Motivation

Immediate assertions may report false failures when 0-width glitches occur in assignments to variables in always blocks or due to races in the design (see Mantis #1833). For example, the following implementation of a MUX gate will execute the **always** block twice when **a** changes. The assertion will incorrectly fire on the first execution of the always block:

```
assign not_a = !a;
always_comb begin
  out = a && x || not_a && y;
  assert (out == a ? x : y);
end
```

Concurrent assertions do not suffer from this problem, but there are many situations where a concurrent assertion cannot be used in place of an immediate assertion:

- Concurrent assertions cannot be used inside functions.
- Immediate assertions can be placed in procedural code, but not in structural scopes, so the same combinational checker cannot be used in both contexts.
- Concurrent assertions in always blocks cannot report on intermediate values of variables when assigned more than once in sequential code in an always block.
- Concurrent assertions cannot be used in procedural loops and report on values computed in each iteration (this is covered by Mantis 1995.)

The proposal is to allow concurrent assertions to be placed in more situations:

- in clocked or combinational always blocks, clocks are either explicitly specified or inferred,
- act on values of intermediate assignments to variables using "shadow" variables.
- instantiated in automatic pure functions with some restrictions on the form of the property

The proposal enhances the capability of concurrent assertions in procedural code, but introduces one minor incompatibility with Std 1800-2005, as illustrated in the following example:

```
reg a = 1'b0;
always @(posedge clk) begin
  a = 1'b0;
  a1: assert property(a == 1'b0);
  a = 1'b1;
  a2: assert property(a == 1'b1);
end
```

In P1800-2005, both assertions would be extracted out into the structural context with `@(posedge clk)` as the clocking event, both sampling the variable `a`. Consequently, since the last value assigned to a in any clock cycles is 1'b1, assertion `a1` would always fail. With the new formulation, the assertions would still be extracted out and the clocking event inferred, however, they would sample shadow variables as shown on the following equivalent code:

```
reg a = 1'b0;
reg shadow_a_a1;
reg shadow_a_a2;
always @(posedge clk) begin
  a = 1'b0;
  shadow_a_a1 = a;
  a1: assert property(shadow_a_a1 == 1'b0);
  a = 1'b1;
  shadow_a_a2 = a;
  a2: assert property(shadow_a_a2 == 1'b1);
end
```

Notice that after the first clock tick, both assertions will succeed on all subsequent clock ticks, because they now monitor the value of the variable a at the location where it is assigned. In this way, concurrent assertions can be used like immediate assertions, but be insensitive to glitches and races.

Modify clause 16.14.5 as follows:

16.14.5 Embedding concurrent assertions in procedural code

A concurrent assertion statement can also be embedded in a procedural block. For example:

```
property rule;
  a ##1 b ##1 c;
endproperty
always @(posedge clk) begin
  <statements>
  assert property (rule);
end
```

If the statement appears in an **always** block, the property is always monitored. If the statement appears in an **initial** block, then the monitoring is performed only on the first clock tick.

Two inferences are made from the procedural context: the clock from the event control of an **always** block and the enabling conditions.

A clock is inferred if the statement is placed in an **always** or **initial** block with an event control abiding by the following rules:

— The clock to be inferred must be placed as the first term of the event control as an edge specifier (**posedge** *expression* or **negedge** *expression*).

— The variables in expression must not be used anywhere in the always or initial block.

For example:

```
property r1;
  q != d;
endproperty
always @(posedge mclk) begin
  q <= d1;
  r1_p: assert property (r1);
end
```

The above property can be checked by writing statement r1_p outside the always block and declaring the property with the clock as follows:

```
property r1;
  @(posedge mclk)q != d;
endproperty
always @(posedge mclk) begin
  q <= d1;
end
r1_p: assert property (r1);
```

If the clock is explicitly specified with a property, ~~then it must be identical to the inferred clock, as shown below:~~ it can be different from the inferred clock as shown below:

```
property r2;
  @(posedge ~~mclk~~clk)(q != d);
endproperty
always @(posedge mclk) begin
  q <= d1;
  r2_p: assert property (r2);
end
```

In the above example, ~~(posedge mclk)~~ (**posedge** clk) is the clock for property r2 even though the event control expression is (**posedge** mclk).

If the assertion is placed in an **always** block which does not respect the above restrictions on the event control of the **always** block, then the assertion must have the clock explicitly specified or the **always** block must be in the scope of a **default clocking** declaration in which case the clock of the assertion is inferred.

If a variable assigned in an always block is assigned more than once or is assigned only on some conditional branches, then the location of the assertion in the **always** block that observes that variable is important. The assertion observes the sampled value of the variable that it would have at the location of the assertion. Consider the following example:

```
reg [1:0] a = 2'b00;
always @(posedge clk) begin
   a = 2'b00;
   a1: assert property(a == 2'b00);
   a = 2'b11;
   a2: assert property(a == 2'b11);
end
```

After the first clock tick, both assertions always succeed because they observe the value of a which it carries at the location of the assertions, namely, 2'b00 at the location of a1 and 2'b11 at the location of a2. This kind of position-dependent sampling can be formulated by introducing *shadow* variables of the boolean expressions that appear in the assertion (a in this case) . These variables are assigned the value of the boolean expression at the location of the assertion. In the above example, the equivalent code after extracting assertions from the **always** block and inferring the clocks are as follows:

```
reg a = 1'b00;
bit shadow_a1;
bit shadow_a2;
```

```
always @(posedge clk) begin
   a = 2'b00;
   shadow_a1 = ((a == 2'b00)  != 'b0);
   a = 2'b11;
   shadow_a2 = ((a == 2'b11)  != 'b0);
end
   a1: assert property(@(posedge clk) shadow_a1);
   a2: assert property(@(posedge clk) shadow_a2);
```

A *Shadow variable* is thus defined as follows:

- There is a set of shadow variables for each assertion in the procedural code.
- There are as many shadow variables as there are Boolean expressions in the assertion.
- The type of each shadow variable is **bit.**
- It is assigned using a blocking assignment in place of the assert property statement in the procedural code.
- The right-hand side of the assignment to the shadow variable is the boolean expression it represents compared (!=) with 'b0.

A particular implementation of assertions in procedural code need not use shadow variables, but the observable behavior must the same as if the shadow variables were used.

Functions are commonly used as part of the expressions on the right-hand side of assignments in **always** or **initial** blocks. Concurrent assertions can be placed in functions, and similarly as in **always** blocks the values the assertions observe depend on the location of the function in the function. In contrast to **always** blocks, functions can be called in many places and in that sense they appear as distinct instances of the functions. For concurrent assertions to be legal in a function, the function must satisfy the following criteria:

- The function must be **automatic** and contain no **static** variables.
- The function must be pure and may not access any global variables.
- The variables that appear in assertions must respect the restriction on their types as stated in 16.5.1..
- The function may be called only in modules or interfaces (directly or indirectly through nested function calls), within `always` and `initial` blocks or on the right-hand side of continuous assignments.

The assertions are allowed to contain

- clocking event
- **disable iff** (expression)
- no temporal operators, only a single boolean expression (a combinational assertion)

The assertion must have explicitly specified clock(s) or the function declaration must be in the scope of a **default clocking** declaration in which case the assertion clock is inferred.

Each specific call of the function requires that a new copy of the function be created and declared together with the required shadow variables in the scope where the **always** or **initial** block, or continuous assignment is located. These variables are assigned in the function at the location of the assertions using side-effect assignments.

When a new copy of the function is created, the assertion names within the copy of the function will be renamed by prefixing the existing label by <*function_name*><*index*>., where
function_name is the name of the function that was replicated, and
index is the index number of the copy of the function.
When reporting assertion successes and failures, this extended name becomes part of the reported path name.

This is illustrated in the following example:

```
module m();
  bit clk, reset, z;
  bit [3:0] x, y;

  default clocking ck @(posedge clk);
  endclocking

  // return 1'b1 iff  x has a 1 in the position of every 1 in mask
  function automatic bit test(bit [3:0] a, mask);
    bit tmp;
    my_check: assert property ($onehot0(a));
    tmp = &(a | ~mask);
  endfunction
  ...
  always_comb
    z = test(x, 4'b0100);

  always @(posedge clk)
    y = z ? x : ~'b0;

endmodule
```

The equivalent form of the code becomes

```
module m();
  bit clk, reset, z;
  bit [3:0] x, y;

  bit test1_shadow_a; // shadow variable assigned in copy 1 of the function

  default clocking ck @(posedge clk);
  endclocking

  // return 1'b1 iff  x has a 1 in the position of every 1 in mask
  function automatic bit test_1(bit [3:0] a, mask);
    bit tmp;
    test1_shadow_a = a;
    tmp = &(a | ~mask);
  endfunction

  begin : test1 // generated block
    my_check: assert property ($onehot0(test1_shadow_a));
  end
 ...
  always_comb
    z = test(x, 4'b0100);

 always @(posedge clk)
    y = z ? x : ~'b0;
```

```
endmodule
```

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. Shadow variables must be introduced for the enabling condition as stated above.

```
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 functionally equivalent to the following:

```
bit shadow_a;
bit shadow_q_ne_d;
property r3;
   @(posedge mclk) a |-> (q != d) shadow_a |-> shadow_q_ne_d;
endproperty
r3_p: assert property (r3);
always @(posedge mclk) begin
   if (a) begin
      q <= d1;
      shadow_a = (a != 'b0);
      shadow_q_ne_d = ((q != d) != 'b0;
   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:

```
bit shadow_a_eq_1;
bit shadow_q_ne_d;
```

```
property r4;
   @(posedge mclk)(a==1) |-> (q != d) shadow_a_eq_1 |-> shadow_q_ne_d);
endproperty
r4_p: assert property (r4);
always @(posedge mclk) begin
   case (a)
      1: begin q <= d1;
                  shadow_q_eq_1 = (a != 'b0);
                  shadow_q_ne_d = ((q != d) != 'b0);
         end
      default: q1 <= d1;
   endcase
end
```

The enabling condition is inferred from procedural code inside an **`always`** or **`initial`** block.~~, with the following restrictions:~~ When assertions are embedded in functions, then the enabling condition inference must take into consideration the inferred condition from the initial or always block, as well as from any conditional statements preceding the assertion in the function body. In an **`always`** or **`initial`** block, a concurrent assertion shall obey 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 nested scope of the looping statement.~~

Note to editor: Insert additional text from Mantis 1995 (concurrent assertions in loops) here.