OpenVera Assertions Quick Reference

Comments?
E-mail your comments about VCS documentation to
vcs_support@synopsys.com

SYNOPSYS®

Copyright Notice and Proprietary Information

ExpressModel, Formal Model Checker, FoundryModel, Frame Compiler, Galaxy, Gatran, HANEX, HDL Advisor, HDL Compiler, Hercules, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSIM^{plus}, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Milkyway, ModelSource, Module Compiler, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Raphael, Raphael-NES, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, Softwire, Source-Level Design, Star-RCXT, Star-SimXT, Taurus, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

Contents

Guide Conventions

This guide is a convenient quick reference for OpenVera Assertions (OVA), a high-level assertions language. For additional information, see the *OpenVera Assertions Language Reference Manual*.

The following conventions are used in this guide:

Convention	Description
Courier	Indicates syntax.
Courier italic	Indicates a user-defined value such as *object_name*.
Courier bold	Indicates keywords.
[ ]	Denotes optional parameters, such as pin1 [pin2 ... pinN]
|	Indicates a choice among alternatives, such as low | medium | high. (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)

Section 1: Getting Started

This chapter shows an example .ova file, and compile and runtime options for using OVA with Synopsys verification tools.

Temporal Assertion File Example

Defining the Unit and Assertion

Define a unit with expressions and assertions (or select one from the Checker Library).

```
unit 4step
  #(parameter integer s0 = 0)// Define parameters
  (logic en, logic clk,     // Define ports
  logic [7:0] result);

  // Define a clock to synchronize attempts:
  clock posedge (clk)
  {
    // Define expressions:
    event t_0 : (result == s0);
    event t_1 : (result == 6);
    event t_2 : (result == 9);
    event t_3 : (result == 3);

    event t_normal_s:
      // Define a precondition to limit reports:
      if (en) then
        (t_0 #1 t_1 #1 t_2 #1 t_3);
  }

  // Define an assertion:
  assert c_normal_s : check(t_normal_s,
    "Missed a step.");

endunit
```

Binding the Unit

Bind the unit to one or more instances in the design.

```
// bind module cnt : // All instances of cnt or
bind instances cnt_top.dut : // one instance.
  4step start_4      // Name the unit instance.
  #(4)               // Specify parameters.
  (start, m_clk, outp);// Specify ports.
```

Inlining the Unit

Or add the unit directly to the design as an inline pragma:

```
/* ova 4step start_4 // Name the unit instance.
    #(4)             // Specify parameters.
    (start, m_clk, outp); // Specify ports.
*/
```

OVA Compile Options

Option	Description
-ova_cov	Required to view results with functional coverage.
-ova_cov_events	Enables coverage reporting of expressions.
-ova_cov_hier *filename*	Limits functional coverage to the module instances specified in *filename*. Specify the instances using the same format as VCM. If this option is not used, coverage is implemented on the whole design.
-ova_debug | -ova_debug_vpd	Required to view results with VirSim.
-ova_file *filename*	Identifies *filename* as an assertion file. Not required if the file name ends with .ova. For multiple assertion files, repeat this option with each file.
ova_filter_past	Ignores assertion subsequences containing past operators that have not yet eclipsed the history threshold.
-ova_dir *pathname*	Specifies an alternative name and location for the Verification Database directory.
-ova_enable_diag	Enables further control of result reporting with runtime options. See Table , "Runtime Options for Reporting," on page 11.
-ova_inline	Enables compiling of OVA code that is written inline with a Verilog design.
-vera	Required if using assertions with a Vera testbench that uses static signal binding.
-vera_dbind	Required if using assertions with a Vera testbench that uses dynamic signal binding.

Runtime Options That Are Always Available

Option	Description
-ova_quiet [1]	Disables printing results on screen. The report file is not affected. With the 1 argument, only a summary is printed on screen.
-ova_report [*filename*]	Generates a report file in addition to printing results on screen. Specifying the full path name of the report file overrides the default report name and location.
-ova_verbose	Adds more information to the end of the report including assertions that never triggered and attempts that did not finish, and a summary with the number of assertions present, attempted, and failed.

Runtime Options for Reporting

Enabled only if compiled with -ova_enable_diag.

Option	Description
-ova_filter	Blocks reporting of trivial if-then successes. These happen when an if-then construct registers a success only because the if portion is false (and so the then portion is not checked). With this option, reporting only shows successes in which the whole expression matched.
-ova_max_fail *N*	Limits the number of failures for each assertion to *N*. When the limit is reached, the assertion is disabled. *N* must be supplied, otherwise no limit is set.
-ova_max_success *N*	Limits the total number of reported successes to *N*. *N* must be supplied, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.
-ova_simend_max_fail *N*	Terminates the simulation if the number of failures for any assertion reaches *N*. *N* must be supplied, otherwise no limit is set.
-ova_success	Enables reporting of successful matches in addition to failures. The default is to report only failures.

Runtime Options for Functional Coverage

Enabled only if compiled with -ova_cov.

Option	Description
-ova_cov	Enables functional coverage reporting.
-ova_cov_name *filename*	Specifies the file name or the full path name of the functional coverage report file. This option overrides the default report name and location. If only a file name is given, the default location is used resulting in: ./*simv*.vdb/fcov/*filename*.db.
-ova_cov_db *executablename.db*	Specifies the path name of the initial coverage file. The initial coverage file is needed to set up the data base. By default, an empty coverage file is loaded from simv.vdb/snps/fcov/*executablename*.db.

Section 2: Defining Simple Expressions and Assertions

This chapter describes the language for expressing simple timing relationships between design objects. Using this language, you can specify one or more expressions, their functional and timing relationships, and a set of criteria for the relationships to fail or succeed.

Specifying Edge Events and Clocks

Edge Events

```
posedge | negedge | edge bit_vector_expr
matched event_name
ended event_name
```

Clocks

```
clock edge_expr
{
  statements other than clock
}
```

Specifying Time Shift Relationships

```
# int | [int .. int] | [int ..]
->>
```

Defining Expressions

```
bool name [(param1, ..., paramN)] : boolean_expr;
event name [[index]] [(param1, ..., paramN)] :
  sequence_expr ;
```

Specifying Temporal Assertions

```
assert [name] [[index]] : check | forbid
  (sequence_expr | event_name [, message]);
```

Packing Assertions for Use

Declaring Units

```
unit name
[#(parameter_list1; ...; parameter_listN)]
(port1, ..., portN);
  assert, clock, and bool statements
  template and unit instances
endunit
```

```
parameter_list:
parameter [integer | real | string]
  name1 = const_expr1, ..., nameN = const_exprN
port:
logic [[msb:lsb]] name [[i:j]...]
```

Instantiating Units

```
unit_name [instance_name]
  [#(parameter1, ..., parameterN)]
  [(port1, ..., portN)];
```

Binding Units to the Design

```
bind module module_name : unit_instance;
bind instances instance1, ..., instanceN :
  unit_instance;
```

Name Resolution

v'name to denote a design variable

Constructing Complex Sequences

This chapter describes the language for combining expressions into complex sequences. This chapter also describes adding variables to expressions and combining expressions and assertions into reusable libraries.

Specifying Composite Sequences

Logically ANDing Sequences

sequence_expr && *sequence_expr*

Logically ORing Sequences

sequence_expr || *sequence_expr*

Specifying Conditional Sequence Matching

if *boolean_expr* then *sequence_expr*
  [else *sequence_expr*]

Matching Repetition of Sequences

sequence_expr * **[**int**]** | **[**int .. int**]** | **[**int .. **]**

Specifying Conditions Over Sequences

cond_spec1, ..., *cond_specN* in *sequence_expr*

With *cond_spec* being either of:
  • istrue *boolean_expr*
  • length **[**int**]** | **[**int .. int**]** | **[**int .. **]**

Specifying an Unconditional True

any

Manipulating and Checking Data

var [**[**int:int**]**] *name* = initial_const_value;

or

init *var_name_ref* = *initial_const_value*;

var_name_ref <= *bit_vector_expr*;

past(*name* [, *number_of_ticks*])

count(*bit_vector_expr*)

Grouping Assertions as a Library

```
template name [(formal_param1, ...,
formal_paramN)] :
{
  template_body
}
```

With *formal_param* being:
```
name [= boolean_expr | sequence_expr]
```

A *template* is instantiated with the following syntax:
```
name [ins_name] [(actual_param1, ...,
actual_paramN)];
```

For Loops

```
for (name = expr; term_expr; name = incr_expr)
{
  for_loop_body
}
```

With *term_expr* being:
```
name op1 expr
| expr op1 name
```

The *op1* operator can be: ==, !=, >, >=, <, or <=.

With *incr_expr* being:
```
name op2 expr
```

The *op2* operator can be: + or -.

Building Expressions Iteratively

```
expr [index] : sequence_expr ;
```

Inlining OVA in Verilog

```
/* ova unit_name [instance_name]
  [#(parameter1, ..., parameterN)]
  [(port1, ..., portN)];
*/
```

Section 4: Checker Library

The OpenVera Assertions Checker Library is a collection of temporal expressions and assertions for a variety of commonly needed tests. Using these checkers will speed coding of your own temporal assertions.

Conventions

All of the checkers described in this chapter are available in both unit and template form. The two forms have identical functions. Use the unit form to bind checkers to a design. Use the template form to build more complex checkers inside your own units.

A few of the arguments are common to almost all of the checkers.

en

  If 1, enables the start of a check. Default = 1.

edge_expr

  The active edge for the *clk* signal in unit syntax. Use the following values to specify the edge type:

  • posedge: 0 (the default)
  • negedge: 1
  • edge: 2

clk

  The clock signal on which inputs are sampled and the checks are performed.

msg

  The message reported if the assertion fails. Default = "assertion triggered".

severity

  Specifies the severity level of the assertion (default is 0). This parameter can be used to group assertions used for a similar purpose, and provide a selection/filtering mechanism to enable/disable individual or groups of assertions.

category

  Specifies the category of the assertion (default is 0). This parameter can be used to group assertions used for a similar purpose, and provide a selection/filtering mechanism to enable/disable individual or groups of assertions.

In the unit form, all parameters are integers except for *msg*, which is a quoted string. All ports are the logic type. Port widths are 1 bit unless otherwise indicated.

Coverage Properties

The checkers contain OVA assertions used in verification of the intended behavior (the original purpose of the checkers), and a large number of them also contain coverage properties that can be used to

detect the occurrence of events related to the behavior, in particular such as triggering conditions and corner cases.

The coverage can be controlled globally by a macro symbol OVA_COVER_ON and locally on a per-instance basis using a parameter coverage_level.

Shared Syntax

Macro Symbols

The way checkers are used, whether for checking by assertions or coverage gathering or both, can be selected using two global 'define symbols:

OVA_ASSERT_OFF: When the symbol is **defined**, all assertions are **removed** from the checker. That is, when left undefined the behavior is backward compatible with earlier versions of the library.

OVA_COVER_ON: When the symbol is **defined**, the cover statements in the checkers are **included**.

Parameters and Ports

The following are shared ports and parameters of all the checkers.

en: Used as a guard expression (port), this expression enables the start of a check. Default = 1 (if *en* is not specified, it defaults to true).

en : Specifies the active edge for the clock signal (clk) in unit syntax. Use the following parameter values to specify the edge type:
   • posedge: 0 (the default)
   • negedge: 1
   • edge: 2

clk: Specifies the clock signal (port) on which inputs are sampled and the checks are performed.

msg: severity: Specifies the severity level (parameter) of the assertion, default is 0. This parameter can be used to group assertions used for a similar purpose, and provide a selection/filtering mechanism to enable/disable individual or groups of assertions.

category: Specifies the category of the assertion, default is 0. This parameter can be used to group assertions used for a similar purpose, and provide a selection/filtering mechanism to enable/disable individual or groups of assertions.

   • All standard OVA checkers described in this chapter have *severity*, *category*, and coverage_level (if present at all) parameters as the last items on the unit parameter list and the template argument list. Note that coverage properties are not available in the template form of the checkers.

coverage_level: Specifies which coverage levels should be enabled (provided that the symbol COVER_ON is defined.) The following levels are supported (default is 2):

Level 1: Basic coverage, implemented using cover statements. Used by simulation and Magellan.

Level 1 is enabled by setting bit 0 of coverage_level to 1. Example: The number of Enqueues and Dequeues in a FIFO.

Level 2: This level is intended mainly for data coverage using cover groups in System Verilog. Since OVA does not support such constructs this level is absent in all but a few checkers where it is implemented using cover (property) statements.

Level 2 is enabled setting bit 1 of coverage_level to 1. **This is the default level selected by this parameter (for compatibility with SVA checkers.)**

Example: Individual bits in a vector asserted at-least once.

Level 3: Mostly cover statements for specific corner points as specified by parameters of the checker. Used primarily by formal tools as goals, but can be enabled in simulation too. These coverage items ensure that the corner case condition of the RTL/design block are verified during testing.

Level 3 is enabled setting bit 2 of coverage_level to 1.

Examples: The number of times FIFO reached HIGH water mark. The number of times ACK was received at the next clock after REQ was issued. The number of times the specified Min latency value was reached.

Checkers

---

ova_arbiter

Ensures that a resource arbiter provides grants to corresponding requests between the specified minimum and maximum number of clock cycles between a request and a grant.

Unit Syntax:

```
ova_arbiter
#(no_chnl, bw_prio, grant_one_chk, fairness_chk,
priority_chk, fifo_chk, min_lat, max_lat,
edge_expr, msg, severity, category,
coverage_level)
instance_name (en, clk, requests, priority,
grants);
```

Template Syntax:

```
arbiter(en, clk, requests, priority, bw_prio,
grants, no_chnl, grant_one_chk, fairness_chk,
priority_chk, fifo_chk, min_lat, max_lat, msg,
severity, category);
```

Arguments

no_chnl: The number of channels (bits) in requests and grants. Default = 2.

bw_prio: The number of bits in *priority* values. Default = 1.

grant_one_chk: If 1, checks that only one grant is issued per clock cycle. Default = 1.

fairness_chk, req_priority_chk, and fifo_chk :

Indicate which arbitration scheme is to be verified. These must be compile-time constants. If no arbitration scheme is asserted, no checks will be performed to verify the arbitration scheme.

fairness_chk: If true, then this unit will ensure that no channel will be issued more than one grant while other channels have requests pending except if this is the only request at the highest req_priority when *req_priority_chk* is asserted. Default = 0.

req_priority_chk: If true, then this checker will ensure that grants are issued according to the priority indicated in the *req_priority* vector.

If 0 (disabled), then req_priority is not taken into account in any of the other checks. However, the argument *req_priority* must still have the correct dimension even though the actual values do not matter (e.g., pass vector of 0's). The req_priority vector may be a design vector (i.e., not a constant array). However, while a request is being processed the req_priority should not change, otherwise certain checks may produce incorrect results (success or failure). Default = 0.

fifo_chk: If true, then this unit will ensure that grants are issued according to the order that their requests were received unless *req_priority_chk* is asserted which means that the fifo check is performed only on requests of the current highest req_priority. Default = 0.

min_lat: The minimum global grant latency. Default = 1.

If 0, then the grant is expected starting the same cycle as the request (i.e., combinational arbiter is possible with *max_lat* = 0). If priority arbitration check is enabled, then *min_lat* should be 0 or 1 only.

max_lat: If *max_lat* > 0, it specifies the maximum global grant latency regardless of the selection criterion. That is, a persistent request must be granted within <max_lat> clock cycles. The check is useful in systems where a request must be granted within a certain latency even in the presence of other requests.

If *max_lat* = 0, the global latency check is disabled. Default = 0.

requests: Requests signals as vectors.

Vector of size [no_chnl-1:0] where the bits correspond to the corresponding channels in reqs . req is assumed to be 1 when active.

priority
A [bw_prio*no_chnl-1 : 0] bitvector of bw_prio*no_chnl bits formed by concatenating non-negative integer req_priority values corresponding to the request lines. The right-most bw_prio bits in req_priority corresponds to channel 0, etc. The req_priority value 0 is the lowest req_priority.

For the assertions in the checker to operate correctly, the priority

assignments to the requests should remain constant over time. Otherwise, the assertions may report unwanted failures.

grants Grants signals as vectors. Vector of size [no_chnl-1:0] where the bits correspond to the corresponding channels in reqs . Assumed to be 1 when active.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_arbiter_req_granted counts the number of granted requests, for each channel

Cover cover_abandoned_req counts the number of abandoned request, for each channel

Level_3 (bit 2 set in coverage_level) : Cover cover_req_granted_exactly_after_min_lat indicates how many times the req-to-grant latency was exactly equal to the specified min_lat value.

Cover cover_req_granted_exactly_after_max_lat indicates how many times the req-to-grant latency was exactly equal to the specified max_lat value.

Coverage Level 1 enabled by default in the unit instance.

---

ova_arith_overflow

Checks that the value of a signal does not overflow the range of a specified target signal.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Unit Syntax:

```
ova_arith_overflow
#(target_bw, exp_bw, edge_expr, msg, severity, category)
instance_name (en, clk, target, exp);
```

Template Syntax:

```
arith_overflow(en, clk, target, target_bw, exp, exp_bw, msg,  severity, category);
```

Arguments

target_bw: Number of bits in the specified target signal (target). Default = 1.

target: Signal that receives the signal of interest (exp).

exp: Signal of interest whose value is compared against the target signal (target).

ova_asserted

Once the specified start expression (*start*) evaluates as true, this checker makes sure that the signal under test (*exp*) is asserted (1 or true) until the stop expression (*stop*) evaluates true.

Unit Syntax:

```
ova_asserted
#(delay, edge_expr, msg,  severity, category,
coverage_level)
instance_name (en, clk, exp, start, stop);
```

Template Syntax:

```
asserted(en, clk, exp, start, stop, delay, msg,
severity, category);
```

Arguments

delay: The number of clock cycles after the start signal (*start*) is true and before the signal under test (*exp*) is asserted. Default = 0.

exp: Signal being tested.

start: Signal that marks the start of the window.

stop: Signal that marks the end of the window.

Coverage Modes

Level_1 (bit 0 set in coverage_level): Cover cover_num_of_start_events indicates how many times start occurred.

Cover cover_num_of_matches indicates how many times exp remained true within the required interval from start plus delay to stop.

---

ova_bits

Checks that the value of the signal being tested (exp) falls between the specified minimum (*min*) and maximum (*max*) number of bits (inclusive) that are asserted or deasserted as indicated by the *deasserted* flag.

Unit Syntax:

```
ova_bits
#(min, max, deasserted, exp_bw, edge_expr, msg,
severity, category, coverage_level)
instance_name (en, clk, exp);
```

Template Syntax:

```
bits(en, clk, exp, min, max, deasserted, msg,
severity, category);
```

Arguments

min: Minimum number of bits asserted or deasserted. Default = 1.

max: Maximum number of bits asserted or deasserted. Default = 1.

deasserted: If 1, checks for deasserted (0) bits. If 0, checks for asserted (1) bits. Default = 0.

exp_bw: The number of bits in *exp*. Default = 2.

exp: Signal being tested.

Coverage Modes

Level_1 (bit 0 set in coverage_level): Cover cover_bits_exp_change indicates how many times exp changed value.

Cover cover_bits indicates how many times the required behavior was matched on a change of exp value.

Level_3 (bit 2 set in coverage_level): Cover cover_max_bits_asserted indicates how many times the max number of bits was (de)asserted on a change of exp value.

Cover cover_min_bits_asserted indicates how many times the min number of bits was (de)asserted on a change of exp value.

Covers cover_bits_asserted[i] indicate how many times bit exp[i] was (de)asserted on a change of exp value.

---

ova_check_bool

Verifies that the specified expression is always true.

```
ova_check_bool
#(edge_expr, msg, severity, category)
instance_name (expr, clk);
```

Template syntax:

```
check_bool(expr, msg,  severity, category, clk);
```

Arguments

expr: Signal being tested.

---

ova_code_distance

Checks that when the specified signal (*exp*) changes, the number of bits that are different compared to *exp2* fall within the specified minimum (*min*) and maximum (*max*) number of bits.

Unit Syntax:

```
ova_code_distance
#(min, max, bw, edge_expr, msg,  severity,
category, coverage_level)
instance_name (en, clk, exp, exp2);
```

Template Syntax:

```
code_distance(en, clk, exp, exp2, min, max, msg,
severity, category);
```

Arguments

min: The minimum number of bits that are different. Default = 1.

max: The maximum number of bits that are different. Default = 1.

bw: The number of bits in *exp* and *exp2*. Default = 2.

exp: Signal being tested.

exp2: Signal that the signal under test (*exp*) is compared to.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_exp_change indicates how many times exp value changed.

Cover cover_code_distance_match indicates how many time the code distance matched the requirements.

Level_3 (bit 2 set in coverage_level) : Cover cover_code_distance_eq_to_min indicates how many times the code distance was exactly min bits.

Cover cover_code_distance_eq_to_max indicates how many times the code distance was exactly max bits.

---

ova_const

Checks that the value of the signal being tested (*exp*) is always constant.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Unit Syntax:

```
ova_const
#(bw, edge_expr, msg,  severity, category)
instance_name (en, clk, exp);
```

Template Syntax:

```
const(en, clk, exp, msg,  severity, category);
```

Arguments

bw: The number of bits in the signal being tested (*exp*). Default = 1.

exp: Signal being tested.

---

ova_cover_bool

Collects information about when the mandatory argument *expr* is high, low, posedge or negedge as selected by the parameter *cover_kind* and sampled by the clock.

Unit Syntax:

```
ova_cover_bool
#(edge_expr, msg, cover_kind, severity, category)
instance_name (expr, clk)
```

Template Syntax:

```
cover_bool(expr, msg, clk, cover_kind, severity,
category);
```

Arguments

expr: Signal being tested.

```
cover_kind
```
Specifies value at which to test expr.
```
cover_kind
```

Expression covered

0	expr == 1'b0
1	expr == 1'b1 (default)
2	posedge expr
3	negedge expr

---

ova_data_used

Checks that data from the source signal (src[sleft:sright]) appears in the destination signal (dest[dleft:dright]) within the specified window.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Unit Syntax:
```
ova_data_used
#(sleft, sright, dleft, dright, start, finish,
edge_expr, msg, severity, category)
instance_name (en, clk, trigger, src, dest);
```

Template Syntax:
```
data_used(en, clk, trigger, src, sleft, sright,
dest, dleft, dright, start, finish, msg,
severity, category);
```

Arguments

sleft: The most significant bit of the source signal's (src) bit slice. Default = 1.

sright: The least significant bit of the source signal's (src) bit slice. Default = 0.

dleft: The most significant bit of the destination signal's (dest) bit slice. Default = 1.

dright: The least significant bit of the destination signal's (dest) bit slice. Default = 0.

start: The number of cycles after the trigger signal (trigger) asserts to start the window. Default = 1.

finish: The number of cycles after the trigger signal (trigger) asserts to stop the window. Default = 1.

trigger: Signal that is part of starting the window.

src: Source signal.

dest: Destination signal.

ova_deasserted

Once the start expression (*start*) evaluates true, this checker makes
sure that the signal being tested (*exp*) is deasserted (0 or false) until
the stop expression (*stop*) evaluates true (excluding the clock tick
when *stop* is true).

Unit Syntax:

```
ova_deasserted
#(delay, edge_expr, msg,  severity, category)
instance_name (en, clk, exp, start, stop);
```

Template Syntax:

```
deasserted(en, clk, exp, start, stop, delay, msg,
severity, category);
```

Arguments

delay: The number of clock cycles after the start signal (*start*)
   goes true before the signal being tested (*exp*) is deasserted. Default
   = 0.

exp: Signal being tested.

start: Signal that marks the start of the window.

stop: Signal that marks the end of the window.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
   cover_num_of_start_events indicates how many times start
   occurred.

   Cover cover_num_of_matches indicates how many times exp
   remained deasserted within the required interval from start plus
   delay to stop.

---

ova_dec

Checks that when the signal being tested (*exp*) changes value, the
new value is always between the specified minimum (*min*) and
maximum (*max*) less than the previous value.

Note: You can use a four-state version of this checker. See Appendix
A, Four-State OVA Checkers.

Unit Syntax:

```
ova_dec
#(min, max, bw, edge_expr, msg,  severity,
category)
instance_name (en, clk, exp);
```

Template Syntax:

```
dec(en, clk, exp, min, max, msg,  severity,
category);
```

Arguments

min: The minimum change in value. Default = 1.

max: The maximum change in value. Default = 1.

bw: The number of bits in the signal being tested (*exp*). Default = 2.

exp: Signal being tested.

---

ova_delta

Checks that when the signal being tested (*exp*) changes value, the new value is ± the specified minimum (*min*) to maximum (*max*) change of the previous value.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Unit Syntax:

```
ova_delta
#(min, max, bw, edge_expr, msg,  severity,
category)
instance_name (en, clk, exp);
```

Template Syntax:

```
delta(en, clk, exp, min, max, msg,  severity,
category);
```

Arguments

min: The minimum change in value. Default = 1.

max: The maximum change in value. Default = 1.

bw: The number of bits in the signal being tested (*exp*). Default = 2.

exp: Signal being tested.

---

ova_driven

Checks that all bits are driven (none are floating Z or X).

Unit Syntax:

```
ova_driven
#(bw, edge_expr, msg, severity, category,
coverage_level)
instance_name (en, clk, exp);
```

Template Syntax:

```
driven(en, clk, exp, msg, severity, category);
```

Arguments

bw: The number of bits in the signal being tested (*exp*). Default = 2.

exp: Signal being tested.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  cover_exp_not_x_or_z indicates the number of times exp was
  neither x nor z. Not usable with Magellan.

ova_dual_clk_fifo

Implements a checker for a dual-clock, single in- and single out-port queue.

Unit Syntax:

```
ova_dual_clk_fifo
#(depth, elem_sz, hi_water_mark, enq_lat,
deq_lat, oflow_chk, uflow_chk, value_chk,
enq_edge_expr, deq_edge_expr, msg, severity,
category, coverage_level)
instance_name (reset, enq_clk, enq, enq_data,
deq_clk, deq, deq_data);
```

Template syntax:

```
dual_clk_fifo(reset, depth, hi_water_mark,
elem_sz, enq_clk, enq, enq_lat, enq_data,
deq_clk, deq, deq_lat, deq_data, oflow_chk,
uflow_chk, value_chk, msg, severity, category);
```

Arguments

depth: The maximum number of elements in the queue. Default = 2. The specified

depthelem_sz: The size of queue elements in bits. Default = 1.

hi_water_mark: IIf positive, then the depth of the queue after enqueue will be checked to see if *hi_water_mark* is reached. Default = 0.

enq_lat: The number of specified cycles (*enq_clk*) between *enq* being asserted 1 and *enq_data* being valid. Default = 0.

deq_lat: The number of *deq_clk* cycles between *deq* being asserted 1 and *deq_data* being valid. Default = 0.

oflow_chk: If 1, checks that queue does not overflow the maximum size given by the *depth* specification. Default = 1.

uflow_chk: If 1, checks that queue is not empty before dequeuing data. Default = 1.

value_chk: If 1, checks that *deq_data* matches the data at the head of the queue. Default = 1.

enq_edge_expr: The active clock edge of *enq_clk*. Default = 0.

deq_edge_expr: The active clock edge of *deq_clk*. Default = 0.

reset: Initializes the queue to empty when set to 1. : enq_clk: Clock signal for enqueue side.

enq: Set to 1 when data is being enqueued.

enq_data: Data being enqueued.

deq_clk: Clock signal for dequeue side.

deq: Set to 1 when data is being dequeued.

deq_data: Data being dequeued.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  cover_number_of_enqs indicates the number an enqueues.

  Cover cover_number_of_deqs indicates the number of
  dequeues.

  Cover cover_enq_followed_eventually_by_deq indicates
  the number of times an enqueue was followed later by a dequeue.

Level_3 (bit 2 set in coverage_level) : Cover
  cover_fifo_hi_water_chk indicates how many times the high
  water mark was reached on an enqueue.

  Cover cover_number_of_empty indicates how many times
  empty was reached on dequeue.

  Cover cover_number_of_full indicates how many times full
  was reached on enqueue.

- Coverage Levels 1 and 3 are enabled in the unit instance
  (*coverage_level* = 5).

---

ova_even_parity

Checks that the value of the signal being tested (*exp*) always has an
even number of bits set to 1.

Unit Syntax:

```
ova_even_parity
#(bw, edge_expr, msg, severity, category)
instance_name (en, clk, exp);
```

Template Syntax:

```
even_parity(en, clk, exp, msg, severity,
category);
```

Arguments

bw: The number of bits in the signal being tested (*exp*). Default = 2.

 exp: Signal being tested.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  cover_exp_change indicates how many times exp changed
  value.

---

ova_fifo

Implements a checker for a single-clock, single in- and single out-port
queue.The width of the fifo elements is set using the parameter
elem_sz, and the width of the head and tail pointers is set using
ptr_width.

Unit Syntax:

```
ova_fifo
#(depth, elem_sz, hi_water_mark, enq_lat,
deq_lat, oflow_chk, uflow_chk, value_chk,
pass_thru, edge_expr, msg, severity, category,
```

coverage_level, ptr_width1)
instance_name (reset, *clk*, *enq*, *enq_data*, *deq*, *deq_data*);

Template Syntax:

fifo(reset, *clk*, *depth*, *hi_water_mark*, *elem_sz*, *enq*, *enq_lat*, *enq_data*, *deq*, *deq_lat*, *deq_data*, *oflow_chk*, *uflow_chk*, *value_chk*, *pass_thru*, msg, severity, category, ptr_width);

Arguments

depth: The maximum size of the queue. Default = 2. The specified depth

elem_sz: The size of queue elements in bits. Default = 1.

hi_water_mark: If positive, then the depth of the queue after enqueue will be checked to see if *hi_water_mark* is reached. Default = 0.

enq_lat: The number of *enq_clk* cycles between *enq* being asserted 1 and *enq_data* being valid. Default = 0.

deq_latv: The number of *deq_clk* cycles between *deq* being asserted 1 and *deq_data* being valid. Default = 0.

oflow_chk: If 1, checks that queue does not overflow the maximum size given in *depth*. Default = 1.

uflow_chk: If 1, checks that queue is not empty before dequeuing data. Default = 1.

value_chk: If 1, checks that *deq_data* matches the data at the head of the queue. Default = 1.

pass_thru: Specifies behavior when enqueue and dequeue operations happen at the same time with an empty queue. If 0, dequeue happens first, triggering an underflow. If 1, enqueue happens first and the data is passed through. Default = 0.

reset: Initializes the queue to empty when set to 1.

enq: Set to 1 when data is being enqueued.

enq_data: Data being enqueued.

deq: Set to 1 when data is being dequeued.

deq_data: Data being dequeued.

ptr_width: Width of the pointer. Default = 16..

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_number_of_enqs indicates the number of enqueue operations.

Cover cover_number_of_deqs indicates the number of dequeue operations.

Cover cover_simultaneous_enq_deq indicates the number of simultaneous enqueue and dequeue operations.

Cover cover_enq_followed_eventually_by_deq matches whenever there is an enqueue followed eventually by a dequeue.

Level_3 (bit 2 set in coverage_level) : Cover cover_fifo_hi_water_chk indicates how many time the high water mark was reached on an enqueue.

Cover cover_simultaneous_enq_deq_when_empty indicates how many times there were simultaneous enqueue and dequeue operations on an empty queue.

Cover cover_simultaneous_enq_deq_when_full indicates how many times there were simultaneous enqueue and dequeue operations on a full queue.

Cover cover_number_of_empty indicates how many times empty is reached on dequeue.

Cover cover_number_of_full indicates how many times empty is reached on enqueue.

Coverage Levels 1 and 3 are enabled (*coverage_level* = 5).

---

ova_follows

Checks that the follower expression (*follower*) evaluates true within the specified minimum *min_lat* and maximum *max_lat* latency period once the leader expression (*leader*) evaluates true.

Unit Syntax:

```
ova_follows
#(min_lat, max_lat, edge_expr, msg, severity,
category, coverage_level)
instance_name (en, clk, leader, follower);
```

Template Syntax:

```
follows(en, clk, leader, follower, min_lat,
max_lat, msg, severity, category);
```

Arguments

min_lat : Number of clock cycles between the leader signal (*leader*) going true and the beginning of the latency period. Default = 0.

max_lat : Number of clock cycles between leader signal (*leader*) going true and the end of the latency period. Default = 0.

leader : Signal that precedes the follower signal (*follower*).

follower : Signal that follows the leader signal (*leader*).

---

ova_forbid_bool

Checks that the expression is never true

Unit Syntax:

```
ova_forbid_bool
#(edge_expr, msg, severity, category)
instance_name (expr, clk);
```

Template Syntax:

```
forbid_bool(expr, msg, severity, category, clk);
```

Arguments

expr: Signal being tested.

---

ova_hold

Checks that the value of the signal being tested (*exp*) remains constant for the minimum (*min* +1) to maximum (max) number of cycles.

Unit Syntax:

```
ova_hold
#(min, max, bw, edge_expr, msg, severity,
category)
instance_name (en, clk, exp);
```

Template Syntax:

```
hold(en, clk, exp, min, max, msg, severity,
category);
```

Arguments

min: The minimum number of clock cycles (minus one) to hold the value. Default = 0.

max: The maximum number of clock cycles (minus one) to hold the value. Default = 0.

bw: The number of bits in the signal being tested (*exp*). Default = 1.

exp: Signal being tested.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_num_of_exp_changes indicates the number of times exp changed value.

   Cover cover_num_of_matches indicates the number of matches of exp changing value within the specified interval.

Level_3 (bit 2 set in coverage_level) : Cover cover_num_of_matches_exactly_on_min indicates the number of times exp changed exactly min clock cycles.

   Cover cover_num_of_matches_exactly_on_max indicates the number of times exp changed exactly max clock cycles.

---

ova_hold_value

Checks that the signal being tested (*exp*) remains the value to hold (value) from the specified minimum (*min*) to maximum (*max*) number of cycles.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers

Unit Syntax:

```
ova_hold_value
#(min, max, bw, edge_expr, msg, severity,
category, coverage_level)
instance_name (en, clk, exp, value);
```

Template Syntax:

```
hold_value(en, clk, exp, value, min, max, msg,
severity, category);
```

Arguments

min: Minimum number of clock cycles to hold the value. Default =
  0.

max: Maximum number of clock cycles to hold the value. Default =
  0.

bw: The number of bits in the signal being tested (*exp*). Default = 2.

exp: Signal being tested.

value: Value to hold.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  cover_exp_change indicates the number of times exp changed
  to value.

  Cover cover_num_of_matches indicates the number of matches
  of exp holding value within the specified interval.

Level_3 (bit 2 set in coverage_level) : **ova_inc**

Checks that when the signal being tested (exp) changes value, the
new value is always between the specified minimum (min) and
maximum (max) more than the previous value.

Note: You can use a four-state version of this checker. See Appendix
A, Four-State OVA Checkers.

```
ova_inc
#(min, max, bw, edge_expr, msgseverity,
category)
(en, clk, exp);
```

severity, *category*min: The minimum change in value.
  Default = 1.

max: The maximum change in value. Default = 1.

bw: The number of bits in the signal being tested (exp). Default = 2.

exp: Signal being tested.

---

ova_inc

Checks that when the signal being tested (*exp*) changes value, the
new value is always between the specified minimum (*min*) and
maximum (*max*) more than the previous value.

Note: You can use a four-state version of this checker. See Appendix
A, Four-State OVA Checkers

Unit Syntax:

```
ova_inc
#(min, max, bw, edge_expr, msg, severity,
category)
instance_name (en, clk, exp);
```

Template Syntax:

```
inc(en, clk, exp, min, max, msg, severity,
category);
```

Arguments

min: The minimum change in value. Default = 1.

max: The maximum change in value. Default = 1.

bw: The number of bits in the signal being tested (exp). Default = 2.

exp: Signal being tested.

---

ova_memory

Checks the integrity of synchronous memory contents and accesses.

Unit Syntax:

```
ova_memory
#(data_bits, addr_bits, mem_sz, addr_chk,
init_chk, conflict_chk, pass_thru, read1_chk,
write1_chk, value_chk, w_edge_expr, r_edge_expr,
msg, severity, category, coverage_level)
instance_name (start_addr, end_addr, ren, raddr,
rclk, rdata, wen, waddr, wclk, wdata);
```

Template Syntax:

```
memory(data_bits, addr_bits, start_addr,
end_addr, mem_sz, ren, raddr, rclk, rdata, wen,
waddr, wclk, wdata, addr_chk, init_chk,
conflict_chk, pass_thru, read1_chk, write1_chk,
value_chk, msg, severity, category);
```

Arguments

data_bits: Number of bits in the data. Default = 1.

addr_bits: Number of bits in the addresses. Default = 1.

mem_sz: The number of words in the memory. Default = 2.

addr_chk: If 1, checks that address is valid. Default = 1.

init_chk: If 1, checks that addresses read have been previously
   written. Default = 1.

conflict_chk: If 1, checks that simultaneous reading and
   writing of the same address does not occur. Default = 0.

pass_thru: Specifies behavior when read and write happen at the
   same time on the same address. If 0, read gets the old data before
   the write. If 1, read gets the new data after the write. Default = 0.

read1_chk: If 1, checks that an address has at most one read
   between writes. Default = 0.

write1_chk: If 1, checks that an address is read at least once
   before it is over-written. Default = 0.

value_chk: If 1, checks that the value read from an address is the
   value that was written to that address. Default = 0.

w_edge_expr: The active clock edge of wclk. Default = 0.

r_edge_expr: The active clock edge of *rclk*. Default = 0.

start_addr: Starting address of the memory.

end_addr: Ending address of the memory.

ren: Read enable.

raddr: Read address.

rclk: Read clock.

rdata: Read data.

wen: Write enable.

waddr: Write address.

wclk: Write clock.

wdata: Write data.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_number_of_reads indicates the number of read operations to any address.

Cover cover_number_of_writes indicates the number of write operations to any address.

Cover write_followed_by_read indicates how many times a write was followed by a read to the same address.

Level_3 (bit 2 set in coverage_level): Cover cover_two_or_more_writes_without_intervening_read indicates how many times two writes occurred to the same (any) address without an intervening read operation to that address.

Cover cover_two_or_more_reads_without_intervening_write indicates how many times two reads occurred to the same (any) address without an intervening write operation to that address.

Cover simultaneous_read_and_write_to_same_addr indicates how many times (quasi)simultaneous read and write operations occurred to the same (any) address as seen by the read clock rclk.

Cover simultaneous_read_and_write_to_different_addr how many times (quasi)simultaneous read and write operations occurred to the different addresses as seen by the read clock rclk.

Cover read_to_start_addr indicates how many read operations occurred to the address start_addr.

Cover write_to_start_addr indicates how many write operations occurred to the address start_addr.

Cover *read_to_end_addr* indicates how many read operations occurred to the address end_addr.

Cover write_to_end_addr indicates how many write operations occurred to the address end_addr.

Cover write_followed_by_read_to_start_addr indicates how many write operations were followed by a read to the address start_addr.

Cover `write_followed_by_read_to_end_addr` indicates how many write operations were followed by a read to the address `end_addr`.

Coverage Levels 1 and 3 are enabled in the unit instance (*coverage_level* = 5).

---

ova_memory_async

Checks the integrity of asynchronous memory contents and accesses.

Unit Syntax:

```
ova_memory_async
#(data_bits, addr_bits, mem_sz, addr_chk,
init_chk, read1_chk, write1_chk, value_chk, msg,
severity, category)
instance_name (start_addr, end_addr, ren, raddr,
rdata, wen, waddr, wdata);
```

Template Syntax:

```
memory_async(data_bits, addr_bits, start_addr,
end_addr, mem_sz, ren, raddr, rdata, wen, waddr,
wdata, addr_chk, init_chk, read1_chk,
write1_chk, value_chk, msg, severity, category);
```

Arguments

data_bits: Number of bits in the data. Default = 1.

addr_bits: Number of bits in the addresses. Default = 1.

mem_sz: The number of words in the memory. Default = 2.

addr_chk: If 1, checks that address is valid. Default = 1.

init_chk: If 1, checks that addresses read have been previously written. Default = 1.

read1_chk: If 1, checks that an address has at most one read between writes. Default = 0.

write1_chk: If 1, checks that an address is read at least once before it is over-written. Default = 0.

value_chk: If 1, checks that the value read from an address is the value that was written to that address. Default = 0.

start_addr: Starting address of the memory.

end_addr: Ending address of the memory.

ren: Read enable.

raddr: Read address.

rdata: Read data.

wen: Write enable.

waddr: Write address.

wdata: Write data.

ova_multiport_fifo

Implements a checker for a single-clock, multi-port in- and multi-port out queue.

Unit Syntax:

```
ova_multiport_fifo
#(depth, elem_sz, no_ports, hi_water_mark,
enq_lat, deq_lat, oflow_chk, uflow_chk,
value_chk, pass_thru, edge_expr, msg, severity,
category)
instance_name (reset, clk, enq, enq_data, deq,
deq_data);
```

Template Syntax:

```
multiport_fifo(reset, clk, depth,
hi_water_mark, elem_sz, no_ports, enq, enq_lat,
enq_data, deq, deq_lat, deq_data, oflow_chk,
uflow_chk, value_chk, pass_thru, msg, severity,
category);
```

Arguments

reset: Asserted 1 initializes the queue to empty. All operations are synchronous to *clk* ticks, including reset.

enq and *deq*: Bit vectors of equal size *no_ports.*

enq_data: A 2-D array of data. It is assumed that it is dimensioned as `[elem_size-1:0] enq_data [0:no_ports-1]`.

enq_lat: A compile-time, non-negative integer constant that indicates the number of cycles between *enq* being asserted 1 and *enq_data* being valid in the corresponding position.

oflow_chk: When a *enq* is asserted 1: If *oflow_chk* evaluates true, ensures that queue does not overflow the maximum size given in *depth*. The *depth* can be at most 2**16.

hi_water_mark: If positive value, then the level of fill of the queue after enqueue will be checked to see if *hi_water_mark* is reacched. Once high water has been reached once, this check is disabled until the FIFO size falls below the mark again. If *hi_water_mark* = 0 then the high-water mark check is disabled and only overflow is checked, i.e. when the depth of the queue is exceeded (provided that *oflow_chk* = 1).

deq_data: A 2-D array of data. It is assumed that it is dimensioned as `[elem_size-1:0] deq_data [0:no_ports-1]`.

deq_lat: A compile-time non-negative integer constant that indicates the number of cycles between when deq is asserted and deq_data is valid.

uflow_chk: If this evaluates true, it ensures that the queue is not empty (underflow) when a *deq* bit is asserted. If a dequeue on empty is detected then the check is disabled until the next enqueue operation.

value_chk: If this evaluates true, it ensures *deq_data* as selected by the same position as the highest priority The *deq* bit is the same as that at the head of the queue.

pass_thru: If an enqueue and dequeue operation happens simultaneously on an empty queue, then the behavior depends on the *pass_thru* argument to the checker instance (it must be a compile-time constant).

If *pass_thru* = 0 then the dequeue happens before enqueue, hence the empty condition is detected and reported, and an underflow (provided that *uflow_chk* = 1). If *value_chk* = 1 then the value check fails.

If *pass_thru* = 1 then it is assumed that enqueue happens first and the data is immediately dequeued and compared with *deq_data* if *value_chk* is enabled. Also, there is no underflow error reported.

If an enqueue and dequeue operation happens simultaneously on a full queue then no overflow is reported and the new element is enqueued while the element at the head of the queue is dequeued without changing the size of the queue.

---

ova_mutex

Checks that the specified signals (a) and (b) never evaluate true at the same time.

Unit Syntax:

```
ova_mutex
#(edge_expr, msg, severity, category,
coverage_level)
instance_name (en, clk, a, b);
```

Template Syntax:

```
mutex(en, clk, a, b, msg, severity, category);
```

Arguments

a : First signal being tested.

b : Second signal being tested.

signal being tested.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_changes_on_a indicates how many times a changed value.

Cover cover_changes_on_b indicates how many times b changed value.

ova_next_state

Checks that when the signal being tested (*exp*) is in the specified current state (*cs*) it will transition to one of the specified legal next states.

Unit Syntax:

```
ova_next_state
#(no_ns, width, min_hold, max_hold, disallow,
edge_expr, msg, severity, category,
coverage_level)
instance_name (en, clk, exp, cs, ns);
```

Template Syntax:

```
next_state(en, clk, exp, cs, no_ns, ns, min_hold,
max_hold, disallow, msg, severity, category);
```

Arguments

no_ns: The number of legal next states possible from the specified current state (*cs*). Default = 1.

width: The number of bits in the signal being tested (*exp*), the current state (*cs*), and each element of a bitvector of the concatenated legal state values (ns). The vector is ns[width * o_ns-1:0]. Default = 1.

min_hold: The minimum number of clock ticks the signal being tested (*exp*) must hold at the current state (*cs*) value. Default = 1.

max_hold: The maximum number of clock ticks the signal being tested (*exp*) can hold at the current state (*cs*) value. Default = 0.

disallow: If 1, checks that the signal being tested (*exp*) does not transition to any of the values in a specified array of legal states (*ns*). Default = 0.

exp: Signal being tested.

cs: The current state. The check starts when $exp = cs$.

ns: A bitvecor of concatenated legal states that *exp* can transition to from *cs*.

Coverage Modes

Level_1 (bit 0 set in coverage_level): When disallow == 0:

Cover cover_exp_state_transitions indicates how many times a valid transition to a state in ns occurred.

When disallow == 1:

Cover cover_exp_state_transitions indicates how many times a transition to a state other than those in ns occurred.

Level_3 (bit 2 set in coverage_level)
Exists only when disallow == 0: Cover cover_exp_changes_to_ns[i] indicates how many times there was a transition from state cs to state ns[i].

ova_no_contention

Checks that bus signal being tested (*bus*) always has a single active driver and that there is no X or Z on the bus when driven.

Unit Syntax:

```
ova_no_contention
#(min_quiet, max_quiet, bw_en, bw_bus,
edge_expr, msg, severity, category,
coverage_level)
instance_name (en, clk, en_vector, bus);
```

Template Syntax:

```
no_contention(en, en_vector, clk, bus,
min_quiet, max_quiet, msg, severity, category);
```

Arguments

min_quiet : The minimum number of clock cycles between bus transactions. Default = 0.

max_quiet : The maximum number of clock cycles between bus transactions. Default = 0.

bw_en : The number of bits in *en_vector*. Default = 2.

bw_bus : The number of bits in the bus signal being tested (*bus*). Default = 2.

en_vector : Enable signals for bus drivers as a vector.

bus : Bus signal being tested.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_driver_enable indicates how many times bit en_vector[i] was set to 1 (enabled).

Level_3 (bit 2 set in coverage_level) : Cover cover_no_contention_quiet_time_equal_to_min_quiet indicates how many times the observed quiet time is exactly equal to the specified min value.

Cover cover_no_contention_quiet_time_equal_to_max_quiet indicates how many times the observed quiet time is exactly equal to the specified max value.

---

ova_odd_parity

Checks that the value of the signal being tested (*exp*) always has an odd number of bits set to 1.

Unit Syntax:

```
ova_odd_parity
#(bw, edge_expr, msg, severity, category)
instance_name (en, clk, exp);
```

Template Syntax:

```
odd_parity(en, clk, exp, msg, severity, category);
```

Arguments

bw: The number of bits in the signal being tested (*exp*). Default = 2.

exp: Signal being tested.

---

ova_one_cold

Checks that only one bit is set to zero or, optionally, that all bits are set to 1 in the state value.

Unit Syntax:

```
ova_one_cold
#(strict, bw, edge_expr, msg, severity,
category)
instance_name (en, clk, state);
```

Template Syntax:

```
one_cold(en, clk, state, strict, msg, severity,
category);
```

Arguments

strict: If 1, checks for a strict one-cold state encoding. Default = 0.

bw: The number of bits in the signal being tested (*state*). Default = 2.

state: Signal being tested.

---

ova_one_hot

Checks that only one bit is set to one or, optionally, that all bits are set to zero in the state value.

Unit Syntax:

```
ova_one_hot
#(strict, bw, edge_expr, msg, severity,
category)
instance_name (en, clk, state);
```

Template Syntax:

```
one_hot(en, clk, state, strict, msg, severity,
category);
```

Arguments

strict: If 1, checks for a strict one-hot state encoding. Default = 0.

bw: The number of bits in the signal being tested (*state*). Default = 2.

state: Signal being tested.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_state_change indicates how many times test_expr changed value.

```
Level_3 (bit 2 set in coverage_level) : Cover
  cover_state_bit_is_1[i] indicates how many times bit i was
  1 after a change of value of state[i].
```

---

ova_overflow

Checks that the signal being tested (*exp*) does not transition from ≥ *max* to ≤ *min*.

Unit Syntax:

```
ova_overflow
#(min, max, bw, edge_expr, msg, severity,
category, coverage_level)
instance_name (en, clk, exp);
```

Template Syntax:

```
overflow(en, clk, exp, min, max, msg, severity,
category);
```

Arguments

min: The minimum value allowed. Default = 0.

max: The maximum value allowed. Default = 1.

bw: The number of bits in the signal being tested (*exp*). Default = 2.

eexp: Signal being tested.

Coverage Modes

```
Level_1 (bit 0 set in coverage_level) : Cover
  cover_exp_change indicates how many times exp changed
  value.
```

```
Level_3 (bit 2 set in coverage_level) : Cover
  cover_exp_reached_min indicates how many times exp reached
  the min value.
```

Cover cover_exp_reached_max indicates how many times exp reached the max value.

---

ova_quiescent_state

Checks that when *eos_exp* evaluates true, *exp* has value of *fstate*.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Unit Syntax:

```
ova_quiescent_state
#(bw, edge_expr, msg, severity, category)
instance_name (en, clk, exp, fstate, eos_exp);
```

Template Syntax:

```
quiescent_state(en, clk, exp, fstate, eos_exp,
msg, severity, category);
```

Arguments

bw: The number of bits in the signal being tested (*exp*) and the state to match (*fstate*). Default = 2.

exp: Signal being tested.

fstate: State to match.

eos_exp: When true, signals that the state to match (*exp*) is in the state.

---

ova_range

Checks that the signal being tested greater than or equal to the specified minimum value (*min*), and less than or equal to the specified maximum value (*max*).

Unit Syntax:

```
ova_range
#(bw, edge_expr, msg, severity, category,
coverage_level)
instance_name (en, clk, exp, min, max);
```

Template Syntax:

```
range(en, clk, exp, min, max, msg, severity,
category);
```

Arguments

bw: The number of bits in the signal being tested (*exp*), the minimum value allowed (*min*), and the maximum value allowed (*max*). Default = 1.

exp: Signal being tested.

min: The minimum value allowed.

max: The maximum value allowed.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_exp_change indicates how many times exp changed value.

Level_3 (bit 2 set in coverage_level) : Cover cover_exp_reached_min indicates how many times exp reached the min value.

Cover cover_exp_reached_max indicates how many times exp reached the max value.

---

ova_reg_loaded

Checks that the register being tested (*reg*) is loaded with source data (*src*).

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Unit Syntax:

```
ova_reg_loaded
#(delay, end_cycle, bw, edge_expr, msg,
severity, category)
instance_name (en, clk, trigger, src, dst_reg,
stop);
```

Template Syntax:

```
reg_loaded(en, clk, trigger, src, reg, delay,
end_cycle, stop, bw, msg, severity, category);
```

Arguments

delay: The number of cycles after the trigger signal (*trigger*) goes true to start the window. Default = 1.

end_cycle: The number cycles after the trigger signal (*trigger*) goes true to end the window. Default = 1.

bw: The number of bits in the source date (*src*) and the register under test (*reg*). Default = 2.

trigger: Signal that is part of starting the window.

src: Data loaded into the register.

reg: Register being tested.

stop: Signal that stops the check.

---

ova_req_ack_unique

Verifies that each *req* receives an *ack* within the specified interval *min_time* and *max_time* clock *clk* ticks. Note that acks are attributed to reqs in a fifo manner.

Unit Syntax:

```
ova_req_ack_unique
#( min_time, max_time, max_time_log_2,
 edge_expr, msg, version severity, category, ,
coverage_level)
instance name (reset,clk, req, ack): :
```

min_time: Defines the minimum time separation between a req and an ack (default is 1).

max_time

  Defines the maximum time separation between a req and an ack (default is 15).

max_time_log_2

  Specifies the superior integer of log2 of *max_time*, used to dimension the data structures. The default is $\overline{4}$ (= sup(log2(15))).

version: This parameter specifies two versions of the checker: •0 — Selects a version that is suitable for *max_time* <= 15. It uses IDs to identify requests and then generates as many assertions as the *max_time* clock ticks.
  • 1 — Selects a version that is suitable for *max_time* > 15. It uses a time stamp computed mod 2 *max_time* to mark the requests, the time stamp is enqueued. When an ack arrives it verifies that the time stamp at the head of the queue satisfies the timing requirements.

reset: Synchronous reset, active high (1), initializes all request history to nill.

req and ack: The signals of interest.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  cover_number_of_req indicates how many times req was
  asserted.

  Cover cover_number_of_ack indicates how many times ack was
  asserted.

 NOTE: Coverage at Level 3 is only available with version = 1.

Level_3 (bit 2 set in coverage_level) : Cover
  cover_ack_with_exact_min_lat indicates how many time the
  observed latency was exactly equal to the specified min value.

  Cover cover_ack_with_exact_max_lat indicates how many
  time the observed latency was exactly equal to the specified max
  value.

---

ova_req_requires

Checks that if the first expression in a sequence (*trig_req*)
evaluates true, then the second (*follow_req*) and third
(*follow_resp*) expressions in the sequence evaluate true before
the last expression (*trig_resp*) evaluates true.

Unit Syntax:

```
ova_req_requires
#(min_lat, max_lat, edge_expr, msg, severity,
category, coverage_level)
instance_name (en, clk, trig_req, follow_req,
follow_resp, trig_resp);
```

Template Syntax:

```
req_requires(en, clk, trig_req, follow_req,
follow_resp, trig_resp, min_lat, max_lat, msg,
severity, category);
```

Arguments

min_lat: Minimum number of clock cycles between the first
  expression in a sequence (*trig_req*) going true and the last
  expression in the same sequence (*trig_resp*) going true. Default
  = 1.

max_lat: Maximum number of clock cycles between the first
  expression in a sequence (trig_req) going true and the last
  expression in the same sequence (trig_resp) going true. Default = 0.

trig_req: First signal in the sequence.

follow_req: Second signal in the sequence.

follow_resp: Third signal in the sequence.

trig_resp: Last signal in the sequence.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  cover_no_of_trig_reqs indicates the number of times
  trig_req was asserted.

Cover `cover_cover_req_requires` indicates how many times the specified sequence occurred.

Level_2 (bit 1 set in `coverage_level`) : Cover `cover_trig_req_follow_req` indicates how many times there was a a `follow_req` after a `trig_req`.

Cover `cover_trig_req_follow_req_follow_resp` indicates how many times there was a `follow_req` after a `trig_req` and then followed by `follow_resp`.

Level_3 (bit 2 set in `coverage_level`) : Cover `cover_trig_resp_exactly_on_min_lat` indicates how many times the observed latency between `trig_req` and `trig_resp` was exactly equal to the `min` value.

Cover `cover_trig_resp_exactly_on_max_lat` indicates how many times the observed latency between `trig_req` and `trig_resp` was exactly equal to the `max` value.

> **Note:** Coverage is collected correctly only when the transactions delimited by `trig_req` and `trig_resp` asserted do not overlap, i.e., there is no new assertion of `trig_req` while such a transaction is in progress.

---

ova_req_resp

Checks that the rising edge of a bit in the vector of a request signal (*req*) is followed by a single rising edge of the corresponding bit of the vector of a response signal (*resp*) within the latency response window specified by the minimum (*min_lat*) and maximum number of clock cycles (*max_lat*).

Unit Syntax:

```
ova_req_resp
#(no_chnl, min_lat, max_lat, resp_cycles,
no_req4resp, req_till_resp,
req_drop_after_resp, edge_expr, msg, severity,
category)
instance_name (en, clk, req, resp);
```

Template Syntax:

```
req_resp(en, clk, req, resp, no_chnl, min_lat,
max_lat, resp_cycles, no_req4resp,
req_till_resp, req_drop_after_resp, msg,
severity, category, severity, category);
```

Arguments

no_chnl: The number of bits in the vector of the request signal (*req*) and the vector of the response signal (*resp.*) Default = 1.

min_lat: Minimum number of clock cycles between a bit in the specified vector of a request signal (*req*) going true and a bit in the vector of a response signal (*resp*) going true. Default = 1.

max_lat: Maximum number of clock cycles between a bit in the specified vector of a request signal (*req*) going true and a bit in the vector of a response signal (*resp*) going true. Default = 1.

resp_cycles: Number of clock cycles that the vector of a response signal (*resp*) must stay asserted. Default = 0.

no_req4resp: If 1, checks that each response has a corresponding request. Default = 0.

req_till_resp: If 1, checks that the request remains asserted until the response is received. Default = 0.

req_drop_after_resp: Number of cycles after the response is deasserted that the request must be deasserted. Default = 0.

req: Vector of 1-bit request signals.

resp: Vector of 1-bit response signals.

---

ova_sequence

Ensure that *exp* takes on values in the order implied by their sequence in the *vals* bitvector. *bw* is the number of bits in *exp* and in each of the required values.

Unit Syntax:

```
ova_sequence
# (no_vals, min_hold, max_hold, bw, disallow,
edge, exp, msg, severity, category);
(en, clk, exp, vals);
```

Template Syntax:

```
sequence(en, clk, exp, no_vals, vals, min_hold,
max_hold, disallow, bw, msg, severity, category)
```

Arguments

no_vals: An integer value indicating the number of values in the sequence. For example, if *bw* = 3, *no_vals* = 2, and the values are 3'b000 and 3'b110 (to be reached in that order) then the value bound to the *vals* port is 6'b110_000. Default = 2.

min_hold

The minimum number of clock ticks *exp* must hold for each value. Default = 1.

max_hold: The maximum number of clock ticks *exp* must hold for each value. Default = 1.

bw: Number of bits in *exp* and in each of the required values.

disallow: If 1, checks that the sequence is forbidden. Default = 0, sequence is required.

exp: The assertion will check the sequence whenever *exp* takes on the first value in the sequence and *en* evaluates true.

vals: The constant bitvector is formed by concatenating the bitvectors of each of the required values in the sequence, such as., *vals* has *bw* * *no_vals* bits.

ova_stack

Checks operations on a stack.

Unit Syntax:

```
ova_stack
#(depth, elem_sz, hi_water_mark, push_lat,
pop_lat, value_chk, push_pop_chk, edge_expr,
msg, severity, category, coverage_level)
instance_name (reset, clk, push, push_data, pop,
pop_data);
```

Template Syntax:

```
stack(reset, clk, depth, elem_sz, hi_water_mark,
push, push_lat, push_data, pop, pop_lat,
pop_data, value_chk, push_pop_chk, msg,
severity, category);
```

Arguments

depth: The maximum size of the stack. Default = 2.

elem_sz: The size of data elements in bits. Default = 1.

hi_water_mark: If positive, then the depth of the queue after enqueue will be checked to see if *hi_water_mark* is reached. Default = 0.

push_lat: The number of *enq_clk* cycles between *push* being asserted 1 and *push_data* being valid. Default = 0.

pop_lat: The number of *deq_clk* cycles between *pop* being asserted 1 and *pop_data* being valid. Default = 1.

value_chk: If 1, checks that *pop_data* matches the data at the top of the stack. Default = 1.

push_pop_chk: If 1, checks that push and pop operations do not occur simultaneously. Default = 1.

reset: Initializes the stack to empty when set to 1.

push: Set to 1 when data is being pushed.

push_data: Data being pushed.

pop: Set to 1 when data is being popped.

pop_data: Data being popped.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover cover_number_of_pushes indicates how many times there was a push operation.

Cover cover_number_of_pops indicates how many times there was a pop operation.

Cover cover_push_followed_eventually_by_pop indicates how many times a push was followed eventually by a pop without an intervening push.

Level_3 (bit 2 set in coverage_level): Cover
  cover_simultaneous_push_pop indicates how many times
  there were simultaneous push and pop operations.

  Cover cover_simultaneous_push_pop_when_empty indicates
  how many times there were simultaneous push and pop operations
  while the stack was empty.

  Cover cover_simultaneous_push_pop_when_full indicates
  how many times there were simultaneous push and pop operations
  while the stack was full.

  Cover cover_stack_hi_water_chk indicates how many times
  the high water mark was reached.

  Cover cover_number_of_full indicates how many times the
  stack became full after a push.

  Cover cover_number_of_empty indicates how many times the
  stack became empty after a pop.

• Levels 1 and 3 are enabled in the unit instance
  (coverage_level = 6).

---

ova_timeout

Checks that the value of the signal being tested (*exp*), a bit vector,
changes within the specified number of cycles (*period*).

Note: You can use a four-state version of this checker. See Appendix
A, Four-State OVA Checkers.

Unit Syntax:

```
ova_timeout
#(period, bw, edge_expr, msg, severity,
category, coverage_level)
instance_name (en, clk, exp);
```

Template Syntax:

```
timeout(en, clk, exp, period, msg, severity,
category);
```

Arguments

period: The maximum number of clock cycles before the
  specified signal (*exp*) changes. Default = 1.

bw: The number of bits in the specified signal (*exp*). Default = 1.

exp: Signal being tested.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  property exp_change indicates how many times exp changed
  value.

  Cover property cover_exp_changes_within_period indicates
  how many times the change occurred within the required period.

Level_3 (bit 2 set in coverage_level) : Cover
  property cover_exp_changes_exactly_at_period_clks
  indicates how many times exp changed exactly at period clock
  cycles.

---

ova_tri_state

Checks that the tri-states of the specified input and output signals are equal (*inp* == *outp*) at the start of the assertion (*en* is 1).

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Unit Syntax:

```
ova_tri_state
#(bw, edge_expr, msg, severity, category)
instance_name (en, clk, inp, outp);
```

Template Syntax:

```
tri_state(en, clk, inp, outp, msg, severity,
category);
```

Arguments

bw: The number of bits in input (*inp*) and output (*outp*) signals.
  Default = 1.

einp: Input signal.

outp: Output signal.

---

ova_underflow

Checks that the signal being tested (*exp*) does not transition between the specified minimum (*min*) and maximum (*max*) values.

Unit Syntax:

```
ova_underflow
#(min, max, bw, edge_expr, msg, severity,
category)
instance_name (en, clk, exp);
```

Template Syntax:

```
underflow(en, clk, exp, min, max, msg, severity,
category);
```

Arguments

min: The minimum value allowed. Default = 0.

max: The maximum value allowed. Default = 1.

bw: The number of bits in the signal being tested (*exp*). Default = 2.

exp: Signal being tested.

ova_valid_id

Checks that IDS are issued and returned.

Unit Syntax:
```
ova_valid_id
#(id_bw, max_ids, max_out_ids, max_out_per_id,
min_lat, max_lat, edge_expr, msg, severity,
category)
instance_name (en, clk, issued_sig, issued_id,
ret_sig, ret_id, reset_sig, reset_id);
```

Template Syntax:
```
valid_id(en, clk, id_bw, max_ids, max_out_ids,
max_out_per_id, issued_sig, issued_id, ret_sig,
ret_id, reset_sig, reset_id, min_lat, max_lat,
msg, severity, category);
```

Arguments

id_bw: The number of bits in *issued_id*, *ret_id*, and *reset_id*. Default = 2.

max_ids: The maximum number of IDs. Default = 4.

max_out_ids: The maximum number of IDs that can be outstanding. Default = 1.

max_out_per_id: The maximum number of issues outstanding per ID. Default = 1.

min_lat: Minimum number of clock cycles between an ID being issued and returned. Default = 1.

max_lat: Maximum number of clock cycles between an ID being issued and returned. Default = 0.

issued_sig: If 1, *issued_id* has a valid value.

issued_id: The ID being issued.

ret_sig: If 1, *ret_id* has a valid value.

ret_id: The ID being returned.

reset_sig: If 1, *reset_id* has a valid value.

reset_id: The ID whose outstanding count is being reset.

---

ova_value

Checks that the signal being tested (*exp*) is only one of the specified values.

Unit Syntax:
```
ova_value
#(no_vals, disallow, bw, edge_expr, msg,
severity, category, coverage_level)
instance_name (en, clk, exp, vals);
```

Template Syntax:
```
value(en, clk, exp, no_vals, vals, disallow, bw,
msg, severity, category);
```

Arguments

no_vals: The number of entries in the `vals` specification.
  Default = 1.

disallow: If 1, checks that the signal being tested (`exp`) does not
  match any of the values in the specified array of values (`val`).
  Default = 0.

bw: Number of bits in the signal being tested (`exp`) and each
  element of the specified array of values (`val`). Default = 2.

expv: Signal being tested.

vals: A bitvector of concatenated values that the signal being
  tested (`exp`) must evaluate to (`logic [bw*no_vals-1:0] vals`)

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  cover_exp_change indicates how many times exp changed value.

  When disallow == 0:

  Cover cover_value indicates how many times a valid value from
  vals occurred.

  When disallow == 1:

  Cover cover_exp_state_transitions indicates how many
  times a value other than those in vals occurred.

Level_3 (bit 2 set in coverage_level)
Exists only when disallow == 0: Cover
  cover_exp_changes_to_value[i] indicates how many times
  exp was equal to the value vals[i].

---

ova_window

Checks that individual bits in the bitvector signal
(`assert_vector`) are asserted or deasserted either within or
outside the window.

Unit Syntax:

```
ova_window
#(check_type, delay, win_time, bw, edge_expr,
msg, severity, category, coverage_level)
instance_name (en, clk, start_sig, stop_sig,
assert_vector);
```

Template Syntax:

```
window(en, clk, start_sig, delay, stop_sig,
win_time, assert_vector, bw, check_type, msg,
severity, category);
```

Arguments

check_type: The type of check. Default = 0.

- 0: Each bit must be asserted at some time during the window,
  but not necessarily at the same time.
- 1: Each bit must be asserted at least once outside the window.
- 2: Each bit must be de-asserted during the entire length of the
  window.

- **3**: Each bit must be de-asserted on every <clk> outside the window.

delay: The number of cycles after the start signal (*start_sig*) evaluates true to the beginning of the window. Default = 0.

win_time: The maximum length of the window in clock cycles. Default = 1.

bw: The number of bits in the *assert_vector* specification. Default = 1.

start_sig: Signal that is part of starting the window.

stop_sig: Signal that marks the end of the window.

assert_vector: Signal being tested.

.Coverage Modes

Check_type = 0 (asserted inside): Level_1 (bit 0 set in coverage_level) : Cover cover_asserted_in indicates how many times there was a match within the window.

Level_3 (bit 2 set in coverage_level) : Cover cover_num_of_times_bit_asserted_just_after_start_sig_plus_delay[i] indicates how many times bit assert_vector[i] was set to 1 exactly delay cycles after start_sig was asserted.

Cover cover_num_of_times_bit_asserted_just_at_stop_sig[i] indicates how many times bit assert_vector[i] as set to 1 exactly when stop_sig occurred.

Cover cover_num_of_times_bit_asserted_just_at_win_time_expires[i] indicates how many times bit assert_vector[i] as set to 1 exactly when win_time expired.

Cover cover_num_of_times_all_bits_asserted_just_after_start_sig_plus_delay indicates how many times all bits were asserted at the same time delay cycles after start_sig was asserted.

Cover cover_num_of_times_all_bits_asserted_just_at_stop_sig indicates how many times all bits were asserted at the same time when stop_sig was asserted.

Cover cover_num_of_times_all_bits_asserted_just_at_win_time_expires indicates how many times all bits were asserted at the same time when win_time expired.

Check_type = 1 (asserted outside): Level_1 (bit 0 set in coverage_level) : Cover cover_asserted_out indicates how many times there was a match outside the window.

Level_3 (bit 2 set in coverage_level) : Cover cover_num_of_times_bit_asserted_just_at_start_sig_plus_delay[i] indicates how many times bit

assert_vector[i] was set to 1 exactly delay cycles after start_sig was asserted.

Cover
cover_num_of_times_bit_asserted_just_after_stop_sig[i] indicates how many times bit assert_vector[i] as set to 1 just after stop_sig occurred.

Cover
cover_num_of_times_bit_asserted_just_after_win_time_expires[i] indicates how many times bit assert_vector[i] as set to 1 just after win_time expired.

Cover
cover_num_of_times_all_bits_asserted_just_at_start_sig_plus_delay indicates how many times all bits were asserted at the same time at delay cycles after start_sig was asserted.

Cover
cover_num_of_times_all_bits_asserted_just_after_stop_sig indicates how many times all bits were asserted at the same time just after stop_sig was asserted.

Cover
cover_num_of_times_all_bits_asserted_just_after_win_time_expires indicates how many times all bits were asserted at the same time just after win_time expired.

Check_type = 2 (deasserted inside): Level_1 (bit 0 set in coverage_level) : Cover
cover_deasserted_in indicates how many times there was a match within the window.

Level_3 (bit 2 set in coverage_level) : Cover
cover_num_of_times_bit_deasserted_just_after_start_sig_plus_delay[i] indicates how many times bit assert_vector[i] as set to 1 exactly delay cycles after start_sig was asserted.

Cover
cover_num_of_times_bit_reasserted_just_after_stop_sig[i] indicates how many times bit assert_vector[i] rose right after stop_sig occurred.

Cover
cover_num_of_times_bit_reasserted_just_after_win_time_expires[i] indicates how many times bit assert_vector[i] rose right after win_time expired.

Cover
cover_num_of_times_all_bits_deasserted_just_after_start_sig_plus_delay indicates how many times all bits were deasserted at the same time just after delay cycles after start_sig was asserted.

Cover
cover_num_of_times_all_bits_reasserted_just_after_stop_sig indicates how many times all bits were reasserted at the same time just after stop_sig was asserted.

Cover
cover_num_of_times_all_bits_reasserted_just_after_

`win_time_expires` indicates how many times all bits were reasserted at the same time just after `win_time` expired.

`Check_type = 3` (deasserted outside): Level_1 (bit 0 set in `coverage_level`) : Cover `cover_deasserted_out` indicates how many times there was a match outside the window.

`Level_3` (bit 2 set in `coverage_level`) : Cover `cover_num_of_times_bit_reasserted_just_after_start_sig_plus_delay[i]` indicates how many times bit `assert_vector[i]` rose just after delay cycles after start_sig was asserted.

Cover `cover_num_of_times_bit_deasserted_just_after_stop_sig[i]` indicates how many times bit `assert_vector[i]` fell just after `stop_sig` occurred.

Cover `cover_num_of_times_bit_deasserted_just_after_win_time_expires[i]` indicates how many times bit `assert_vector[i]` fell just after `win_time` expired.

Cover `cover_num_of_times_all_bits_reasserted_just_after_start_sig_plus_delay` indicates how many times all bits were reasserted at the same time just after delay cycles after `start_sig` was asserted.

Cover `cover_num_of_times_all_bits_deasserted_just_after_stop_sig` indicates how many times all bits were deasserted at the same time just after `stop_sig` was asserted.

Cover `rvmLIMIT3cover_num_of_times_all_bits_deasserted_just_after_win_time_expires` indicates how many times all bits were deasserted at the same time just after `win_time` expired.

Coverage Levels 1 and 3 are selected (`coverage_level` = 5).

Section 5: OVL-Equivalent Checkers

This chapter describes OVA checkers that verify the same behavior as checkers available in Accellera's proposed "Open Verification Library", Version 02.09.24.

This chapter covers the following topics:

- Converting from a Verilog OVL Library to OVA
- Descriptions of OVL-Equivalent OVA Checkers

Converting from a Verilog OVL Library to OVA

There are several methods you can use to convert OVL Verilog checker instances in a Verilog model to equivalent inlined OVA checkers:

- Single Line Replacement
- Multiple Line Replacement
- Combining OVA and OVL Checkers in the Same Design

This section describes each of these methods.

Single Line Replacement

If the OVL module instance extends over only one line of code, you only need to place the prefix "//ova bind " in front of the original OVL instance.

For example, suppose you want to replace the following OVL module instance:

```
assert_always my_inst (clk, reset_n,
expression);
```

The equivalent OVA is as follows:

```
//ova bind assert_always my_inst (clk, reset_n,
expression);
```

Multiple Line Replacement

If the OVL module instance extends over multiple lines, you can use either of the following two multi-line specification techniques:

Technique #1:

1. Insert a line "/* ova bind" before the instance.

2. insert a line with "*/" after the instance.

Technique #2:

1. Insert a line with "//ova_begin" before the instance.

2. Prefix the OVL instance with "bind".

3. Insert a line with "//ova_end" after the instance.

For example, suppose you want to replace the following OVL instance with an equivalent OVA checker.

```
assert_always #(1, 0, "my_message")
my_always_instance (clk, reset_n, expression);
```

The following example shows a valid forms of specifying OVA inlined checkers:

```
/* ova bind
assert_always #(1, 0, "my_message")
my_always_instance (clk, reset_n, expression);
*/
```

or

```
ova_begin bind
assert_always #(1, 0, "my_message")
my_always_instance (clk, reset_n, expression);
ova_end
```

Combining OVA and OVL Checkers in the Same Design

You can use original OVL checkers and new OVA-based checkers in the same design. Using the `define macro, you can select which checker to use in which situation (e.g., use OVA to get functional coverage information).

For example:

```
`ifdef OVA
//ova bind assert_always #(,,"message-OVA")
my_AG_instance (clk, rst_n, expr);
`elsif
assert_always #(,,"message-OVL") my_AG_instance (clk,
rst_n, expr);
`endif
```

Restrictions

Note the following restrictions when using OVL-equivalent checkers:

- The OVL checker assert_proposition is not available in OVA because it is an asynchronous checker that does not require variable sampling.
- Unit parameters that control the extent of synchronous delays (number of clock ticks) and assertion variants in a checker must be compile-time constants — they must not be specified using design parameters. This restriction concerns the following checkers and parameters:

Checker	Parameters
assert_always_on_edge	edge_type
assert_change	num_cks, flag
assert_cycle_sequence	necessary_condition, num_cks

assert_frame	min_cks, max_cks, flag
assert_handshake	min_ack_cycle, max_ack_cycle, req_drop, deassert_count, max_ack_length
assert_next	num_cks, check_overlapping, only_if
assert_one_cold	inactive
assert_time	num_cks, flag
assert_unchange	num_cks, flag
assert_width	min_cks, max_cks

Inlining OVA Units in a Verilog Wrapper Module

Many of the OVA units described in this chapter can be inlined in a Verilog wrapper module, which then can be instantiated in the design (Note: Of the 30 available checkers, 20 can be used for inlining; exceptions are described in the "Restrictions" section below). The compile command must include the "-ova_inline" option to indicate that the OVA checkers should be processed.

To change from an original OVL checker to a OVA-based checker, you must do the following:

1. Remove any reference to the original OVL library.

2. Compile with the -ova_inline option.

Using OVL-Equivalent Checkers with VHDL Designs

OVA assertions and checkers cannot yet be inlined in VHDL designs. Therefore, the only way to add the checkers to such designs is by creating an external OVA file that contains the appropriate OVA bind statements.

Descriptions of OVL-Equivalent OVA Checkers

This section provides descriptions of OVL-equivalent checkers:

Note: The severity and category parameters for these checkers can take default values as specified by the set_severity and set_category OVA commands.

assert_always

This checker continuously monitors test_expr at every positive edge of clock, clk. It verifies that test_expr will always evaluate TRUE. If test_expr evaluates to FALSE, the assertion will fire.

Syntax

```
assert_always
[#(severity_level, options, msg, category,
coverage_level)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

options: Currently, the only supported option is options=1, which defines the assertion as an assumption for formal tools. The default is 0 (no options specified).

msg: Error message printed when the checker fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the posedge of clk.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover property cover_always indicates the number of times test_expr was asserted when enabled by reset_n.

---

assert_always_on_edge

This checker continuously monitors the test_expr at every specified edge of the sampling_event that coincides with the positive edge of clock, clk. The test_expr should always evaluate TRUE at the sampling_event. If test_expr evaluates to FALSE, the assertion will fire.

Syntax

```
assert_always_on_edge
[#(severity_level, edge_type, options, msg,
category, coverage_level)]
instance_name (clk, reset_n, sampling_event,
test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0.)

edge_type: Selects the transition for sampling_event:

    0 — no edge (default)

    1 — positive edge

    2 — negative edge

    3 — any edge

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: The error message that will be printed if the assertion fires.

category: Checker type (default is 0).

clk: Triggering or clocking event that monitors the assertion.

reset_n: Signal indicating completed initialization.

sampling_event: Expression defines when to evaluate
  test_expr. Transition of sampling_event are selected by
  edge_type.

test_expr: Expression being verified at the positive edge of clk,
  AND if   sampling_event matches transition selected by
  edge_type.

Coverage Modes

Level_1 (bit 0 set in coverage_level): Cover
  property cover_always_on_edge indicates the number of times
  test_expr was asserted on the specified edge of
  sampling_event.

---

assert_change

This checker continuously monitors the start_event at every
positive edge of the clock. When start_event is TRUE, the
checker ensures that the expression, test_expr, changes values on a
clock edge at some point within the next num_cks number of clocks.
This assertion will fire upon a violation.

Syntax
```
assert_change
[#(severity_level, width, num_cks, flag,
options, msg, category, coverage_level)]
instance_name (clk, reset_n, start_event,
test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of the expression, test_expr (default is 1.

num_cks: Number of clocks for test_expr to change its value
  before an error is triggered after start_event is asserted (default
  is 1).

flag: 0 — Ignore any start_event assertion after the first one has
  been   detected.

  1 — Restart the monitoring test_expr, if start_event is asserted
  in any subsequent clock while monitoring test_expr.

  2 — Issue an error if an asserted start_event occurs in any clock
  cycle while monitoring test_expr.

options: Currently, the only supported option is options=1, which
  defines the assertion is an assumption for formal tools.

 msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal that indicates a completed initialization.

start_event: Starting event that triggers monitoring of the test_expr.

test_expr: Expression or variable being verified at the positive edge of clk.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover property cover_change indicates the number of times exp changed within num_cks.

Cover property cover_start_event indicates the number of times start_event occurred.

Level_3 (bit 2 set in coverage_level) : Cover property cover_overlapping_start_events indicates how many times start_event occurred while there was another evaluation attempt in progress.

Cover property cover_change_after_1_clk indicates the number of time test_expr changed value at num_clks clock ticks after start_event.

Cover property cover_change_after_num_cks indicates the number of time test_expr changed value at the next clock ticks after start_event.

---

assert_cycle_sequence

This checker verifies the following conditions:

- When necessary_condition = 0, if all num_cks-1 first events of a sequence (event_sequence[num_cks-1:1]) are TRUE, the last sequence (event_sequence[0]) should follow.
- When necessary_condition = 1, if the first event of a sequence (event_sequence[num_cks-1]) is TRUE, then all the remaining event_sequence[num_cks-2:0] events should follow.

Syntax

```
assert_cycle_sequence
[#(severity_level, num_cks, necessary_condition,
options,  msg, category)]
instance_name (clk, reset_n, event_sequence);
```

Arguments

severity_level: Severity of the failure (default is 0).

num_cks: The length of the event_sequence (number of clock cycles of the event_sequence) that must be valid. Otherwise, the checker will fire.

necessary_condition: Either 1 or 0 (default 0).

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

event_sequence: A Verilog concatenation expression, where each bit represents an event.

---

assert_decrement

This checker continuously monitors the test_expr at every positive edge of the clock signal, clk. It checks that the test_expr will never decrease by anything other than the value specified by value.

Syntax

```
assert_decrement
[#(severity_level, width, value, options, msg,
category)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default 0).

width: Width of test_expr (default is 1).

value: Maximum decrement value allowed for test_expr (default is 1).

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the positive edge of clk.

---

assert_delta

This checker continuously monitors the test_expr at every positive edge of clock signal, clk. It verifies that test_expr will never change value by anything less than "min" and anything more than "max" value.

Syntax

```
assert_delta
[#(severity_level, width, min, max, options,
msg, category)]
instance_name (clk, reset_n, start_event,
test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr (default is 1).

min: Minimum changed value allowed for test_expr in two consecutive clocks of clk (default is 1).

max: Maximum changed value allowed for test_expr in two consecutive clocks of clk (default is 1).

---

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the positive edge of clk.

---

assert_even_parity

This checker continuously monitors the test_expr at every positive edge of the clock signal, clk. It verifies that test_expr will always have an even number of bits asserted.

Syntax
```
assert_even_parity
[#(severity_level, width, options, msg,
category, coverage_level)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr (default is 1).

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the positive edge of clk.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover property cover_test_expr_change indicates how many times test_expr changed value.

---

assert_fifo_index

This checker ensures that the FIFO element:
  • Never overflows and underflows
  • Allows/disallows simultaneous push and pop.

Syntax
```
assert_fifo_index
[#(severity_level, depth, push_width, pop_width,
options, msg, category)]
instance_name (clk, reset_n, push, pop);
```

Arguments

severity_level: Severity of the failure (default is 0).

depth: Depth of the FIFO (default is 1). It should never be set to 0, otherwise an assertion will fire.

push_width: Width of the PUSH signal (default is 1).

pop_width: Width of the POP signal (default is 1).

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

push: FIFO PUSH/enqueue signal.

pop: FIFO POP/dequeue signal.

---

assert_frame

This checker validates proper cycle timing relationships between two events in the design. When a start_event evaluates TRUE, then the test_expr must evaluate TRUE within a minimum and maximum number of clock cycles.

Syntax
```
assert_frame
[#(severity_level, min_cks, max_cks, flag,
options, msg, category)]
instance_name (clk, reset_n, start_event,
test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

min_cks: Minimum number of clock cycles, within which the test_expr should not become TRUE. When min_cks is 0, then test_expr can occur at the same time as start_event or after, as controlled by max_cks. Default is 0.

max_cks: Maximum number of clock cycles, before which test_expr must become TRUE. This check will be disabled when max_cks is not specified. If both min_cks and max_cks are 0 then test_expr must occur at the same time as there is a 0 to 1 transition on start_event. The default is 0.

flag: 0 — Ignores any asserted start_event after the first one has been detected (default).

1 — Restart monitoring test_expr if start_event is asserted in any subsequent clock while monitoring test_expr.

2 — Issue an error if an asserted start_event occurs in any clock cycle while monitoring test_expr.

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0)

clk: Triggering or clocking sampling event for assertion.

reset_n: Signal indicating completed initialization.

start_event

Starting event that triggers monitoring of the test_expr. The start_event is a cycle transition from 0 to 1.

test_expr: Expression being verified at the positive edge of clk.

---

assert_handshake

This checker continuously monitors the req and ack signals at every positive edge of the clock clk. Note that both req and ack must go inactive prior to starting a new cycle.

To activate one or more checks in the checker, the following parameters should be specified with a non-zero value:

min_ack_cycle: When this parameter is greater than 0, the assertion will ensure that an ack does not occur before min_ack_cycle clock ticks.

max_ack_cycle: When this parameter is greater than 0, the assertion will ensure that an ack does not occur after max_ack_cycle clock ticks.

req_drop: When this parameter is greater than 0, the assertion will ensure that req remains active until an ack occurs.

deassert_count: When this parameter is greater than 0, the assertion will ensure that req becomes inactive (0) within deassert_count clock ticks after an ack.

max_ack_length: When this parameter is greater than 0, the assertion will ensure that ack is not asserted for greater than max_ack_length clock cycles and does not become inactive (0) within deassert_count clocks after ack is asserted (that is, check for ack stuck active).

Note that if you do not specify a parameter with a non-zero value, the corresponding check will not be active.

Syntax

```
assert_handshake
[#(severity_level, min_ack_cycle,
max_ack_cycle, req_drop, deassert_count,
max_ack_length, options, msg, category)]
instance_name (clk, reset_n, req, ack);
```

Arguments

severity_level: Severity of the failure (default is 0).

min_ack_cycle: Activate min_ack_cycle check if greater than 0.

max_ack_cycle: Activate max_ack_cycle check if greater than 0.

req_drop: Activate req_drop check if greater than 0.

deassert_count: Activate deassert_count if greater than 0.

max_ack_length: Activate max_ack_length check if greater than 0.

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type, default 0.

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

---

assert_implication

This checker continuously monitors antecedent_expr. If it evaluates to TRUE, then this checker will verify that consequent_expr is TRUE.

When antecedent_expr is evaluated to FALSE, then consequent_expr expression will not be checked at all and the implication is satisfied.

Syntax

```
assert_implication [#(severity_level, options,
msg, category)]
instance_name (clk, reset_n, antecedent_expr,
consequent_expr);
```

Arguments

severity_level: Severity of the failure, default 0.

msg: Error message that will be printed when the assertion fires.

category: Checker type, default 0.

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

antecedent_expr: Expression verified at the positive edge of the clock, clk.

consequent_expr: Expression verified at the positive edge of the clock, clk.

---

assert_increment

This checker continuously monitors test_expr at every positive edge of the clock, clk. It verifies that test_expr will never increase by anything other than the value specified by value. The test_expr can be any valid Verilog expression. The check will not start until the first clock after the reset_n is asserted.

Syntax

```
assert_increment
[#(severity_level, width, value, options, msg,
category)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr (default is 1).

value: Maximum increment value allowed for test_expr
  (default is 1).

options: Currently, the only supported option is options=1, which
  defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the positive edge of clk.

---

assert_never

This checker continuously monitors test_expr at every positive
edge of clock, clk. It verifies that test_expr will never evaluate
TRUE. The test_expr can be any valid Verilog expression. When
test_expr evaluates TRUE, this checker will fail.

Syntax

```
assert_never
[#(severity_level, options, msg, category)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

options: Currently, the only supported option is options=1, which
  defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type, default 0.

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the positive edge of clk.

---

assert_next

This checker verifies the proper cycle timing relationship between two
events in the design at every posedge of the clock, clk. When a
start_event evaluates TRUE, then test_expr must evaluate
TRUE exactly num_cks number of clock cycles later.

This checker supports overlapping sequences. For example, if you assert that `test_expr` will evaluate TRUE exactly four cycles after `start_event`, it is not necessary to wait until the sequence finishes before another sequence can begin.

Syntax

```
assert_next [#(severity_level, num_cks,
check_overlapping, only_if, options, msg,
category)]
instance_name (clk, reset_n, start_event,
test_expr);
```

Arguments

`severity_level`: Severity of the failure (default is 0).

`num_cks`: Number of clocks for the test_expr to become TRUE after `start_event` is asserted (default is 0).

`check_overlapping`: If set to 1, permits overlapping sequences. In other words, a new `start_event` can occur (starting a new sequence in parallel) while the previous sequence continues. (Default is 1.)

`only_if`: If set to 1, a `test_expr` can only evaluate TRUE if preceded `num_cks` earlier by a `start_event`. If `test_expr` occurs without a `start_event`, then an error is reported. Default 0.

`options`: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`: Error message that will be printed when the assertion fires.

`category`: Checker type (default is 0).

`clk`: Sampling clock of the assertion on posedge clk.

`reset_n`: Signal indicating completed initialization.

`start_event`: Starting event that triggers monitoring of the `test_expr`.

`test_expr`: Expression or variable being verified at the positive edge of clk.

---

assert_no_overflow

This checker ensures that the expr, from 'max' value, never goes to a value that is less than or equal to 'min' and greater than 'max', at every posedge of the clock, clk.

Syntax

```
assert_no_overflow
[#(severity_level, width, min, max, options,
msg, category)]
instance_name (clk, reset_n, expr);
```

Arguments

`severity_level`: Severity of the failure (default is 0).

`width`: Width of the monitored expression, expr (default is 1).

min: Minimum value limit for the expr at clock tick t+1 when expr == max at clock tick 't' (This value is excluded from the acceptable range). Default is 0.

max: Maximum value limit for the expr at clock tick t+1 when expr == max at clock tick 't' (This value is included in the acceptable range). Default is 1.

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type, default 0.

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

expr: Expression being verified at the positive edge of clk.

---

assert_no_transition

This checker ensures that, when the state variable test_expr reaches a value specified by start_state, it does not transit to a state/value specified by next_state. All variables are sampled at posedge of the clock, clk.

Syntax

```
assert_no_transition
[#(severity_level, width, options, msg,
category)]
instance_name (clk, reset_n, test_expr,
start_state, next_state);
```

Arguments

severity_level: Severity of the failure (default is 0.)

width: Width of test_expr, start_state, and next_state signals (default is 1).

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the positive edge of clk.

start_state: State value at the start. When test_expr equals this value, the evaluation starts.

next_state: Next state value. Once test_expr matches with start_state, test_expr should not transit to this value at the next clock tick.

assert_no_underflow

This checker ensures that test_expr never changes from 'min' value to a value that is less than 'min' and greater than or equal to 'max'.

Syntax

```
assert_no_underflow
[#(severity_level, width, min, max, options,
msg, category)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of the monitored expression, test_expr. Currently, this value is limited to 32 bits due to a Verilog limitation on the number of bits in a parameter. Default 1.

min: Minimum value limit for the test_expr at clock tick t+1 when test_expr == max at clock tick 't' (this value is included in the acceptable range).

max: Maximum value limit for the test_expr at clock tick t+1 when test_expr == max at clock tick 't' (This value is excluded from the acceptable range).

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the positive edge of clk.

assert_odd_parity

This checker monitors for odd number of '1's in test_expr at every positive edge of the clock, clk.

Syntax

```
assert_odd_parity
[#(severity_level, width, options, msg,
category, coverage_level)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr (default is 1).

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: The sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at every positive edge of clk.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover property cover_test_expr_change indicates how many times test_expr changed.

---

assert_one_cold

This checker ensures that the variable, test_expr, has only one bit low at any positive clock edge when the checker is configured for no inactive states.

The checker can also be configured to accept all bits equal to either 0 or 1 as the inactive level.

Syntax

```
assert_one_cold
[#(severity_level, width, inactive, options,
msg, category, coverage_level)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr (default is 32).

inactive: Specifies the inactive state of test_expr:

   inactive = 0 allows the inactive state of test_expr to be all zeros.

   inactive = 1 allows the inactive state of test_expr to be all ones.

   inactive = 2 (default) specifies that no inactive state is allowed.

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: The sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression to be verified for "one cold" at the positive edge of clk.

Coverage Modes

Level_1 (bit 0 set in coverage_level): Cover property cover_test_expr_change indicates how many times test_expr changed value.

Level_2 (bit 1 set in coverage_level) : Cover property cover_test_expr_with_all_1 indicates how many times test_expr was all 1s. Enabled when inactive == 1.

Cover property cover_test_expr_with_all_0 indicates how many times test_expr was all 0s. Enabled when inactive == 0.

---

assert_one_hot

This checker ensures that the variable, test_expr, has only one bit high at any positive clock edge.

Syntax

```
assert_one_hot
[#(severity_level, width, options, msg,
category, coverage_level)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr (default is 32).

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: The sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression to be verified for "one hot" at the positive edge of clk.

Level_3 (bit 2 set in coverage_level) : Cover property cover_test_expr_bit_is_0[i] indicates how many times bit i of test_expr was 0 when test_expr changed value.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover property cover_test_expr_change indicates how many times test_expr changed value.

Level_3 (bit 2 set in coverage_level) : Cover property cover_test_expr_bit_is_1[i] indicates how many times bit i of test_expr was 1 when test_expr changes value.

---

assert_quiescent_state

This checker verifies that the value in the variable state_expr, is equal to the value specified by check_value when a sampled positive edge is detected on sample_event.

Syntax

```
assert_quiescent_state
[#(severity_level, width, options, msg,
category)]
instance_name (clk, reset_n, state_expr,
check_value, sample_event);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of state_expr and check_value signals (default is 1).

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

state_expr: Variable to be checked at every posedge of clk.

check_value: Signal that holds the value to be compared with state_expr when sample_event is asserted.

sample_event: Sampling trigger signal.

---

assert_range

This checker ensures that the value of test_expr will always be within the 'min' and 'max' value range.

Syntax

```
assert_range
[#(severity_level, width, min, max, options,
msg, category)]
instance_name (clk, reset_n, test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr (default is 1).

min: Minimum value allowed for range check (default is 0).

max: Maximum value allowed for range check. (default is 1).

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression being verified at the positive edge of clk.

---

assert_time

This checker continuously monitors the start_event at every positive edge of the clock, clk. When start_event is TRUE, the checker ensures that the expression, test_expr, is TRUE up to num_cks number of clock ticks.

Syntax
```
assert_time
[#(severity_level, num_cks, flag, options, msg,
category)]
instance_name (clk, reset_n, start_event,
test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

num_cks: Number of clock ticks for test_expr to remain TRUE
  after start_event is asserted.

Flag: 0 - Ignores any asserted start_event after the first one has
  been   detected.

   1 - Restart monitoring test_expr, if start_event is asserted in
      any subsequent clock cycle while monitoring test_expr.

   2 - Issue an error if an asserted start_event occurs in any
      clock cycle while monitoring test_expr.

options: Currently, the only supported option is options=1, which
  defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the assertion.

reset_n: Signal indicating completed initialization.

start_event: Starting event that triggers monitoring of
  test_expr.

test_expr: One-bit variable verified at the positive edge of clk.

---

assert_transition

This checker ensures that, when the state variable test_expr reaches
the value specified by start_state, it does transit to a state/value
specified by next_state.

Syntax
```
assert_transition
[#(severity_level, width, options, msg,
category)]
instance_name (clk, reset_n, test_expr,
start_state, next_state);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr, start_state, and next_state
  signals (default is 1).

options: Currently, the only supported option is options=1, which
  defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression verified at the positive edge of clk.

start_state: Start value of test_expr. When test_expr equals this value, the verification begins.

next_state: Next value. Once test_expr matches with start_state, test_expr should transit to this next value (or hold at start_state).

---

assert_unchange

This checker monitors the start_event at every positive edge of the clock, clk. When start_event is TRUE, the checker ensures that the expression, test_expr does not change its value within num_cks clocks.

Syntax

```
assert_unchange
[#(severity_level, width, num_cks, flag,
options, msg, category, coverage_level)]
instance_name (clk, reset_n, start_event,
test_expr);
```

Arguments

severity_level: Severity of the failure (default is 0).

Width: Width of test_expr (default is 1).

Num_cks: Number of clock ticks for test_expr to remain unchanged after start_event is asserted.

Flag: 0 - Ignores any asserted start_event after the first one has been detected.

  1 - Re-start monitoring test_expr if start_event is asserted in any subsequent clock while monitoring test_expr.

  2 - Issue an error if an asserted start_event occurs in any clock cycles while monitoring test_expr.

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

start_event: Starting event that triggers monitoring of test_expr.

test_expr: Expression verified at the positive edge of clk.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover property `cover_start_event` indicates how many times `start_event` was asserted.

Cover property `cover_unchange` indicates how many times `test_expr` remained stable the required time interval.

Level_3 (bit 2 set in coverage_level): Cover property `cover_overlapping_start_events` indicates how many times a `start_event` occurred while a previously triggered evaluation attempt was still in progress.

---

assert_width

This checker ensures that, when `test_expr` becomes TRUE it should remain TRUE at least for 'min' number of clock cycles and at most 'max' number of clock cycles. It should never remain TRUE beyond that limit.

Syntax

```
assert_width
[#(severity_level, min_cks, max_cks, options,
msg, category)]
instance_name (clk, reset_n, test_expr);
```

Arguments

`severity_level`: Severity of the failure (default is 0).

`min_cks`: `test_expr` should be held TRUE at least for `min_cks` number of clocks (default is 1).

`max_cks`: `test_expr` should not be held TRUE for more than `max_cks` number of clocks (default is 1).

`options`: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`: Error message that will be printed when the assertion fires.

`category`: Checker type (default is 0).

`clk`: Sampling clock of the checker.

`reset_n`: Signal indicating completed initialization.

`test_expr`: Expression verified at every positive edge of `clk`.

---

assert_win_change

This checker ensures that `test_expr` changes its value at least once between the assertions of `start_event` and `end_event`.

Syntax

```
assert_win_change
[#(severity_level, width, options, msg,
category)]
instance_name (clk, reset_n, start_event,
test_expr, end_event);
```

Arguments

`severity_level`: Severity of the failure (default is 0).

width: Width of the monitored expression, `test_expr`. (default is 1).

options: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type, (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

start_event: Start of the window.

test_expr: Expression verified at the positive edge of `clk`.

end_event: End of the window.

---

assert_win_unchange

This checker ensures that the `test_expr` never changes its value between the assertions of `start_event` and `end_event`.

Syntax

```
assert_win_unchange
[#(severity_level, width, options, msg,
category)]
instance_name (clk, reset_n, start_event,
test_expr, end_event);
```

Arguments

severity_level: Severity of the failure (default is 0).

width: Width of the monitored expression, test_expr (default is 1).

options: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

start_event: Start of the window.

test_expr: Expression being verified at the positive edge of `clk`.

end_event: End of the window.

---

assert_window

This checker ensures that `test_expr` is asserted 1 as long as the window is open. Window open and close events are signaled by `start_event` and `end_event` expressions. The verification starts on the next clock tick following `start_event`.

Syntax
```
assert_window
[#(severity_level, options, msg, category)]
instance_name (clk, reset_n, start_event,
test_expr, end_event);
```
Arguments

severity_level: Severity of the failure (default is 0).

options: Currently, the only supported option is options=1, which
  defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

start_event: Start of the window.

test_expr: Signal being verified at the positive edge of clk.

end_event: End of the window.

---

assert_zero_one_hot

This checker ensures that the variable, test_expr, has only one bit 1
or all bits 0 at any positive edge of the clock, clk.

Syntax
```
assert_zero_one_hot
[#(severity_level, width, options, msg,
category, coverage_level)]
instance_name (clk, reset_n, test_expr);
```
Arguments

severity_level: Severity of the failure (default is 0).

width: Width of test_expr (default is 32).

options: Currently, the only supported option is options=1, which
  defines the assertion is an assumption for formal tools.

msg: Error message that will be printed when the assertion fires.

category: Checker type (default is 0).

clk: Sampling clock of the checker.

reset_n: Signal indicating completed initialization.

test_expr: Expression to be verified for "one hot or all bits 0" at
  the positive edge of clk.

Coverage Modes

Level_1 (bit 0 set in coverage_level) : Cover
  property cover_test_expr_change indicates how many times
  test_expr changed value.

Level_2 (bit 1 set in coverage_level) : Cover
  property cover_test_expr_with_all_0 indicates how many
  times the all 0 value occurred when test_expr changed value.

Level_3 (bit 2 set in coverage_level) : Cover
  property `cover_test_expr_bit_is_1[i]` indicates how many
  times bit `test_expr[i]` was 1 after a change of value.

Section 6: System Tasks

This chapter describes OVA system tasks.

Calls from within Code

To start monitoring:
```
$ova_start[(levels [, module, entity, or scope
arguments])];
```

To stop monitoring:
```
$ova_stop[(levels [, module, entity, or scope
arguments])];
```

To control category and severity-based assertion monitoring
```
$ova_category_start(category)
```
Starts all assertions associated with the specified category level (an unsigned integer from 0 to $2^{24} - 1$).

```
$ova_category_stop(category)
```
Stops all assertions associated with the specified category.

```
$ova_severity_start(severity)
```
Starts all assertions associated with the specified severity level (an unsigned integer from 0 to 255).

```
$ova_severity_stop(severity)
```
Stops all assertions associated with the specified severity level.

To specify the response to an assertion failure:

```
$ova_severity_action(level, action);
$ova_severity_action(level, action);
```

Where action can be specified as continue, stop or finish.

Task Invocation from CLI
```
$ova_stop levels
$ova_start levels modname
$ova_category_start(category)
$ova_category_stop(category)
$ova_category_action level "action"
$ova_severity_start(severity)
$ova_severity_stop(severity)
$ova_severity_action level "action"
```

Debug Control Tasks
```
$ovadumpoff;
$ovadumpon;
```

Section 7: Additional OpenVera Assertions Features

This section lists some additional features.

Compatibility with Verilog Logical Expression

Verilog Operators

Operator	Description
{}, {{}}	Concatenation, replication
+ - * / **	Arithmetic
%	Modulus
> >= < <=	Relational
!	Logical negation
&&	Logical and
||	Logical or
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality
~	Bit-wise negation
&	Bit-wise and
|	Bit-wise inclusive or
^	Bit-wise exclusive or
^~ or ~^	Bit-wise equivalence
&	Reduction and
|	Reduction or
~|	Reduction nor
^	Reduction exclusive or
~^ or ^~	Reduction xnor
<<	Left shift
>>	Right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift
? :	Conditional

Precedence of Binary Operators

`[:]`	Highest precedence
`+    -    !    ~`	
`**`	
`*    /    %`	
`+    -   (binary)`	
`<<   >>   <<<   >>>`	
`<    <=    >    >=`	
`==    !=    ===    !==`	
`&    ~&`	
`^    ^~    ~^`	
`|    ~|`	
`&&`	
`||`	
`?:   (conditional operator)`	
`*[]`	
`->>    #`	
`length in   istrue in`	
`ended   matched`	Lowest precedence

Verilog Compiler Directives

```
`include  "filename"

`ifdef text_macro_name
  first_group_of_lines
  [ `else
    second_group_of_lines
  ]
`endif

`define text_macro_name  macro_text

`undef text_macro_name
```

Appendix A Four-State OVA Checkers

Checkers in the OVA Checker Library, except ova_driven, ova_forbid_bool, and ova_no_contention, use boolean equality (==, !=) in their underlying assertions. While the checkers may be used in four-state simulation, they do not detect equality or inequality on x and z (a check of x == y will be false if any operand has an x or z, even if it is in the same bit position).

The following come in two versions. The default one uses boolean equality while another version, also available in the library under a different name, supports case equality (===). The latter checkers in both unit and template forms are located in files having the postfix .4state in the OVA Checker Library. Note that most formal tools, supports only synthesizable assertions, and so for use with such tools, only the default checkers should be used (case equality (===) as in all .4state checkers is not synthesizable, whereas boolean equality is).

.

ova_arith_overflow	ova_hold_value
ova_const	ova_inc
ova_data_used	ova_quiescent_state
ova_dec	ova_reg_loaded
ova_delta	ova_tri-state
ova_hold	ova_timeout

To use the 4-state checkers, you define a macro of the same name as the checker in the OVA file, and then `include the checker file.

For example, to use four-state version of the std unit ova_inc, add the following two lines at the beginning of the OVA file:

```
`define inc
`include "$VCS_HOME/etc/ova/inc_u.ova.4state"
```

If you need only the inc template, the two lines become:

```
`define inc
`include "$VCS_HOME/etc/ova/inc.ova.4state"
```

Index