

Mantis 3295 & 2341

Note: This document is based on P1800-2012-draft1.

## Section 20.11

### Assertion control system tasks

#### Motivation

Currently System Verilog does not provide a way to control assertions based on their type. There is no way for the users to only disable cover directives but keep assert and assumes enabled. Similarly, there is no way to only enable concurrent assertions and switch off immediate assertions. This proposal suggests enhancements to be able to do this. One thing to decide is that do we want to provide selection based on severity also. Since, severity of the assertion is not defined in the LRM currently, it is not possible to do that without defining the severity first. We talked about using comparison of attributes in the argument of new system task but that does not seem to be legal as per the rules defined for attributes in SystemVerilog.

Now that we are going to have a new system task with better capabilities, we are going to also define assertion action control tasks based on this new task.

Also updating the text to clarify that any scopes can be specified as arguments to these tasks (mantis 2341)

#### Suggested Resolution

*In Syntax 20.11, change:*

```
assert_control_task ::=
assert_task [ ( levels [ , list_of_modules_or_assertions ] ) ] ;
assert_task ::=
$asserton
| $assertoff
| $assertkill
list_of_modules_or_assertions ::=
module_or_assertion { , module_or_assertion }
```

```
module_or_assertion ::=
module_identifier
| assertion_identifier
| hierarchical_identifier
```

TO:

```
assert_control_task ::=
assert_task [ ( levels [ , list_of_modulesscopes_or_assertions ] ) ] ;
| assert_action_task [ ( levels [ , list_of_scopes_or_assertions ] ) ] ;
| $assertcontrol ( control_type [ , [ assertion_type ] [ , [ directive_type ] [ , [ levels ] [ , list_of_scopes_or_assertions ] ] ] ] ) ;
```

```
assert_task ::=
$asserton
| $assertoff
| $assertkill
```

```
assert_action_task ::=
$assertpasson
| $assertpassoff
| $assertfailon
| $assertfailoff
| $assertnonvacuouson
| $assertvacuousoff
```

```
list_of_modulescopes_or_assertions ::=
modulescope_or_assertion { , modulescope_or_assertion }
```

```
modulescope_or_assertion ::=
module_identifier
| assertion_identifier
| hierarchical_identifier
```

Add the following tables after Syntax 20.11:

**Table 20-5: Values for control\_type for assertion control tasks**

control_type values	Effect
1	Lock
2	Unlock
3	On
4	Off
5	Kill
6	PassOn
7	PassOff
8	FailOn
9	FailOff
10	NonvacuousOn
11	VacuousOff

**Note to the Editor: If 3206 is approved, pick the following table 20-6:**

**Table 20-6: Values for assertion\_type for assertion control tasks**

assertion_type values	Types of assertions affected
1	Concurrent
2	Simple Immediate
4	Observed Deferred Immediate
8	Final Deferred Immediate
16	Expect

**Note to the Editor: If 3206 is not approved, pick the following table 20-6:**

**Table 20-6: Values for `assertion_type` for assertion control tasks**

<code>assertion_type</code> values	Types of assertions affected
1	Concurrent
2	Simple Immediate
4	Deferred Immediate
8	Expect

**Table 20-7: Values for `directive_type` for assertion control tasks**

<code>directive_type</code> values	Types of directives affected
1	Assert directives
2	Cover directives
4	Assume directives

**Note to the Editor: Please change numbers of tables after these accordingly.**

*In the body of Clause 20.11, change:*

SystemVerilog provides the following three system tasks to control the evaluation of assertions (see 16.2):  
— `$assertoff` shall stop the checking of all specified assertions until a subsequent `$asserton`. No new attempts will be started. Attempts that are already executing for the assertions, and their pass or fail statements, are not affected.

In the case of a deferred assertion (see 16.4), currently queued reports are not flushed and may still mature, though further checking is prevented until the `$asserton`. In the case of a pending procedural assertion instance (see 16.15.6), currently queued instances are not flushed and may still mature, though no new instances may be queued until the `$asserton`.

— `$assertkill` shall abort execution of any currently executing attempts for the specified assertions and then stop

the checking of all specified assertions until a subsequent `$asserton`. This also flushes any queued pending reports of deferred assertions (see 16.4) or pending procedural assertion instances (see 16.15.6) that have not yet matured.

— `$asserton` shall reenable the execution of all specified assertions.

The details related to the behavior of `$assertkill` and `$assertoff` for assertions referring to global clocking future sampled value functions are explained in 16.9.4.

When invoked with no arguments, the system task shall apply to all assertions. When the task is specified with arguments, the first argument indicates levels of the hierarchy, consistent with the corresponding argument to the `$dumpvars` system task (see 21.7.1.2). Subsequent arguments specify which scopes of the model to control. These arguments can specify entire modules or individual assertions.

Table 20-5 lists the VPI callbacks (see 36.9.2 and 39.4) corresponding to the assertion control system task's invocation.

TO:

SystemVerilog provides the ~~following three system tasks~~ `$assertcontrol` system task to control the evaluation of assertions (16.2). The `$assertcontrol` system task can also be used to control the execution of assertion action blocks associated with assertions and `expect` statements. This system task

provides the capability to enable/disable/kill the assertions based on assertion type or directive type. Similarly, this task also provides the capability to enable/disable action block execution of assertions and expect statements based on assertion type or directive type. The arguments for the `$assertcontrol` system task are described below:

- `control_type`: This argument controls the effect of the `$assertcontrol` system task. The type of this argument is integer. The valid values for this argument are described in table 20-5.

**Note to the Editor: If 3206 is approved, pick the following bullet:**

- `assertion_type`: This argument selects the assertion types that are affected by the `$assertcontrol` system task. The type of this argument is integer. The valid values for this argument are described in table 20-6. Multiple `assertion_type` values can be specified at a time by OR-ing different values. For example, a task with `assertion_type` value of 3 (which is the same as `Concurrent|SimpleImmediate`) shall apply to concurrent and simple immediate assertions. If `assertion_type` is not specified, then it defaults to 31 (`Concurrent|SimpleImmediate|ObservedDeferredImmediate|FinalDeferredImmediate|Expect`) and the system task applies to all types of assertions and `expect` statements.

**Note to the Editor: If 3206 is not approved, pick the following bullet:**

- `assertion_type`: This argument selects the assertion types that are affected by the `$assertcontrol` system task. The type of this argument is integer. The valid values for this argument are described in table 20-6. Multiple `assertion_type` values can be specified at a time by OR-ing different values. For example, a task with `assertion_type` value of 3 (which is the same as `Concurrent|SimpleImmediate`) shall apply to concurrent and simple immediate assertions. If `assertion_type` is not specified, then it defaults to 15 (`Concurrent|SimpleImmediate|DeferredImmediate|Expect`) and the system task applies to all types of assertions and `expect` statements.
- `directive_type`: This argument selects the directive types that are affected by the `$assertcontrol` system task. The type of this argument is integer. The valid values for this argument are described in table 20-7. This argument is checked only for assertions. Multiple `directive_type` values can be specified at a time by OR-ing different values. For example, a task with `directive_type` value of 3 (which is same as `Assert|Cover`) shall apply to assert and cover directives. If `directive_type` is not specified, then it defaults to 7 (`Assert|Cover|Assume`) and the system task applies to all types of directives.
- `levels`: This argument specifies the levels of hierarchy, consistent with the corresponding argument to the `$dumpvars` system task (see 21.7.1.2). If this argument is not specified, it defaults to 0. The type of this argument is integer.
- `list_of_scopes_or_assertions`: This argument specifies which scopes of the model to control. These arguments can specify any scopes or individual assertions.

The effect of the `$assertcontrol` system task is determined by the value of its first argument `control_type`, which shall be an integer expression. The effect of the system task based on the value of `control_type` is described below:

- `Lock`: A value of 1 for this argument prevents status change of all specified assertions and expect statements until they are unlocked. Once an `$assertcontrol` with `control_type` of value 1

(Lock) is applied to an assertion or expect statement, it becomes locked and no `$assertcontrol` shall affect it until the locked state is removed by a subsequent `$assertcontrol` with a `control_type` value of 2 (Unlock).

— **Unlock:** A value of 2 for this argument shall remove the locked status of all specified assertions and expect statements.

— **On:** A value of 3 for this argument shall re-enable the execution of all specified assertions. This `control_type` value does not affect expect statements.

— ~~**\$assertoff Off:**~~ A value of 4 for this argument shall stop the checking of all specified assertions until a subsequent ~~**\$asserton**~~ `$assertcontrol` with a `control_type` value of 3 (On). No new attempts will be started. Attempts that are already executing for the assertions, and their pass or fail statements, are not affected. In the case of a deferred assertion (see 16.4), currently queued reports are not flushed and may still mature, though further checking is prevented until a subsequent ~~the~~ ~~**\$asserton**~~ `$assertcontrol` with a `control_type` value of 3 (On). In the case of a pending procedural assertion instance (see 16.15.6), currently queued instances are not flushed and may still mature, though no new instances may be queued until a subsequent ~~the~~ ~~**\$asserton**~~ `$assertcontrol` with a `control_type` value of 3 (On). This `control_type` value does not affect expect statements.

— ~~**\$assertkill Kill:**~~ A value of 5 for this argument shall abort execution of any currently executing attempts for the specified assertions and then stop the checking of all specified assertions until a subsequent ~~**\$asserton**~~ `$assertcontrol` with a `control_type` value of 3 (On). This also flushes any queued pending reports of deferred assertions (see 16.4) or pending procedural assertion instances (see 16.15.6) that have not yet matured. This `control_type` value does not affect expect statements.

— ~~**\$asserton shall reenable the execution of all specified assertions.**~~

— **PassOn:** A value of 6 for this argument shall enable execution of the pass action for vacuous and nonvacuous success of all the specified assertions. An assertion that is already executing, including execution of the pass or fail action, is not affected.

— **PassOff:** A value of 7 for this argument shall stop execution of the pass action for vacuous and nonvacuous success of all the specified assertions. Execution of the pass action for both vacuous and nonvacuous successes can be re-enabled subsequently by `$assertcontrol` with a `control_type` value of 6 (PassOn), while the execution of the pass action for only nonvacuous successes can be enabled subsequently by `$assertcontrol` with a `control_type` value of 10 (NonvacuousOn). An assertion that is already executing, including execution of the pass or fail action, is not affected. By default, the pass action is executed.

— **FailOn:** A value of 8 for this argument shall enable execution of the fail action of all the specified assertions. An assertion that is already executing, including execution of the pass or fail action, is not affected. This task also affects the execution of the default fail action block.

— **FailOff:** A value of 9 for this argument shall stop execution of the fail action of all the specified assertions until a subsequent `$assertcontrol` with a `control_type` value of 8 (FailOn). An assertion that is already executing, including execution of the pass or fail action, is not affected. By default, the fail action is executed. This task also affects the execution of default fail action block, i.e., `$error`, which is called in case no **else** clause is specified for the assertion.

— **NonvacuousOn:** A value of 10 for this argument shall enable execution of the pass action of all the specified assertions on nonvacuous success. An assertion that is already executing, including execution of the pass or fail action, is not affected. Refer to 16.15.8 for the definition of vacuous success.

— `VacuousOff`: A value of 11 for this argument shall stop execution of the pass action of all the specified assertions on vacuous success until a subsequent `$assertcontrol` with a `control_type` value of 6 (`PassOn`). An assertion that is already executing, including execution of the pass or fail action, is not affected. By default, the pass action is executed on vacuous success. Refer to 16.15.8 for the definition of vacuous success.

The assertion action control tasks or `$assertcontrol` with `control_type` values of 6 (`PassOn`) to 11 (`VacuousOff`) do not affect statistics counters for the assertions.

The details related to the behavior of `$assertkill`, `$assertcontrol` and `$assertoff` for assertions referring to global clocking future sampled value functions are explained in 16.9.4.

~~When invoked with no arguments, the system task shall apply to all assertions. When the task is specified with arguments, the fourth argument indicates levels of the hierarchy, consistent with the corresponding argument to the `$dumpvars` system task (see 21.7.1.2). Subsequent arguments, specify which scopes of the model to control. These arguments can specify entire modules or individual assertions.~~

The `$assertcontrol` system task provides finer grain assertion selection controls than the `$asserton`, `$assertoff` and `$assertkill` system tasks. The `$asserton`, `$assertoff` and `$assertkill` system tasks are provided for convenience and backward compatibility. They can be defined as:

**Note to the Editor: Pick the following three bullets if 3206 is approved:**

- `$asserton[(levels[, list])]` is equivalent to `$assertcontrol(3, 15, 7, levels [,list])`
- `$assertoff[(levels[, list])]` is equivalent to `$assertcontrol(4, 15, 7, levels [,list])`
- `$assertkill[(levels[, list])]` is equivalent to `$assertcontrol(5, 15, 7, levels [,list])`

**Note to the Editor: Pick the following three bullets if 3206 is not approved:**

- `$asserton[(levels[, list])]` is equivalent to `$assertcontrol(3, 7, 7, levels [,list])`
- `$assertoff[(levels[, list])]` is equivalent to `$assertcontrol(4, 7, 7, levels [,list])`
- `$assertkill[(levels[, list])]` is equivalent to `$assertcontrol(5, 7, 7, levels [,list])`

Similarly, assertion action control tasks `$assertpasson`, `$assertpassoff`, `$assertfailon`, `$assertfailoff`, `$assertvacuousoff` and `$assertnonvacuouson` are provided for convenience and backward compatibility. These tasks can be defined as:

**Note to the Editor: Pick the following six bullets if 3206 is approved:**

- `$assertpasson[(levels[, list])]` is equivalent to `$assertcontrol(6, 31, 7, levels [,list])`
- `$assertpassoff[(levels[, list])]` is equivalent to `$assertcontrol(7, 31, 7, levels [,list])`
- `$assertfailon[(levels[, list])]` is equivalent to `$assertcontrol(8, 31, 7, levels [,list])`
- `$assertfailoff[(levels[, list])]` is equivalent to `$assertcontrol(9, 31, 7, levels [,list])`
- `$assertnonvacuouson[(levels[, list])]` is equivalent to `$assertcontrol(10, 31, 7, levels [,list])`

— \$assertvacuousoff[(levels[, list])] is equivalent to \$assertcontrol(11, 31, 7, levels [,list])

**Note to the Editor: Pick the following six bullets if 3206 is not approved:**

— \$assertpasson[(levels[, list])] is equivalent to \$assertcontrol(6, 15, 7, levels [,list])

— \$assertpassoff[(levels[, list])] is equivalent to \$assertcontrol(7, 15, 7, levels [,list])

— \$assertfailon[(levels[, list])] is equivalent to \$assertcontrol(8, 15, 7, levels [,list])

— \$assertfailoff[(levels[, list])] is equivalent to \$assertcontrol(9, 15, 7, levels [,list])

— \$assertnonvacuouson[(levels[, list])] is equivalent to \$assertcontrol(10, 15, 7, levels [,list])

— \$assertvacuousoff[(levels[, list])] is equivalent to \$assertcontrol(11, 15, 7, levels [,list])

In the following example assertion control tasks are used to control the directive behavior.

```
module test;
logic clk;
logic a, b;
logic c, d;

// now define lets to make the code more readable
let LOCK = 1;
let UNLOCK = 2;
let ON = 3;
let OFF = 4;
let KILL = 5;

let CONCURRENT = 1;
let S_IMMEDIATE = 2; // simple immediate
```

**Note to the Editor: Pick the following two lines if 3206 is approved:**

```
let D_IMMEDIATE = 12; // Final and Observed deferred immediate
let EXPECT = 16;
```

**Note to the Editor: Pick the following two lines if 3206 is not approved:**

```
let D_IMMEDIATE = 4; // deferred immediate
let EXPECT = 8;

let ASSERT = 1;
let COVER = 2;
let ASSUME = 4;

let ALL_DIRECTIVES = (ASSERT|COVER|ASSUME);
let ALL_ASSERTS = (CONCURRENT|S_IMMEDIATE|D_IMMEDIATE|EXPECT);

let VACUOUSOFF = 11;
```

```

a1: assert property (@(posedge clk) a | => b)$info("assert passed"); else
$error("assert failed");
c1: cover property (@(posedge clk) a ##1 b);

always @(posedge clk)
begin
    ia1: assert (a);
end

always_comb
begin
    if (c)
        df1: assert #0 (d);
end

initial
begin
    // the following systasks affect the whole design so no modules
    // are specified

    // disable vacuous pass action for all the concurrent asserts,
    // covers and assumes in the design. Also disable vacuous pass
    // action for expect statements.
    $assertcontrol(VACUOUSOFF, CONCURRENT | EXPECT);

    // disable concurrent and immediate asserts and covers.
    // The following systask does not affect expect
    // statements as control type is Off.

    $assertcontrol(OFF); // using default values of all the
                        // arguments after first argument
    // After 20 time units, enable assertions.
    // explicitly specifying second, third and fourth arguments
    // in the following task call
    #20 $assertcontrol(ON, CONCURRENT|S_IMMEDIATE|D_IMMEDIATE,
    ASSERT|COVER|ASSUME, 0);

    // kill currently executing concurrent assertions after
    // 100 time units but do not kill concurrent covers/assumes
    // and immediate/deferred asserts/covers/assumes

    // using appropriate values of second and third arguments
    #100 $assertcontrol(KILL, CONCURRENT, ASSERT, 0);

    // the following assertion control task does not have any effect as
    // directive_type is assert but it has selected cover directive c1
    #10 $assertcontrol(ON, CONCURRENT|S_IMMEDIATE|D_IMMEDIATE, ASSERT, 0,
    c1);

    // now, after 10 time units, enable all the assertions except a1.
    // To accomplish this, first we'll lock a1 and then we'll enable all
    // the assertions and then unlock a1 as we want future assertion
    // control tasks to affect a1.

    #10 $assertcontrol(LOCK, ALL_ASSERTS, ALL_DIRECTIVES, 0, a1);
    $assertcontrol(ON); // enable all the assertions except a1

```

```
$assertcontrol(UNLOCK, ALL_ASSERTS, ALL_DIRECTIVES, 0, a1);
```

```
end
endmodule
```

~~Table 20-5~~ Table 20-8 lists the VPI callbacks (see 36.9.2 and 39.4) corresponding to the assertion control system task's invocation.

Table 20-5, change:

**Table 20-~~58~~—VPI callbacks for assertion control tasks**

Task	No arguments – assertion system callback (see 39.4.1)	With arguments – assertion callback (see 39.4.2)
<del>\$asserton</del> \$asserton	cbAssertionSysOn	cbAssertionEnable
<del>\$assertoff</del> \$assertoff	cbAssertionSysOff	cbAssertionDisable
<del>\$assertkill</del> \$assertkill	cbAssertionSysKill	cbAssertionReset + cbAssertionDisable
\$assertpasson	cbAssertionSysEnablePassAction	cbAssertionEnablePassAction
\$assertfailon	cbAssertionSysEnableFailAction	cbAssertionEnableFailAction
\$assertpassoff	cbAssertionSysDisablePassAction	cbAssertionDisablePassAction
\$assertfailoff	cbAssertionSysDisableFailAction	cbAssertionDisableFailAction
\$assertnonvacuouson	cbAssertionSysEnableNonvacuousAction	cbAssertionEnableNonvacuousAction
\$assertvacuousoff	cbAssertionSysDisableVacuousAction	cbAssertionDisableVacuousAction
\$assertcontrol with control_type 1 (lock)	cbAssertionSysLock	cbAssertionLock
\$assertcontrol with control_type 2 (unlock)	cbAssertionSysUnlock	cbAssertionUnlock
\$assertcontrol with control_type 3 (On)	cbAssertionSysOn	cbAssertionEnable
\$assertcontrol with control_type 4 (Off)	cbAssertionSysOff	cbAssertionDisable
\$assertcontrol with control_type 5 (Kill)	cbAssertionSysKill	cbAssertionReset + cbAssertionDisable
\$assertcontrol with control_type 6 (PassOn)	cbAssertionSysEnablePassAction	cbAssertionEnablePassAction
\$assertcontrol with control_type 8 (FailOn)	cbAssertionSysEnableFailAction	cbAssertionEnableFailAction
\$assertcontrol with control_type 7 (PassOff)	cbAssertionSysDisablePassAction	cbAssertionDisablePassAction
\$assertcontrol with control_type 9 (FailOff)	cbAssertionSysDisableFailAction	cbAssertionDisableFailAction
\$assertcontrol with control_type 10 (NonvacuousOn)	cbAssertionSysEnableNonvacuousAction	cbAssertionEnableNonvacuousAction
\$assertcontrol with control_type 11 (VacuousOff)	cbAssertionSysDisableVacuousAction	cbAssertionDisableVacuousAction

Clause 20, change:

**Assertion control tasks (20.11)**

```
$asserton                      $assertoff
```

\$assertkill

### Assertion action control tasks (20.12)

\$assertpasson \$assertpassoff  
\$assertfailon \$assertfailoff  
\$assertnonvacuouson  
\$assertvacuousoff

TO:

### Assertion control tasks (20.11)

\$asserton \$assertoff  
\$assertkill \$assertcontrol  
\$assertpasson \$assertpassoff  
\$assertfailon \$assertfailoff  
\$assertnonvacuouson \$assertvacuousoff

### ~~Assertion action control tasks (20.12)~~

~~\$assertpasson \$assertpassoff  
\$assertfailon \$assertfailoff  
\$assertnonvacuouson  
\$assertvacuousoff~~

*Note to the Editor: Clause 20.12 is getting deleted, so clause numbers have to be adjusted accordingly.*

### ~~20.12 Assertion action control system tasks~~

~~assert\_action\_control\_task ::=  
assert\_action\_task [ ( levels [ , list\_of\_modules\_or\_assertions ] ) ] ;  
assert\_action\_task ::=  
\$assertpasson  
| \$assertpassoff  
| \$assertfailon  
| \$assertfailoff  
| \$assertnonvacuouson  
| \$assertvacuousoff  
list\_of\_modules\_or\_assertions ::=  
module\_or\_assertion { , module\_or\_assertion }  
module\_or\_assertion ::=  
module\_identifier  
| assertion\_identifier  
| hierarchical\_identifier~~

### ~~Syntax 20-12—Assertion action control syntax (not in Annex A)~~

~~SystemVerilog provides the following six system tasks to control the execution of assertion action blocks that are associated with assertions and the expect statement:~~

~~—\$assertpassoff shall stop execution of the pass action for vacuous and nonvacuous success of all the specified assertions. Execution of the pass action for both vacuous and nonvacuous successes can be re-enabled subsequently by \$assertpasson, while the execution of pass action for only nonvacuous successes can be enabled subsequently by \$assertnonvacuouson. An assertion that is already executing, including execution of the pass or fail action, is not affected. By default, the pass action is executed.~~

~~—\$assertfailoff shall stop execution of the fail action of all the specified assertions until a subsequent \$assertfailon. An assertion that is already executing, including execution of the pass or fail action, is not affected. By default, the fail action is executed. This task also affects the execution of default fail action block.~~

~~—\$assertvacuousoff shall stop execution of the pass action of all the specified assertions on vacuous success until a subsequent \$assertpasson. An assertion that is already executing, including execution of the pass or fail action, is not affected. By default, the pass action is executed on vacuous success. Refer to 16.15.8 for the definition of vacuous success.~~

~~—\$assertpasson shall enable execution of the pass action for vacuous and nonvacuoussuccess of all the specified assertions. An assertion that is already executing, including execution of the pass or fail action, is not affected.~~

~~—\$assertfailon shall enable execution of the fail action of all the specified assertions. An assertion that is already executing, including execution of the pass or fail action, is not affected. This task also affects the execution of the default fail action block.~~

~~—\$assertnonvacuouson shall enable execution of the pass action of all the specified assertions on nonvacuous success. An assertion that is already executing, including execution of the pass or fail action, is not affected. Refer to 16.15.8 for the definition of vacuous success.~~

~~The details related to the behavior of \$assertpassoff, \$assertfailoff, and \$assertvacuousoff for assertions referring to global clocking sampled future value functions are explained in 16.9.4. When invoked with no arguments, the system task shall apply to all the assertions. When the system task is specified with arguments, the first argument indicates levels of the hierarchy, consistent with the corresponding~~

~~argument to the \$dumpvars system task (see 21.7.1.2). Subsequent arguments specify which scopes of the model to control. These arguments can specify entire scopes (module, program, interface, always procedure, or initial procedure) or individual assertions.~~

~~These system tasks shall not affect the execution of pass or fail actions until the system task is executed.~~

~~These system tasks shall not affect the statistics counters for the assertions.~~

~~Table 20-6 lists the VPI callbacks (see 36.9.2 and 39.4) corresponding to the assertion action control system tasks.~~

~~Table 20-6—VPI callbacks for assertion action control tasks~~

<del>Task</del>	<del>No arguments—assertion system callback (see 39.4.1)</del>	<del>With arguments—assertion callback (see 39.4.2)</del>
<del>\$assertpasson</del>	<del>ebAssertionSysEnablePassAction</del>	<del>ebAssertionEnablePassAction</del>
<del>\$assertfailon</del>	<del>ebAssertionSysEnableFailAction</del>	<del>ebAssertionEnableFailAction</del>
<del>\$assertpassoff</del>	<del>ebAssertionSysDisablePassAction</del>	<del>ebAssertionDisablePassAction</del>
<del>\$assertfailoff</del>	<del>ebAssertionSysDisableFailAction</del>	<del>ebAssertionDisableFailAction</del>
<del>\$assertnonvacuouson</del>	<del>ebAssertionSysEnableNonvacuousAction</del>	<del>ebAssertionEnableNonvacuousAction</del>
<del>\$assertvacuousoff</del>	<del>ebAssertionSysDisableVacuousAction</del>	<del>ebAssertionDisableVacuousAction</del>

*Clause 16.9.4, change:*

The behavior of **disable iff** and other asynchronous assertion related controls such as \$assertkill (see 20.11 and 20.12) is with respect to the interval of the evaluation attempt defined above. If, for example, \$assertkill is executed in a time step strictly after the last tick of the assertion clock for the evaluation attempt, then it shall not affect that attempt, even if \$assertkill is executed no later than the next global clocking tick.

TO:

The behavior of **disable iff** and other asynchronous assertion related controls such as ~~\$assertkill~~ \$assertcontrol (see 20.11 and 20.12) is with respect to the interval of the evaluation attempt defined above. If, for example, ~~\$assertkill~~ \$assertcontrol with control\_type 5 (Kill) is executed in a time step strictly after the last tick of the assertion clock for the evaluation attempt, then it shall not affect that attempt, even if ~~\$assertkill~~ \$assertcontrol is executed no later than the next global clocking tick.

*Clause 16.3, change:*

The immediate **assert** statement specifies that its expression is required to hold. Failure of an immediate **assert** statement indicates a violation of the requirement and thus a potential error in the design. If an **assert** statement fails and no **else** clause is specified, the tool shall, by default, call `$error`, unless `$assertfailoff` is used to suppress the failure.

TO:

The immediate **assert** statement specifies that its expression is required to hold. Failure of an immediate **assert** statement indicates a violation of the requirement and thus a potential error in the design. If an **assert** statement fails and no **else** clause is specified, the tool shall, by default, call `$error`, unless `$assertfailoff` `$assertcontrol` with `control_type 9 (FailOff)` is used to suppress the failure.

*Clause 39.4.1, change:*

- **cbAssertionSysInitialized**. This callback occurs after the system has initialized. No assertionspecific actions can be performed until this callback completes. The assertion system can initialize before `cbStartOfSimulation` does or afterwards.
- **cbAssertionSysOn**. The assertion system has become active and starts processing assertion attempts. This always occurs after **cbAssertionSysInitialized**. By default, the assertion system is “started” on simulation startup, but the user can delay this by using assertion system control actions.

TO:

- **cbAssertionSysInitialized**. This callback occurs after the system has initialized. No assertionspecific actions can be performed until this callback completes. The assertion system can initialize before `cbStartOfSimulation` does or afterwards.
- **cbAssertionSysLock**. This callback occurs when the assertion system is locked, e.g., due to a system control action.
- **cbAssertionSysUnlock**. This callback occurs when the assertion system is unlocked, e.g., due to a system control action.
- **cbAssertionSysOn**. The assertion system has become active and starts processing assertion attempts. This always occurs after **cbAssertionSysInitialized**. By default, the assertion system is “started” on simulation startup, but the user can delay this by using assertion system control actions.

*Change Annex M.2:*

```
#define cbAssertionSysOn 616
#define cbAssertionSysOff 617
#define cbAssertionSysKill 631
```

TO:

```
#define cbAssertionSysOn 616
#define cbAssertionSysOff 617
#define cbAssertionSysKill 631
#define cbAssertionSysLock 659
#define cbAssertionSysUnlock 660
```

*Clause 39.4.2, change:*

- **cbAssertionDisable**. The assertion is disabled (e.g., as a result of a control action, see 39.5.2).
- **cbAssertionEnable**. The assertion is enabled (e.g., as a result of a control action, see 39.5.2).

TO:

- **cbAssertionLock**. The assertion is locked (e.g., as a result of a control action, see 39.5.2).
- **cbAssertionUnlock**. The assertion is unlocked (e.g., as a result of a control action, see 39.5.2).
- **cbAssertionDisable**. The assertion is disabled (e.g., as a result of a control action, see 39.5.2).
- **cbAssertionEnable**. The assertion is enabled (e.g., as a result of a control action, see 39.5.2).

### *Change*

```
#define cbAssertionDisable 611
#define cbAssertionEnable 612
#define cbAssertionReset 613
```

### *TO:*

```
#define cbAssertionLock 661
#define cbAssertionUnlock 662
#define cbAssertionDisable 611
#define cbAssertionEnable 612
#define cbAssertionReset 613
```

### *Change*

```
#define vpiAssertionDisable 620
#define vpiAssertionEnable 621
#define vpiAssertionReset 622
```

### *TO:*

```
#define vpiAssertionLock 645
#define vpiAssertionUnlock 646
#define vpiAssertionDisable 620
#define vpiAssertionEnable 621
#define vpiAssertionReset 622
```

### *Change*

```
#define vpiAssertionSysOn 627
#define vpiAssertionSysOff 628
```

### *TO:*

```
#define vpiAssertionSysLock 647
#define vpiAssertionSysUnlock 648
#define vpiAssertionSysOn 627
#define vpiAssertionSysOff 628
```

### *Change*

The `t_vpi_attempt_info` attempt information structure contains details relevant to the specific event that occurred.

— On disable, enable, reset, kill, pass action, fail action, vacuous action, and nonvacuous action callbacks, the returned `p_vpi_attempt_info` info pointer is NULL, and no attempt information is available.

### *TO:*

The `t_vpi_attempt_info` attempt information structure contains details relevant to the specific event that occurred.

— On **lock**, **unlock**, disable, enable, reset, kill, pass action, fail action, vacuous action, and nonvacuous action callbacks, the returned `p_vpi_attempt_info` info pointer is NULL, and no attempt information is

available.

*Clause 39.5.1, change:*

- Usage example: `vpi_control(vpiAssertionSysOff, handle)`
- **vpiAssertionSysOff** disables any further assertions from being started. Assertions already executing are not affected. This control has no effect on pre-existing assertion callbacks.
- **vpiAssertionSysKill** discards all attempts in progress and disables any further assertions from being started. This control has no effect on pre-existing assertion callbacks.

*TO:*

- Usage example: `vpi_control(vpiAssertionSysOff, handle)`
- **vpiAssertionSysOff** disables any further assertions from being started. Assertions already executing are not affected. This control has no effect on pre-existing assertion callbacks.
- **vpiAssertionSysKill** discards all attempts in progress and disables any further assertions from being started. This control has no effect on pre-existing assertion callbacks.
- Usage example: `vpi_control(vpiAssertionSysLock, handle)`
- **vpiAssertionSysLock** locks the assertions from being changed. The status of the assertions can not be changed without unlocking it.
- Usage example: `vpi_control(vpiAssertionSysUnlock, handle)`
- **vpiAssertionSysUnlock** unlocks the assertions. Now, the status of the assertion can be changed.

*Clause 39.5.2, change:*

- Usage example: `vpi_control(vpiAssertionDisable, assertionHandle)`
- **vpiAssertionDisable** disables the starting of any new attempts for this assertion. This has no effect on any existing attempts or if the assertion is already disabled. By default, all assertions are enabled.

*TO:*

- Usage example: `vpi_control(vpiAssertionLock, assertionHandle)`
- **vpiAssertionLock** locks the status of the assertion from being changed. The status of the assertion cannot be changed without unlocking it.
- Usage example: `vpi_control(vpiAssertionUnlock, assertionHandle)`
- **vpiAssertionUnlock** unlocks the assertion. Now the status of the assertion can be changed.
- Usage example: `vpi_control(vpiAssertionDisable, assertionHandle)`
- **vpiAssertionDisable** disables the starting of any new attempts for this assertion. This has no effect on any existing attempts or if the assertion is already disabled. By default, all assertions are enabled.