Cliff's votes summarized:

See attached document for explanations.

1500 ___ Yes _X__ No

1556 ___ Yes _X__ No

1580 _X__ Yes ___ No

1608 ___ Yes _X__ No

1609 ___ Yes _X__ No

1612 _X__ Yes ___ No

1715 ___ Yes _X__ No

Cliff's votes:

With full explanations.

1500 ___ Yes _X__ No
I did not understand the problem fully that this restriction aims to fix and there have been other votes against this. I believe it should be discussed in committee (no strong objection but I want to understand this better).

1556 ___ Yes _X__ No
I am generally against adding verbosity to declarations. I understand that the static keyword might help to identify why a variable holds a value each time through a loop, but I am not convinced that this is common or compelling. I could be persuaded otherwise, but for now I vote no.

1580 _X__ Yes ___ No

1608 ___ Yes _X__ No
I found the verbiage to be confusing. Examples would help, but I at least need to hear the explanation and have the ability to ask what all of this means before I vote yes. I am not even sure if the terms being used to define the appropriate actions have themselves been defined. References to these assignment terms would help.

1609 ___ Yes _X__ No
I think this proposal is okay but I voted no in order to get an answer to this question. Am I allowed to either import::* or import::type outside of the class and use package items in a class? I think the answer is yes, but I want to be sure before I change my vote to yes on 1609.

1612 _X__ Yes ___ No

1715 ___ Yes _X__ No
I want to understand this need better before I vote yes. Is this enhancement being proposed because clocking_vars are triggered in the Observed Region? Don't we see those triggers in the Reactive regions with simple @(clocking_var) constructs? Isn't this similar to VHDL's var'event construct? Perhaps this should extend to other Verilog constructs. Example:

```
always_ff @(posedge clk or negedge rst_n or negedge set_n)
  if      (!rst_n) q <= '0;
  else if (!set_n) q <= '1;
  else             q <= d;
```

The above has the well-known problem that if rst_n and set_n are both asserted and if rst_n is removed first, the flip-flop, which should set the q output (because set_n is still low) fails to do so in simulation until the next posedge clk. But we cannot remove the negedge on the rst_n and set_n signals in the sensitivity list; otherwise, the flip-flop could assign data if rst_n is removed when set_n is not asserted (no rst_n and no set_n - therefore clock the d input without a clock signal).

```
always_ff @(posedge clk or rst_n or set_n)
  if      (!rst_n)       q <= '0;
  else if (!set_n)       q <= '1;
  else if (clk.triggered) q <= d;
  // else no change when rst_n or set_n are removed
```

Would solve the problem, if it were legal.