

Embedding Concurrent Assertions in Procedural Code

Introduction

Embedding concurrent assertions in procedural code is a controversial subject, since concurrent assertions are not sequential by their nature and do not always follow properly the execution flow. This document describes the essence of the problem, and suggests a roadmap for short-term and long-term solutions. This is not a detailed proposal.

Current rules. Summary

The following rules informally describe embedded concurrent assertion behavior to help understand the rest of the document. Some subtle rules are omitted.

- Embedded concurrent assertion belongs to the procedural code only syntactically. All variables it references are resolved in the scope to which the assertion belongs, but otherwise embedded assertion is equivalent to a standalone assertion written outside the procedural code with appropriately inferred clock, enable and reset data.
- If a concurrent assertion is embedded into an initial procedure, the assertion monitoring is performed only on the first clock tick. Otherwise, the assertion is continuously monitored. E.g.,

```
initial p1: assert property (@(posedge clk) a);
p2: assert property (@(posedge clk) b);
always p3: assert property (@(posedge clk) c);
```

The value of `a` is checked in the assertion `p1` only on the first rising edge of `clk`, while the values of `b` and `c` in assertions `p2` and `p3` are checked on every rising edge of `clk`.

- If a concurrent assertion with no explicitly specified leading clock, is embedded into an `initial`, `always` or `always_ff` procedure with the first control of the form `posedge clk` or `negedge clk` optionally followed by `iff en`, then the assertion leading clock is inferred as `posedge/negedge clk [iff en]`.
- If a concurrent assertion is in the scope of a conditional statement then the condition is inferred as an enabling condition for the assertion. E.g.,

```
if (cond) assert property (a);
```

is equivalent to

```
assert property (cond |-> a);
```

- (Recent addition). If a concurrent assertion is written in an unrollable loop, it is checked several times during the same simulation step. The loop variable is considered each time as a constant. In the following example

```
always @(posedge clk) begin
    for (int i = 0; i < 3; i++) begin
        p: assert property (a[i]);
        ...
    end
end
```

at each rising clock edge the assertion `p` checks `a[0]`, `a[1]`, and `a[2]`.

Problem statement

Hysteresis

Concurrent assertions use sampled values, therefore assertion failure reporting is shifted by one clock towards the past:

```
always @(posedge clk) begin
    ...
    p: assert property(a);
    ...
end
```

In the above example, the user will get report of the assertion failure one cycle after a got value 0.

Execution flow

Concurrent assertions do not follow the execution flow, and therefore they may be misleading. Consider the following example:

```
always @(posedge clk) begin
    a = b1;
    if (a) begin
        c <= d;
        p: assert property(c == $past(d));
    end
    a = b2;
    ...
end
```

The assertion p is equivalent to `assert property(a |-> c == $past(d))`; but the value of a is sampled, and it is always equal to the old value of b2, assuming that a is not modified outside this procedure. Note that it is usually not recommended to mix blocking and nonblocking assignments in the same procedure.

Same situation illustrated for loops:

```
always @(posedge clk) begin
    for (int i = 0; i < 3; i++) begin
        b = a[i];
        p1: assert property(a[i]);
        p2: assert property(b);
        ...
    end
    ...
end
```

In the above example the assertion p1 checks the values of a[0], a[1], and a[2], while the assertion p2 checks the value of a[2] only.

Assertions in functions/tasks

Though the embedding of concurrent assertions in procedural code has been defined, concurrent assertions cannot be embedded into functions/tasks. There are several problems here:

- To allow assertions in functions/tasks, the functions or tasks should be inlinable, it is not clear how can concurrent assertions be defined for arbitrary functions/tasks.
- If to adopt inlining semantics, the assertion naming and scoping issues need to be resolved. The easiest way to do that is to use file and line info in function names, but thus generated name may be

non-unique, it is not verilogish and is inconvenient for practical use, since it is unstable and unreadable (when lines are added or deleted from the file, the assertion name can change),

- It is not clear how to define assertion clock. The clock may be inherited from the function instantiation context, but if there is no clock to inherit, there is no appropriate syntax to define the clock for functions. E.g., suppose that f contains concurrent assertions:

```
always_comb begin
  a = f(x); // which clock to provide?
  ...
end
```

- What happens if a concurrent assertion contains a function containing other concurrent assertions?

Why to allow concurrent assertion embedding?

The question was raised if there are problems in embedding concurrent assertions in procedural code, why to allow such embedding at all? If concurrent assertions are temporal by their nature, why should they be intermixed with the execution flow? Let immediate and deferred assertions do the work.

There are several reasons why to allow concurrent assertion embedding.

Concurrent assertions are not very different from other procedural statements

Concurrent assertions are not the only construct that does not follow directly the execution flow. If blocking and nonblocking assignments are intermixed in a procedure, the execution flow is not linear anymore. One can argue that users know the semantics of blocking and nonblocking assignments and therefore there should be no confusion with them. But the same argument is applicable to embedded concurrent assertions – the user should understand concurrent assertion hysteresis behavior as well.

Checking for register transfer correctness

Embedded concurrent assertions are important to check correctness of the register transfer, e.g.,

```
always @(posedge clk) begin
  for (int i = 0; i < 2; i++) begin
    if (en[i]) begin
      if (a) begin
        ...
      end
    else begin
      c1[i] <= c2[i] || b;
      p: assert property (c2[i] |=> c1[i]);
    end
  end
end
end
```

Compare this code with the rewritten fragment avoiding concurrent assertions embedding:

```
always @(posedge clk) begin
  for (int i = 0; i < 2; i++) begin
    if (en[i]) begin
      if (a) begin
        ...
      end
    else begin
      c1[i] <= c2[i];
    end
  end
end
```

```

        end
    end
end
end
for (genvar i = 0; i < 2; i++) begin
    // Two value semantics assumed:
    p: assert property (en[i] && !a && c2[i] | => c1[i]);
end

```

The rewritten code is less readable, more error prone, and more difficult to support.

Here is another rewriting version (suggested by Steven Sharp):

```

for (genvar i = 0; i < 2; i++) begin
    always @(posedge clk) begin
        if (en[i]) begin
            if (a) begin
                ...
            end
            else begin
                c1[i] <= c2[i] || b;
            end
        end
    end
end
p: assert property (en[i] && !a && c2[i] | => c1[i]);
end

```

This version is more compact, but it is still not very readable and is error prone since the assertion antecedent should be inferred manually. A more serious problem is that it means a paradigm shift for RTL coding style regardless whether specific RTL fragments contains assertions or not: the ABV methodology assumes that assertions may be added later and the code should be appropriately organized. It actually means using generate loops only in RTL, and never using regular loops.

Clocked boolean assertions

Another important application of embedded concurrent assertions is using them as clocked boolean assertions to check combinatorial logic, e.g.,

```

always_comb begin
    a = exp1;
    b = exp2;
    p: assert property (@clk $onehot0({a, b}));
end

```

One of reasons to use concurrent assertions here, and not immediate or deferred is to avoid false negatives caused by an interaction with the test bench. Some testbenches drive DUT input signals with small delays, so that the model signals may be inconsistent between clock ticks.

Concurrent assertions in initial procedure

To monitor concurrent assertions on the first clock tick only one has to put them in an initial procedure. In all other cases concurrent assertions are monitored always. Consider the following example:

```

initial p1: assert property (@(posedge clk) ##[0:$] a);
p2: assert property (@(posedge clk) ##[0:$] a);

```

The assertion p1 states that a should eventually happen, while p2 states that a happens infinitely many times. Therefore the capability of embedding concurrent assertions in procedural code is crucial and cannot be reversed.

Backward compatibility

Disallowing embedded concurrent assertions will cause serious backward compatibility issues. There are large amounts of RTL code using this capability.

Potential short-term enhancements

To prevent the most cases of counterintuitive behavior of embedded concurrent assertions it is sufficient to add the following limitations:

- It shall be illegal to reassign in a procedure a variable referenced by an embedded assertion in that procedure.
- It shall be illegal to embed a concurrent assertion in a procedure with both blocking and nonblocking assignments.

Potential long-term enhancements

The following technique may prevent confusion in case of variable reassignment or computation chains in **always_ff**-like procedures. We will conceptually create for each embedded assertion variable a shadow variable, and substitute in the assertion the original variable by its shadow counterpart. The shadow variable is assigned immediately before the assertion. Consider the following example:

```
always @(posedge clk) begin
  a = b1;
  p1: assert property (a);
  a = b2;
  p2: assert property (a);
  ...
end
```

We introduce two shadow variables: a1 and a2:

```
always @(posedge clk) begin
  a = b1;
  a1 = a;
  p1: assert property (a1);
  a = b2;
  a2 = a;
  p2: assert property (a2);
  ...
end
```

Initialization issues should also be taken into account, they haven't been elaborated here. The same technique may work with appropriate modification for (limited subset) of loops and functions, though for functions the solution is more complicated.

Loops

For each variable referenced by some concurrent assertion in a loop, an array of shadow variables is conceptually created. Then the original variable in the assertion is substituted by the corresponding shadow array element. For example,

```
for (int i = 0; i < 3; i++) begin
  a = b[i];
  c = !a;
  p: assert property (c);
end
```

is understood as:

```
for (int i = 0; i < 3; i++) begin  
  a = b[i];  
  c = !a;  
  c1[i] = c;  
  p: assert property (c1[i]);  
end
```

where `c1` is an array of shadow variables for `c`.

Shadow variables and VPI

For the sake of simplicity the above algorithms used shadow variables for each variable referenced by assertion. Not all assertion variables need to have a shadow counterpart, but only for those variables whose shadow counterparts may have a different sampled value than the original variable itself. For real shadow variables the VPI definition needs to be elaborated.

Conclusions

Ability to embed concurrent assertions is important for ABV methodology, and it is an essential language feature to specify assertion monitoring mode. In most cases the behavior of embedded concurrent assertions is natural, the problems arise when a variable referenced by the assertion is reassigned in the procedure, or when blocking computations are done in `always_ff`-like procedures. In the short term this situation may be declared illegal, in the longer term the assertion semantics may be improved by introducing shadow variables. Introducing concurrent assertions in functions and tasks is a tough problem requiring additional syntax and may be done for a limited class of functions and tasks only.