3033. Proposal: Allow procedural control statements, continuous and blocking assignments in checkers.

This proposal is implemented on top of 3213: Update definition of sampled value.

This proposal addresses the following Mantis items:

- 2743: Allow subroutine_call_statement in a checker
- 2809: Checker instantiation in checkers' always procedure
- 3033: Allow procedural control statements is checkers
- 3034: Allow continuous and blocking assignments in checkers
- 3035: More flexible definition of checker argument sampling

Motivation

The goal of this proposal is to improve checker modeling usability and to make formal verification (FV) modeling syntactically similar to RTL. Currently there are two essentially different languages and coding styles for RTL and FV modeling: in RTL continuous assignments and procedural control statements are used whereas in FV modeling functions and let statements are used instead. This proposal also includes a fix of erratum 2809: Checker instantiation in checkers' always procedure.

This proposal introduces the following features:

- New checker procedures: **always_comb**, **always_latch**, and **always_ff**.
- Continuous assignments of checker variables
- Procedural conditional and looping statements in checkers
- Immediate assertions in checkers
- Task invocation in checkers. This enhancement is required, for example, to instrument checker procedures with `$display` statements.
- **let** declarations in checker procedures.

Example 1:

The following checker:

```
checker check(bit[3:0] a, …);
  let e(i) = f(i, …);
  a1: assert #0 ({e(0),e(1),e(2),e(3)} > 1);
  a2: assert #0 (g(e(0), e(1)));
endchecker : check
```

may be rewritten in a more conventional way using the proposed constructs as:

```
checker check(bit[3:0] a, …);
  bit [3:0] x;
  always_comb
```

```
      for (int i = 0; i < 4; i++)
        x[i] = f(i, …);
    a1: assert #0 (x > 1);
    a2: assert #0 (g(x[1:0]));
  endchecker : check
```

Example 2:

The following checker fragment:

```
  function bit next_window (bit win);
    if (reset || win && end) return 1'b0;
    if (!win && start_flag)  return 1'b1;
    return win;
  endfunction
  always @(clock)
     window <= next_window(window);
```

may be rewritten in a more conventional way using the proposed constructs as:

```
  always_ff @(clock) begin
    if (reset || win && end) window <= 1'b0;
    if (!win && start_flag) window <= 1'b1;
  end
```

Also, introduction of checker continuous assignments is required to support checker output arguments (see Mantis 2093):

```
  checker mutex(input event clk = $inferred_clock, output logic a, b);
     default clocking @clk; endclocking
     rand logic a1, b1;
     m1: assume property ($onehot0({a1,b1}));
     assign a = a1;
     assign b = b1;
  endchecker
```

This proposal also incorporates 3035: More flexible definition of checker argument sampling:

Currently all checker arguments are defined to be sampled. This is problematic when the checker is built on top of deferred assertions. The ugly workaround is to const cast all the actual arguments of its instance. However, this workaround will not work to infer the reset value. For example:

```
checker c(a, rst = $inferred_disable);
  a1: assert #0 (rst || $onehot0(a));
endchecker : c
//...
module m(...);
default disable iff reset;
...
always_comb begin
  x = ...;
  y = ...;
  c c1(const'({x, y}));
end
...
endmodule
```

In this case the values of `x` and `y` are not sampled, but the value of `reset` is sampled.

In this proposal instead of sampling checker arguments, an implicit sampling of all variables in **`always_ff`** procedure except of its event control, is introduced. Expression sampling in **`always_ff`** procedures makes

checkers behave deterministically and not to depend on the order of process execution. Consider the following example:

```
checker check2(logic a, event clk);
   logic x1;
   always_ff @clk
      x1 <= a;
   a1: assert property (@clk a |=> x1);
endchecker : check2

module m(logic a, clock);
   logic x2;
   always_ff @(posedge clock)
      x2 <= a;
   a2: assert property @(posedge clock) a |=> x2);
module : m
```

Because the values in **always_ff** procedures in checkers are sampled, the evaluation order is well defined: x1 is assigned the value of a that it had before the clock change. The assertion a1 passes in this case. However, because the values used in **always_ff** procedures in modules are not sampled, the behavior of the assertion a2 is nondeterministic. If the clock changes first, a2 passes. If a changed first then x2 is assigned the new value of a, and the assertion a2 may fail.

Note that if in the above example a is a design variable, the behavior of the checker without implicit variable sampling in **always_ff** procedures would have been deterministic, but non-intuitive. Since the checker events are scheduled in the Reactive region, the assignment to x1 would have picked the new value of a, and thus would have represented combinational, and not sequential logic, as would be natural to expect.

This checker determinism comes at a price of several restrictions, such as prohibition of using blocking assignments in **always_ff** procedures as described in 17.7.1.

Erratum 2809: Checker instantiation in checkers' always procedure.

In the current LRM, in 17.5 Checker procedures, it is written:

An always procedure in a checker body may contain deferred and concurrent assertions, nonblocking variable assignments (see 17.7.1) and a procedural timing control statement using an event control. All other statements shall not appear inside an always procedure.

From here it follows that a checker cannot be instantiated in an always procedure of another checker.

Nevertheless, 17.3.2 Nested checker instantiations explicitly discusses what happens if one checker is instantiated in another checker.

Since checker instantiation in always procedures of other checkers in not required for practical needs, such instantiation is completely disallowed in the current proposal.

Let declarations in checker procedures

Let declarations are useful in checker procedures, especially in initial and **always_ff** where blocking assignments are not supported. This limitation was not introduced on purpose and should be removed.

The following is not backward compatible in the new proposal:

Deferred assertions in checkers

According to the LRM 2009 the checker arguments are sampled. If these arguments are used in a deferred assertion, the arguments of deferred assertions will also be sampled. According to the current proposal the

arguments of checkers are not sampled, therefore the behavior of deferred assertions in checkers will change. This is done on purpose, as explained in the example above.

Checker variable NBA

Currently the RHS of a checker variable NBA is not sampled. Since the checker arguments are sampled the NBA result is usually the same as if its RHS were sampled. However, if the RHS of an NBA contains an XMR, this XMR is not sampled. According to this proposal everything in **`always_ff`** procedure is sampled, which is backward incompatible in case there are XMR in the RHS of a NBA, as illustrated in the following example:

```
checker check(bit a, b, event clk);
   bit c, d;
   always_ff @(posedge clk) begin
      c <= a && b;
      d <= a && top.unit1.block1.b;
   end
endchecker : check
```

According to the current definition `a && b` is effectively sampled because `a` and `b` are checker arguments, and are therefore sampled. However, `top.unit1.block1.b` is not sampled. According to this proposal `top.unit1.block1.b` is also sampled. The latter is the desired behavior.

16.4.3 Deferred assertions outside procedural code

REPLACE

A deferred assertion statement may also appear outside procedural code, used as a *module_common_item*.

WITH

A deferred assertion statement may also appear outside procedural code~~, used as a *module_common_item*~~, in which case it is referred to as a *static deferred assertion*.

ADD at the end of the subclause

~~See 17.3 for~~Static deferred assertions ~~outside procedural code~~ in ~~a checker~~checkers are described in 17.3.

16.5.1 Sampling

REPLACE on top of Mantis 3213

Concurrent assertions and several other constructs (such as checker actual arguments, see 17.3) have special rules for sampling values of their expressions.

WITH

Concurrent assertions and several other constructs (such as ~~checker actual arguments, see 17.3~~ variables referenced in an **`always_ff`** procedure in a checker, see 17.5) have special rules for sampling values of their expressions.

17.1 Overview

REPLACE

The modeling mechanism in checkers is limited to nonblocking assignments only. Each variable declared in a checker may be either deterministic or random. Checker modeling is explained in 17.7. Random variables are

useful to build abstract nondeterministic models for formal verification. Reasoning about nondeterministic models is sometimes much easier than reasoning about deterministic RTL models.

WITH

~~The modeling mechanism in checkers is limited to nonblocking assignments only.~~ The modeling mechanism in checkers is similar to the modeling mechanism in modules and interfaces, though several limitations apply. For example, no nets can be declared and assigned in checkers. On the other hand, checkers allow non-deterministic modeling, which does not exist in modules and interfaces. Each variable declared in a checker may be either deterministic or random. Checker modeling is explained in 17.7. Random variables are useful to build abstract nondeterministic models for formal verification. Reasoning about nondeterministic models is sometimes much easier than reasoning about deterministic RTL models.

17.2 Checker declaration

REPLACE in Syntax 17-1—Checker declaration syntax and in A.1.2 Checker items

checker_declaration ::= // from A.1.2
    checker checker_identifier [**(** [checker_port_list] **)**] **;**
        { checker_or_generate_item }
    endchecker [**:** checker_identifier]

WITH

checker_declaration ::= // from A.1.2
    checker checker_identifier [**(** [checker_port_list] **)**] **;**
        { { attribute_instance } checker_or_generate_item }
    endchecker [**:** checker_identifier]

REPLACE in Syntax 17-1—Checker declaration syntax and in A.1.8 Checker items

checker_or_generate_item ::=
    checker_or_generate_item_declaration
    | initial_construct
    | checker_always_construct
    | final_construct
    | assertion_item
    | checker_generate_item

WITH

checker_or_generate_item ::=
    checker_or_generate_item_declaration
    | initial_construct
    ~~| checker_always_construct~~
    | always_construct
    | final_construct
    | assertion_item
    | continuous_assign
    | checker_generate_item

REPLACE in Syntax 17-1—Checker declaration syntax and in A.1.8 Checker items

checker_always_construct ::= **always** statement

WITH

~~checker_always_construct ::= **always** statement~~

REPLACE

— **initial**, **always** and **final** procedures (see 9.2)

WITH

— **initial**, **always_comb**, **always_latch**, **always_ff** and **final** procedures (see 9.2)

17.3 Checker instantiation

REPLACE

A checker has different behavior depending on whether it is instantiated inside or outside procedural code. A checker instantiation in procedural code is referred to as a *procedural checker instance*. A checker instantiation outside procedural code is referred to as a *static checker instance*. The differences in behavior are described in 17.3.1. (See 16.15.6 for the corresponding definitions of procedural and static assertion statements.)

When a checker is instantiated, actual arguments are passed to the checker. The mechanism for passing arguments to a checker is similar to the mechanism for passing arguments to a property (see 16.13), and each formal argument shall be assigned the sampled value of its actual argument during the Preponed region of each time step, with the following exceptions and clarifications:

— If `$` is an actual argument to a checker instance, then the corresponding formal argument shall be untyped and each of its references either shall be an upper bound in a *cycle_delay_const_range_expression* or shall itself be an actual argument in an instance of a named sequence or property, or in a checker instance.

— If an actual argument contains any subexpression that is a **const** cast or automatic value from procedural code, then the corresponding formal argument shall be used only in static assertion statements (see 16.15.6) or static checker instances within the checker. In such cases, the current value of each such subexpression shall be substituted before sampling the full actual argument, whenever a static assertion statement in the checker or a statically instantiated subchecker is added to the pending procedural assertion queue (see 16.15.6.1 and 17.3.1).

— Arguments that cannot be sampled, such as events, sequences, and properties, are treated similarly to such arguments for sequences and properties (see 16.8): they are substituted directly for the formal argument when it is used in statements or expressions within the checker.

— If the checker is instantiated within another checker, then all formal arguments are considered to be directly connected to their actual arguments, as in a module instantiation. This also means that if the actual argument is connected to a formal in the parent checker that uses a **const** cast or automatic value from procedural code, it shall only appear in static assertion statements or static checker instantiations.

WITH

A checker has different behavior depending on whether it is instantiated inside or outside procedural code. A checker instantiation in procedural code is referred to as a *procedural checker instance*. A checker instantiation outside procedural code is referred to as a *static checker instance*. ~~The differences in behavior are described in 17.3.1. (~~See 16.15.6 for the corresponding definitions of procedural and static assertion statements.~~)~~

When a checker is instantiated, actual arguments are passed to the checker. The mechanism for passing arguments to a checker is similar to the mechanism for passing arguments to a property (see 16.13).~~, and each formal argument shall be assigned the sampled value of its actual argument during the Preponed region of each time step, with the following exceptions and clarifications:~~ The following restrictions apply:

— ~~If~~ As in the case of sequences and properties, if `$` is an actual argument to a checker instance, then the corresponding formal argument shall be untyped and each of its references either shall be an upper bound in a *cycle_delay_const_range_expression* or shall itself be an actual argument in an instance of a named sequence or property, or in a checker instance.

— If an actual argument contains any subexpression that is a **`const`** cast or automatic value from procedural code, then the corresponding formal argument shall ~~be used only in static assertion statements (see 16.15.6) or static checker instances~~ not be used in a continuous assignment or in the procedural code within the checker. ~~In such cases, the current value of each such subexpression shall be substituted before sampling the full actual argument, whenever a static assertion statement in the checker or a statically instantiated subchecker is added to the pending procedural assertion queue (see 16.15.6.1 and 17.3.1).~~

Formatted: Font color: Red, Strikethrough

~~— Arguments that cannot be sampled, such as events, sequenc5es, and properties, are treated similarly to such arguments for sequences and properties (see 16.8): they are substituted directly for the formal argument when it is used in statements or expressions within the checker.~~

— If the checker is instantiated within another checker, then all formal arguments are considered to be directly connected to their actual arguments, as in a module instantiation. ~~This also means that if the actual argument is connected to a formal in the parent checker that uses a **`const`** cast or automatic value from procedural code, it shall only appear in static assertion statements or static checker instantiations.~~ A checker shall not be instantiated in a procedure of another checker.

REPLACE

17.3.1 Behavior of instantiated checkers

WITH

Note to the editor: Delete the subclause heading.

~~17.3.1 Behavior of instantiated checkers~~

REPLACE in 17.3.1

Immediate assertions, including deferred assertions, are handled normally as described in 16.3 and 16.4. Procedural concurrent assertion statements in a checker shall be treated just like other procedural assertion statements as described in 16.15.6. However, static concurrent assertion statements within a checker are treated as if they appear at the checker's instantiation point:

— If the checker is static, the assertion statements are continually monitored, and begin execution on any time step matching their initial clock event.

— If the checker is procedural, all static assertion statements in the checker are added to the pending procedural assertion queue for their process when the checker instantiation is reached in process execution, and then may mature or be flushed like any procedural concurrent assertion (see 16.15.6.2).

— If the checker is statically instantiated inside another checker, any of its static assertions are treated as if instantiated in the parent checker, and thus will also be queued when an instantiation of its ~~toplevel~~top-level ancestor in the checker hierarchy is visited in procedural code.

— WITH

~~Immediate assertions, including deferred assertions, are handled normally as described in 16.3 and 16.4.~~ Procedural concurrent assertion statements in a checker shall be treated just like other procedural assertion statements as described in 16.15.6. However, static ~~concurrent~~ assertion statements within a checker are treated as if they appear at the checker's instantiation point~~:~~. If the checker is instantiated inside ~~another checker~~some scope, any of its static assertions, both concurrent and deferred, are treated as if instantiated in ~~the parent checker.~~this scope. Therefore, the following applies for static concurrent assertions within a checker:

— If the checker is static, the concurrent assertions ~~assertion statements~~ are continually monitored, and begin execution on any time step matching their initial clock event.

— If the checker is procedural, all static concurrent assertions ~~assertion statements~~ in the checker are added to the pending procedural assertion queue for their process when the checker instantiation is reached in process execution, and then may mature or be flushed like any procedural concurrent assertion (see 16.15.6.2).

— If the checker is statically instantiated inside another checker, any of its static assertions, concurrent, ~~assertions~~and deferred, are treated as if instantiated in the parent checker, and thus will be treated as procedural assertions ~~also be queued~~ when an instantiation of its toplevel ancestor in the checker hierarchy is visited in procedural code. For example, concurrent assertions will be queued in this case.

REPLACE in 17.3.1

```
checker c1(event clk, logic[7:0] a, b);
   logic [7:0] sum;
   always @(clk) begin
      sum <= a + ~~1'b1~~1'b1;
      p0: assert property (sum < `MAX_SUM);
   end
   p1: assert property (@clk sum < `MAX_SUM);
   p2: assert property (@clk a != b);
endchecker
```

WITH

```
checker c1(event clk, logic[7:0] a, b);
   logic [7:0] sum;
   always @(clk) begin
      sum <= a + ~~1'b1~~1'b1;
      p0: assert property (sum < `MAX_SUM);
   end
   p1: assert property (@clk sum < `MAX_SUM);
   p2: assert property (@clk a != b);
   p3: assert #0 ($onehot(a));
endchecker
```

REPLACE in 17.3.1

— For instance `check_outside`, `p1` and `p2` are checked at every positive clock edge. For instance `check_inside`, `p1` and `p2` are queued to mature and be checked on any positive clock edge when `en` is true. For `check_loop`, three procedural instances of `p1` and `p2` are queued to mature on any positive clock edge. For `p1`, all three instances are identical, using the sampled value of `sum`; but for `p2`, the three instances compare the sampled value of `in1` to the sampled value of `in_array` indexed by constant `v1` values of 5, 10, and 20 respectively.

Formatted: Body

Formatted: Font color: Blue

Formatted: Font color: Red, Strikethrough

WITH

— For checker instance `check_outside`, `p1` and `p2` are checked at every positive clock edge. For checker instance `check_inside`, `p1` and `p2` are queued to mature and be checked on any positive clock edge when `en` is true. For `check_loop`, three procedural instances of `p1` and `p2` are queued to mature on any positive clock edge. For `p1`, all three instances are identical, using the sampled value of `sum`; but for `p2`, the three instances compare the sampled value of `in1` to the sampled value of `in_array` indexed by constant `v1` values of 5, 10, and 20 respectively.

— For checker instance `check_outside`, `p3` is checked ~~continuously.~~whenever `a` changes. In checker instances `check_inside` and `check_loop` deferred assertion `p3` behaves as a procedural deferred assertion placed at the instantiation point of its checker.

Note to the editor:

DELETE subclause 17.3.2 Nested checker instantiations

17.5 Checker procedures

REPLACE

An **`initial`** procedure in a checker body may contain deferred and concurrent assertions and a procedural timing control statement using an event control only.

WITH

An **`initial`** procedure in a checker body may contain **`let`** declarations, immediate, deferred and concurrent assertions and a procedural timing control statement using an event control only.

REPLACE

An **`always`** procedure in a checker body may contain deferred and concurrent assertions, nonblocking variable assignments (see 17.7.1) and a procedural timing control statement using an event control. All other statements shall not appear inside an **`always`** procedure.

WITH

~~An **`always`** procedure in a checker body may contain deferred and concurrent assertions, nonblocking variable assignments (see 17.7.1) and a procedural timing control statement using an event control. All other statements shall not appear inside an **`always`** procedure.~~

Formatted: Font color: Red, Strikethrough

The following forms of always procedures are allowed in checkers: **`always_comb`**, **`always_latch`**, and **`always_ff`**. Checker always procedures may contain the following statements:

— Blocking assignments (see 10.4.1; **`always_comb`** and **`always_latch`** procedures only)

— Nonblocking assignments (see 10.4.2)

— Selection statements (see 12.4 and 12.5)

— Loop statements (see 12.7)

— Timing event control (see 9.4.2; **`always_ff`** procedure only)

— Subroutine calls (see Clause 13)

— **let** declarations

Except for the variables mentioned in the event control, all other expressions in **always_ff** procedures are sampled (see 16.5.1). It follows from this rule that the expressions in immediate and deferred assertions instantiated in this procedure are also sampled. Expressions in **always_comb** and **always_latch** procedures are not implicitly sampled and the assignments appearing in these procedures use the current values of their expressions. For example:

```
checker check(logic a, b, c, clk, rst);
   logic x, y, z, v, t;
   assign x = a;     // current value of a
   always_ff @(posedge clk or negedge rst)   // current values of clk and rst
   begin
      a1: assert (b);      // sampled value of b
      if (rst)             // current value of rst
         z <= b;           // sampled value of b
      else z <= !c;        // sampled value of c
   end
   always_comb begin
      a2: assert (b);      // current value of b
      if (a)               // current value of a
         v = b;            // current value of b
      else v = !b;         // current value of b
   end
   always_latch begin
      a3: assert (b);      // current value of b
      if (clk)             // current value of clk
         t <= b;           // current value of b
   end
// ...
endchecker : check
```

17.6 Covergroups in checkers

REPLACE

```
always @(posedge clk) begin
   active_d1 <= active;
end
```

WITH

Note to the editor: font change for ‘end’. Should be typeset as a keyword.

```
always_ff @(posedge clk) begin
   active_d1 <= active;
endend
```

REPLACE two occurrences of this line

```
always @(posedge clk) opcode_d1 <= opcode;
```

WITH

```
always_ff @(posedge clk) opcode_d1 <= opcode;
```

17.7 Checker variables

REPLACE

```
always @$global_clock
```

WITH

```
always_ff @$global_clock
```

17.7.1 Checker variable assignments

REPLACE

Checker variables may be assigned using nonblocking procedural assignment only. Blocking procedural assignments to checker variables are not allowed. The formal semantics of free variable assignment is described in F.3.4.6.

The following example illustrates usage of free variable assignments.

```
// Toggling variable:
// a may have either 0101... or 1010... pattern
rand bit a;
always @clk a <= !a;
```

The right-hand side of a checker variable assignment may contain the sequence method triggered (see 16.14.6).

The following rules apply to both regular and free checker variables:

— It shall be illegal to reference a checker variable using its hierarchical name in assignments (see 23.6). For example:

```
checker check(...)
   bit a;
   ...
endchecker

module m(...)
   ...
   check my_check(...);
   ...
   wire x = my_check.a; // Illegal
   bit y;
   ...
   always @(posedge clk) begin
      my_check.a = y; // Illegal
      ...
   end
...
endmodule
```

— Single Assignment Rule (SAR): it shall be illegal to use the same bit of a checker variable in several assignment-like contexts.

Example 1:

```
bit [2:0] a;
...
bit [2:0] b;
always @(posedge clk) begin
   b[1:0] <= a[1:0];
   b[2:1] <= a[2:1]; // Illegal: SAR violation
end
```

This is illegal because there are two assignment statements to `b[1]` (even though the two assignments are to the same value).

Example 2:

```
bit [2:0] a;
...
bit [2:0] b;
always @(posedge clk) begin
   b[1:0] <= a[2:1];
   b[2] <= a[0];
end
```

This is legal because each bit of `b` is assigned only once.

— The left hand side of an assignment shall be the longest static prefix of a select (see 11.5.3). For example:

```
rand bit [3:0] a;
rand bit [1:0] i;
always @clk
   a[i] <= !a[i]; // Illegal
```

— A checker variable may not be assigned in an **initial** procedure. For example:

```
bit v;
initial v <= 1'b0; // Illegal
```

WITH

~~Checker variables may be assigned using nonblocking procedural assignment only. The following example illustrates usage of free variable assignments.~~

```
// Toggling variable:
// a may have either 0101... or 1010... pattern
rand bit a;
always @clk a <= !a;
```

~~The right hand side of a checker variable assignment may contain the sequence method triggered (see 16.14.6).~~

Checker variables may be assigned using blocking and nonblocking procedural assignments, or non-procedural continuous assignments.

~~The following rules apply to both regular and free checker variables:~~

~~— It shall be illegal to reference a checker variable using its hierarchical name in assignments (see 23.6). For example:~~

```
checker check(...)
   bit a;
   ...
endchecker

module m(...)
   ...
   check my_check(...);
   ...
   wire x = my_check.a; // Illegal
   bit y;
   ...
```

```
  always @(posedge clk) begin
    my_check.a = y; // Illegal
    ...
  end
...
endmodule
```

- Single Assignment Rule (SAR): it shall be illegal to use the same bit of a checker variable in several assignment like contexts.

Example 1:

```
bit [2:0] a;
...
bit [2:0] b;
always @(posedge clk) begin
  b[1:0] <= a[1:0];
  b[2:1] <= a[2:1]; // Illegal: SAR violation
end
```

This is illegal because there are two assignment statements to `b[1]` (even though the two assignments are to the same value).

Example 2:

```
bit [2:0] a;
...
bit [2:0] b;
always @(posedge clk) begin
  b[1:0] <= a[2:1];
  b[2] <= a[0];
end
```

This is legal because each bit of `b` is assigned only once.

- The left-hand side of an assignment shall be the longest static prefix of a select (see 11.5.3). For example:

Formatted: Font color: Red, Strikethrough

```
rand bit [3:0] a;
rand bit [1:0] i;
always @clk
  a[i] <= !a[i]; // Illegal
```

- A checker variable may not be assigned in an **initial** procedure. For example:

```
bit v;
initial v <= 1'b0; // Illegal
```

The following rules and restrictions apply:

— In **always_ff** procedures only nonblocking assignments are allowed.

— Referencing a checker variable using its hierarchical name in assignments (see 23.6) shall be illegal. For example:

```
checker check(...)
  bit a;
  ...
endchecker

module m(...)
```

```
    ...
    check my_check(...);
    ...
    wire x = my_check.a; // Illegal
    bit y;
    ...
    always @(posedge clk) begin
       my_check.a = y; // Illegal
       ...
    end
...
endmodule
```

— Continuous assignments and blocking procedural assignments to free checker variables shall be illegal.

```
checker check1(bit a, b, event clk, ...);
   rand bit x, y, z, v;
   ...
   assign x = a & b; // Illegal
   always_comb
      y = a & b; // Illegal
   always_ff @clk
      z <= a & b; // OK
endchecker : check1
```

— A checker variable may not be assigned in an **`initial`** procedure, but may be initialized in its declaration. For example:

```
bit v;
initial v = 1'b0; // Illegal
bit w = 1'b0;     // OK
```

— The right-hand side of a checker variable assignment may contain the sequence method `triggered` (see 16.14.6).

— The left hand side of a nonblocking assignment may contain a free checker variable. The following example illustrates usage of free variable assignments.

Formatted: DashedList

```
// Toggling variable:
// a may have either 0101... or 1010... pattern
rand bit a;
always_ff @clk a <= !a;
```

17.7.2 Checker variable randomization with assumptions

REPLACE

```
always @(posedge fclk)
```

WITH

```
always_ff @(posedge fclk)
```

17.7.3 Scheduling semantics

REPLACE

Statements and constructs within a checker that are sensitive to changes (e.g., clocking events) are scheduled in the Reactive region (similarly to programs, see 24.3.1).

WITH

Statements and constructs within a checker that are sensitive to changes (e.g., clocking events, continuous assignments), and all blocking statements are scheduled in the Reactive region (similarly to programs, see 24.3.1).

REPLACE

```
always @clk a <= s.triggered;
```

WITH

```
always_ff @clk a <= s.triggered;
```

17.8 Functions in checkers

REPLACE

While procedural statements (**if**, **case**, etc.) may not be placed directly in the **initial** and in the **always** procedures, they may be used in functions called from the right-hand side of a checker variable assignment. The formal arguments and internal variables of functions used in checkers shall not be declared as free variables. However, free variables are allowed to be passed in as actual arguments to a function.

WITH

~~While procedural statements (**if**, **case**, etc.) may not be placed directly in the **initial** and in the **always** procedures, they may be used in functions called from the right-hand side of a checker variable assignment.~~ The formal arguments and internal variables of functions used in checkers shall not be declared as free variables. However, free variables are allowed to be passed in as actual arguments to a function.

17.9 Complex checker example

REPLACE

The checker in the following example makes sure that the expression is true in a window delimited by `start_event` and `end_event`.

```
typedef enum { cover_none, cover_all } coverage_level;
checker assert_window (
   logic test_expr,        // Expression to be true in the window
   sequence start_event,   // Window opens at the completion of the start_event
   sequence end_event,     // Window closes at the completion of the end_event
   event clock = $inferred_clock,
   logic reset = $inferred_disable,
   string error_msg = "violation",
   coverage_level clevel = cover_all //This argument should be bound to an
                                     //elaboration time constant expression
);
   default clocking @clock; endclocking
   default disable iff reset;
   bit window = 0;
   let start_flag = start_event.triggered;
   let end_flag = end_event.triggered;
   // Compute next value of window
   function bit next_window (bit win);
      if (reset || win && end_flag == 1'b1)
         return 1'b0;
      if (!win && start_flag == 1'b1)
```

```
            return 1'b1;
         return win;
      endfunction

      always @(clock)
         window <= next_window(window);

      property p_window;
         start_flag && !window |=> test_expr[*1:$] ##0 end_flag;
      endproperty

      a_window: assert property (p_window) else $error(error_msg);

      generate if (clevel != cover_none) begin : cover_b
         cover_window_open: cover property (start_flag && !window)
         $display("win_open_covered");
         cover_window: cover property (
            start_flag && !window
            ##1 (!end_flag && window) [*0:$]
            ##1 end_flag && window
         ) $display("window covered");
      end : cover_b
      endgenerate
   endchecker : assert_window
```

WITH

~~The checker in the following example makes sure that the expression is true in a window delimited by start_event and end_event.~~

```
   typedef enum { cover_none, cover_all } coverage_level;
   checker assert_window (
      logic test_expr,          // Expression to be true in the window
      sequence start_event,     // Window opens at the completion of the start_event
      sequence end_event,       // Window closes at the completion of the end_event
      event clock = $inferred_clock,
      logic reset = $inferred_disable,
      string error_msg = "violation",
      coverage_level clevel = cover_all //This argument should be bound to an
                                        //elaboration time constant expression
   );
      default clocking @clock; endclocking
      default disable iff reset;
      bit window = 0;
      let start_flag = start_event.triggered;
      let end_flag = end_event.triggered;

      // Compute next value of window
      function bit next_window (bit win);
         if (reset || win && end_flag == 1'b1)
            return 1'b0;
         if (!win && start_flag == 1'b1)
            return 1'b1;
         return win;
      endfunction

      always @(clock)
         window <= next_window(window);

      property p_window;
         start_flag && !window |=> test_expr[*1:$] ##0 end_flag;
      endproperty
```

```
   a_window: assert property (p_window) else $error(error_msg);

   generate if (clevel != cover_none) begin : cover_b
      cover_window_open: cover property (start_flag && !window)
      $display("win_open_covered");
      cover_window: cover property (
         start_flag && !window
         ##1 (!end_flag && window) [*0:$]
         ##1 end_flag && window
      ) $display("window covered");
   end : cover_b
   endgenerate
endchecker : assert_window
```

The checkers in the following examples make sure that the expression is true in a window delimited by `start_event` and `end_event`. When `start_event` and `end_event` are boolean, the checker may be implemented as shown in Example 1.

Example 1.

```
typedef enum { cover_none, cover_all } coverage_level;
checker assert_window1 (
   logic test_expr,        // Expression to be true in the window
   untyped start_event,    // Window opens at the completion of the start_event
   untyped end_event,      // Window closes at the completion of the end_event
   event clock = $inferred_clock,
   logic reset = $inferred_disable,
   string error_msg = "violation",
   coverage_level clevel = cover_all //This argument should be bound to an
                                     //elaboration time constant expression
);
   default clocking @clock; endclocking
   default disable iff reset;
   bit window = 1'b0, next_window = 1'b1;
   bit start_flag, end_flag;
   assign start_flag = start_event;
   assign end_flag = end_event;

   // Compute next value of window
   always_comb begin
      if (reset || window && end_flag == 1'b1)
         next_window =  1'b0;
      else if (!window && start_flag == 1'b1)
         next_window = 1'b1;
      else
         next_window = window;
   end

   always_ff @clock
      window <= next_window;

   property p_window;
      start_flag && !window |=> test_expr[*1:$] ##0 end_flag;
   endproperty

   a_window: assert property (p_window) else $error(error_msg);

   generate if (clevel != cover_none) begin : cover_b
      cover_window_open: cover property (start_flag && !window)
      $display("win_open_covered");
      cover_window: cover property (
         start_flag && !window
         ##1 (!end_flag && window) [*0:$]
```

```
            ##1 end_flag && window
         ) $display("window covered");
      end : cover_b
      endgenerate
   endchecker : assert_window1
```

If `start_event` and `end_event` may be arbitrary sequences, and not necessary boolean values, the checker shall be implemented differently, as shown in Example 2. This case requires a different implementation because the reset of the triggered status of a sequence does not create an event (see 9.4.4), and therefore a sequence `triggered` method shall not be used in the right-hand side of a continuous assignment, or of an assignment in an **`always_comb`** procedure.

Example 2.

```
   typedef enum { cover_none, cover_all } coverage_level;
   checker assert_window2 (
      logic test_expr,        // Expression to be true in the window
      sequence start_event,   // Window opens at the completion of the start_event
      sequence end_event,     // Window closes at the completion of the end_event
      event clock = $inferred_clock,
      logic reset = $inferred_disable,
      string error_msg = "violation",
      coverage_level clevel = cover_all //This argument should be bound to an
                                        //elaboration time constant expression
   );
      default clocking @clock; endclocking
      default disable iff reset;
      bit window = 0;
      let start_flag = start_event.triggered;
      let end_flag = end_event.triggered;

      // Compute next value of window
      function bit next_window (bit win);
         if (reset || win && end_flag == 1'b1)
            return 1'b0;
         if (!win && start_flag == 1'b1)
            return 1'b1;
         return win;
      endfunction

      always @clock
         window <= next_window(window);

      // The rest is the same as in Example 1
   endchecker : assert_window2
```

C.2 Constructs that have been deprecated

ADD

C.2.7 always statement in checkers

The **`always`** procedure in checkers was allowed by IEEE Std 1800-2009, but **`always_comb`**, **`always_latch`**, and **`always_ff`** were forbidden. The limitations imposed on the **`always`** procedure in checkers included the limitations imposed on **`always_ff`** procedures outside checkers. In this version of the standard **`always_comb`**, **`always_latch`**, and **`always_ff`** have been added for checkers. As a result the general **`always`** procedure in checkers would have imposed the same limitations as **`always_ff`** does. Therefore the usage of **`always`** procedures in checkers is deprecated and does not appear in this version of the standard.

REPLACE

F.3.4.6 Checker variable assignment

— rand *t u* ≡ *e* initial assume property (@1 *u* === *e*) .

— **always** @*c u* <= *e* ≡ **always assume property** (@1 $future_gclk(*u*) === c ? *e* : *u*).

WITH

F.3.4.6 ~~Checker~~ Free checker variable assignment

— rand *t u* ≡ *e* initial assume property (@1 *u* === *e*) .

— **always_ff** @*c u* <= *e* ≡ **always_ff assume property** (@1 $future_gclk(*u*) === c ? *e* : *u*) .
If the assignment to *u* is in the scope of one or several conditional statements with a resulting enabling condition *b* then the equivalent assumption shall also be evaluated using the same enabling condition *b* (see F.5.3.1).