This proposal requires the following modifications:

— Modify BNF in Syntax 17-12 to include operator **dist** as part of the expression BNF for assertions, as specified below

— Modify Section 17.12 as specified below.

17.12 Concurent assertions

A property on its own is never evaluated for checking an expression. It must be used within a verification statement for this to occur. A verification statement states the verification function to be performed on the property. The statement can be one of the following:

— **`assert`** to specify the property as a checker to ensure that the property holds for the design

— **`assume`** to specify the property as an assumption for the environment

— **`cover`** to monitor the property evaluation for coverage

A concurrent assertion statement can be specified in:

— an always block or initial block as a statement, wherever these blocks can appear

— a module as a *module_or_generate_item*

— an interface as an *interface_or_generate_item*

— a program as a *non_port_program_item*

— $root

```
procedural_assertion_item ::=
        assert_property_statement
        | assume_property_statement
        | cover_property_statement
concurrent_assertion_item ::=
        concurrent_assert_statement
        | concurrent_cover_statement
concurrent_assert_statement ::=
        [block_identifier:] assert_property_statement
concurrent_cover_statement ::=
        [block_identifier:] cover_property_statement
assert_property_statement::=
        assert property ( property_spec ) action_block
        | assert property ( property_instance ) action_block
assume_property_statement::=
        assume property ( property_spec )
        | assume property ( property_instance )
cover_property_statement::=
        cover property ( property_spec ) statement_or_null
        | cover property ( property_instance ) statement_or_null
```


Syntax 17-16—Concurrent assertion construct syntax (excerpt from Annex A)

The **assert**, **assume** or **cover** statements can be referenced by their optional name. A hierarchical name can be used consistent with the SystemVerilog naming conventions. When a name is not provided, a tool shall assign a name to the statement for the purpose of reporting.

Assertion control tasks are described in Section 22.6.

17.12.1 assert statement

The **assert** statement is used to enforce a **property** as a checker. When the property for the **assert** statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the *action_block* are executed. For example,

```
property abc(a,b,c);
   disable iff (a==2) not @clk (b ##1 c);
endproperty
env_prop: assert property (abc(rst,in1,in2)) pass_stat else fail_stat;
```

When no action is needed, a null statement (i.e.;) is specified. If no statement is specified for ~~the~~ **else**, then `$error` is used as the statement when the assertion fails. The *action_block* shall not include any concurrent **assert**, **assume** or **cover** statement. The *action_block*, however, can contain immediate assertion statements.

Note: The pass and fail statements are executed in the Reactive region. The regions of execution are explained in the scheduling semantics section, Section 14.

17.12.2 assume statement

The purpose of the **assume** statement is to allow properties to be considered as assumptions for formal analysis as well as for dynamic simulation tools. When a property is assumed, the tools constraint the environment so that the property holds.

For formal analysis, there is no obligation to verify that the assumed properties hold. An assumed property may be considered as a hypothesis to prove the asserted properties.

For simulation, the environment must be constrained such that the properties that are assumed shall hold. Like an assert property, an assumed property must be checked and reported if it fails to hold. There is no requirement on the tools to report successes of the assumed properties.

Additionally, for random simulation, biasing on the inputs provides a way to make random choices. An expression may be associated with biasing as shown below

expression **dist {** dist_list **} ;** // from Annex A.1.9

The operator **dist** and the production *dist_list* is explained in Section 12.4.4.

The biasing feature is only useful when properties are considered as assumptions to drive random simulation. When a property with biasing is used in an assertion or coverage, the **dist** operator is equivalent to **inside** operator, and the weight specification is ignored. For example,

```
a1:assume property @(posedge clk) req dist {0:=40, 1:=60} ;
property proto
   @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

This is equivalent to:

```
a1_assertion:assert property req inside {0, 1};
property proto_assertion
  @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

In the above example, signal `req` is specified with distribution in assumption `a1`, and is converted to an equivalent assertion `a1_assertion`.

It should be noted that the properties that are assumed must hold in the same way with or without biasing. When using an **assume** statment for random simulation, the biasing simply provides a means to select values of free variables, according to the specified weights, when there is a choice of selection at a particular time.

Consider an example specifying a simple synchronous request - acknowledge protocol, where variable `req` can be raised at any time and must stay asserted until `ack` is asserted. In the next clock cycle both `req` and `ack` must be deasserted.

Properties governing `req` are:

```
property pr1;
   @(posedge clk) !reset_n |-> !req; //when reset_n is asserted (0),keep req 0
endproperty
property pr2;
   @(posedge clk) ack |=> !req; // one cycle after ack, req must be deasserted
endproperty
property pr3;
   @(posedge clk) req |-> req[*1:$] ##0 ack; // hold req asserted until
                                             // and including ack asserted
endproperty
```

Properties governing `ack` are:

```
property pa1;
   @(posedge clk) !reset_n || !req |-> !ack;
endproperty
property pa2;
   @(posedge clk) ack |=> !ack;
endproperty
```

When verifying the behavior of a protocol controller which has to respond to requests on `req`, assertions `assert_req1` and `assert_req2` should be proven while assuming that statements `a1`, `assume_ack1`, `assume_ack2` and `assume_ack3` hold at all times.

```
a1:assume property @(posedge clk) req dist {0:=40, 1:=60};
assume_ack1:assume property (pr1);
assume_ack2:assume property (pr2);
assume_ack3:assume property (pr3);

assert_req1:assert property (pa1)
   else $display("\n ack asserted while req is still deassrted");
assert_req2:assert property (pa2)
   else $display("\n ack is extended over more than one cycle");
```

Note that **assume** does not provide an action block, as the actions for an assumption serve no purpose.

17.12.3 cover statement

To monitor sequences and other behavioral aspects of the design for coverage, the same syntax is used with the **cover** statement. The tools can gather information about the evaluation and report the results at the end of sim-

ulation. When the property for the **cover** statement is successful, the pass statements can specify a coverage function, such as monitoring all paths for a sequence. The pass statement shall not include any concurrent **assert, assume** or **cover** statement.

~~The **assert, assume** or **cover** statements can be referenced by their optional name. A hierarchical name can be used consistent with the SystemVerilog naming conventions. When a name is not provided, a tool shall assign a name to the statement for the purpose of reporting.~~

~~Assertion control tasks are described in Section 22.6.~~

Coverage results are divided into two: coverage for properties, coverage for sequences.
For sequence coverage, the statement appears as:

cover property `( sequence_spec ) statement_or_null`

The identifier of a particular attempt is called *attemptId*, and the clock tick of the occurrence of the match is called *clock step*.

The results of coverage statement for a property shall contain:

- Number of times attempted
- Number of times succeeded
- Number of times failed
- Number of times succeeded because of vacuity
- Each attempt with an *attemptID* and time
- Each success/failure with an *attemptID* and time

In addition, *statement_or_null* is executed every time a property succeeds.Vacuity rules are applied only when implication operator is used. A property succeeds non-vacuously only if the consequent of the implication contributes to the success.

Results of coverage for a sequence shall include

- Number of times attempted
- Number of times matched (each attempt can generate multiple matches)
- Each attempt with attemptId and time
- Each match with clock step, attemptID, and time

In addition, *statement_or_null* gets executed for every match. If there are multiple matches at the same time, the statement gets executed multiple times, one for each match.

17.5 Sequences

MODIFY syntax box 17-2

sequence_expr ::=

cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
| sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
| ~~expression~~expression_or_dist{ , function_blocking_assignment } [boolean_abbrev]
| (~~expression~~ expression_or_dist{, function_blocking_assignment }) [boolean_abbrev]
| sequence_instance [sequence_abbrev]
| (sequence_expr) [sequence_abbrev]
| sequence_expr **and** sequence_expr
| sequence_expr **intersect** sequence_expr
| sequence_expr **or** sequence_expr
| **first_match** (sequence_expr)
| expression **throughout** sequence_expr
| sequence_expr **within** sequence_expr

cycle_delay_range ::=
constant_expression
| ## [cycle_delay_const_range_expression]

sequence_instance ::=
sequence_identifier [(actual_arg_list)]

formal_list_item ::=
formal_identifier [= actual_arg_expr]

actual_arg_list ::=
(actual_arg_expr { , actual_arg_expr })
| (. formal_identifier (actual_arg_expr) { , . formal_identifier (actual_arg_expr) })

actual_arg_expr ::=
event_expression
| $

boolean_abbrev ::=
consecutive_repetition
| non_consecutive_repetition
| goto_repetition

sequence_abbrev ::= consecutive_repetition

consecutive_repetition ::= [* const_or_range_expression]

non_consecutive_repetition ::= [*= const_or_range_expression]

goto_repetition ::= [*> const_or_range_expression]

const_or_range_expression ::=
constant_expression
| cycle_delay_const_range_expression

cycle_delay_const_range_expression ::=
constant_expression : constant_expression
| constant_expression : $

expression_or_dist::=
expression
| expression **dist {** dist_list **}**

