flush_call1         |-> // Enabling condition from function for assertion.
     $onehot(incr_call1))
    else $error("Incremented valid was not onehot, %0d.", incr_call1);

  assert property          // Assertion extracted from call #2.
    (selected
     && (nextvalid == FULL && selectTwo) |-> //enabling condition for 2nd call
     !flush_call2              |-> //enabling condition from function
     $onehot(incr_call2))
    else $error("Incremented valid was not onehot, %0d.", incr_call2);
end
```

The function bodies are inlined for each call to preserve state needed by the assertion.