```
apCounterMaxed : assert property ( // assuming a default clocking
      go |=> (cntr <= 3) ) else
            $fatal(2, "%0t SVA_ERR: Counter exceeded 3,
               cntr %0d; sampled(cntr) %0d ",
               $time, cntr, $sampled(cntr)); // ch4/counter_err.sv
```

The simulation with QuestaSim provided the following information:

```
# ** Fatal: 1100 SVA_ERR: Counter exceeded 3, cntr 5; sampled(cntr) 4
#    Time: 110 ns Started: 90 ns  Scope: counter.apCounterMaxed File: junk2.sv Line:
13 Expr: cntr<=3
```

- **$error** represents a run-time error. Example:

```
ap_handshake : assert property ($rose(req) |=> ##[0:4] ack) else
   $error ("%m @ time %0t, req = %0h, ack=%0h", $time, $sampled(req), $sampled (ack));
```

- **$warning** is a run-time warning, which can be suppressed in a tool-specific manner. Example:

```
ap_handshake : assert property ( $rose(req) |=> ##[0:4] ack) else
  $warning ("%m @ time %0t, req = %0h, ack=%0h", $time, $sampled(req), $sampled (ack));
```

- **$info** indicates that the assertion failure carries no specific severity. Example:

```
ap_handshake :  assert property ($rose(req) |=> ##[0:4] ack) else
    $info ("%m @ time %0t, req = %0h, ack=%0h", $time, $sampled(req), $sampled (ack));
```

4.2.3.2 UVM severity levels [44]

The Universal Verification Methodology (UVM) 1.1 Class Reference addresses verification complexity and interoperability within companies and throughout the electronics industry for both novice and advanced teams while also providing consistency. UVM has claimed wide acceptance in the verification methodology of advanced designs. For the reporting of messages, UVM provides several macros that resemble the SystemVerilog severity levels, but provide more options and consistency with the UVM verification environment. The UVM severity macros are briefly addressed in this section for those users who are using UVM and want to maintain that consistency. Refer to manual for more information.

Note: The UVM severity macros are not related to the assertion-control system tasks (described in Section 4.2.4) and those macros do not affect the simulation statistics. They just reduce the outputs.

Four macros are presented here: **`uvm_info, `uvm_warning, `uvm_error, `uvm_fatal**

`uvm_info: `uvm_info(ID,MSG,VERBOSITY)

Calls uvm_report_info if VERBOSITY is lower than the configured verbosity of the associated reporter. ID is given as the message tag and MSG is given as the message text. The file and line are also sent to the uvm_report_info call.

`uvm_warning: `uvm_warning(ID,MSG)

Calls uvm_report_warning with a verbosity of UVM_NONE. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. ID is given as the message tag and MSG is given as the message text. The file and line are also sent to the uvm_report_warning call.

`uvm_error: `uvm_error(ID,MSG)

Calls uvm_report_error with a verbosity of UVM_NONE. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. ID is given as the message tag and MSG is given as the message text. The file and line are also sent to the uvm_report_error call.

`uvm_fatal: `uvm_fatal(ID,MSG)

Calls uvm_report_fatal with a verbosity of UVM_NONE. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. ID is given as the message tag and MSG is given as the message text. The file and line are also sent to the uvm_report_fatal call.

---

44 Universal Verification Methodology (UVM) 1.1 Class Reference
http://www.accellera.org/downloads/standards/uvm

The verbosity of the message indicates its relative importance. If this number is less than or equal to the effective verbosity level UVM_HIGH, UVM_MEDIUM, UVM_LOW, UVM_NONE, then the report is issued, subject to the configured action and file descriptor settings. Verbosity is ignored for warnings, errors, and fatals to ensure users do not inadvertently filter them out. Thus, setting the verbosity level with the command option `+UVM_VERBOSITY=UVM_LOW` means that the user wants a low level of outputs, and the triggered action blocks that use the **`uvm_info** macro with verbosity UVM_LOW will be displayed. However, setting the verbosity level with the command option `+UVM_VERBOSITY=UVM_MEDIUM` would have the triggered action blocks that use the **`uvm_info** macro with verbosity `UVM_LOW` or `UVM_MEDIUM` be displayed. Those action blocks that use **`uvm_info** macro with verbosity `UVM_HIGH` or `UVM_FULL` will not be displayed. Setting the verbosity level to `UVM_NONE` inhibits the displays. The following presents an example application of the UVM macros for error reporting and the output displays with various verbosity options. The purpose of this model is to demonstrate the concepts, rather the affirmation that one assertion statement has lower (e.g., **`uvm_info**) relevance than a higher one (e.g., (**`uvm_error**).

```
import uvm_pkg::*;  `include "uvm_macros.svh"
module uvm_sva_ex;   // File c/uvm_sva_ex.sv
    bit clk, a, b, c, req, ack;
    parameter CLK_HPERIOD = 10;
    string tID="UART ";
    initial begin : clk_gen forever #CLK_HPERIOD clk <= !clk; end : clk_gen
    default clocking def_cb @ (posedge clk);  endclocking : def_cb
    ap_LOW: assert property(a) else
        `uvm_info(tID,$sformatf("%m : error in a %b", a), UVM_LOW); // Line 9
    ap_MEDIUM: assert property(a) else
        `uvm_info(tID,$sformatf("%m : error in a %b", a), UVM_MEDIUM); // Line 11
    ap_HIGH: assert property(a) else
        `uvm_info(tID,$sformatf("%m : error in a %b", a), UVM_HIGH);   // Line 13
    ap_FULL: assert property(a) else
        `uvm_info(tID,$sformatf("%m : error in a %b", a), UVM_FULL);   // Line 15
    ap_test2: assert property(a) else
        `uvm_error(tID,$sformatf("%m : error in a %b", a));       // Line 17
    ap_handshake0 : assert property ($rose(req) |=> ##[0:4] ack) else
        $error(tID, $sformatf("%m req = %0h, ack=%0h",
                    $sampled(req), $sampled (ack)));    // Line 20
    ap_handshake : assert property ($rose(req) |=> ##[0:4] ack) else
        `uvm_error(tID, $sformatf("%m req = %0h, ack=%0h",
              $sampled(req), $sampled (ack)));    // Line 23
    //…
endmodule : uvm_sva_ex
```

Simulation produced the following results:

compilation_command uvm_sva_ex.sv

simulation_command +**UVM_VERBOSITY=UVM_HIGH** uvm_sva_ex

Compilation and Simulation with verbosity option

…

run 400ns

From tID="UART

…

```
# UVM_INFO uvm_sva_ex.sv(13) @ 10: reporter [UART ] uvm_sva_ex.ap_HIGH : error in a 0
# UVM_ERROR uvm_sva_ex.sv(17) @ 10: reporter [UART ] uvm_sva_ex.ap_test2 : error in a 0
# UVM_INFO uvm_sva_ex.sv(11) @ 10: reporter [UART ] uvm_sva_ex.ap_MEDIUM : error in a 0
# UVM_INFO uvm_sva_ex.sv(9) @ 10: reporter [UART ] uvm_sva_ex.ap_LOW : error in a 0
# UVM_INFO uvm_sva_ex.sv(13) @ 30: reporter [UART ] uvm_sva_ex.ap_HIGH : error in a 0
# UVM_ERROR uvm_sva_ex.sv(17) @ 30: reporter [UART ] uvm_sva_ex.ap_test2 : error in a 0
# UVM_INFO uvm_sva_ex.sv(11) @ 30: reporter [UART ] uvm_sva_ex.ap_MEDIUM : error in a 0
# UVM_INFO uvm_sva_ex.sv(9) @ 30: reporter [UART ] uvm_sva_ex.ap_LOW : error in a 0
# UVM_INFO uvm_sva_ex.sv(13) @ 50: reporter [UART ] uvm_sva_ex.ap_HIGH : error in a 1
# UVM_ERROR uvm_sva_ex.sv(17) @ 50: reporter [UART ] uvm_sva_ex.ap_test2 : error in a 1
# UVM_INFO uvm_sva_ex.sv(11) @ 50: reporter [UART ] uvm_sva_ex.ap_MEDIUM : error in a 1
# UVM_INFO uvm_sva_ex.sv(9) @ 50: reporter [UART ] uvm_sva_ex.ap_LOW : error in a 1
# ** Error: UART uvm_sva_ex.ap_handshake0 req = 0, ack=0
#    Time: 170 ns Started: 70 ns  Scope: uvm_sva_ex.ap_handshake0 File: uvm_sva_ex.sv Line: 20 Expr: ack
# UVM_ERROR uvm_sva_ex.sv(23) @ 170: reporter [UART ] uvm_sva_ex.ap_handshake req = 0, ack=0
```

$error reporting

```
compilation_command uvm_sva_ex.sv
simulation_command +UVM_VERBOSITY= UVM_LOW uvm_sva_ex
..
run 400ns
..
# UVM_ERROR uvm_sva_ex.sv(17) @ 10: reporter [UART ] uvm_sva_ex.ap_test2 : error in a 0
# UVM_INFO uvm_sva_ex.sv(9) @ 10: reporter [UART ] uvm_sva_ex.ap_LOW : error in a 0
# UVM_ERROR uvm_sva_ex.sv(17) @ 30: reporter [UART ] uvm_sva_ex.ap_test2 : error in a 0
# UVM_INFO uvm_sva_ex.sv(9) @ 30: reporter [UART ] uvm_sva_ex.ap_LOW : error in a 0
# UVM_ERROR uvm_sva_ex.sv(17) @ 50: reporter [UART ] uvm_sva_ex.ap_test2 : error in a 1
# UVM_INFO uvm_sva_ex.sv(9) @ 50: reporter [UART ] uvm_sva_ex.ap_LOW : error in a 1
# ** Error: UART uvm_sva_ex.ap_handshake0 req = 0, ack=0
#    Time: 170 ns Started: 70 ns  Scope: uvm_sva_ex.ap_handshake0 File: uvm_sva_ex.sv Line: 20 Expr: ack
# UVM_ERROR uvm_sva_ex.sv(23) @ 170: reporter [UART ] uvm_sva_ex.ap_handshake req = 0, ack=0
```

$error reporting

4.2.4 Assertion-control system tasks

SystemVerilog provides system tasks to control the evaluation of assertions (e.g., ON/OFF). In simulation, this can be very useful to disable property checking until the design under verification is in a stable initialized state. This feature allows the speedup in simulation when assertion checking is not needed and eliminates the need to use the **disable iff** (reset_n) to prevent false reporting when a user does not need to verify the assertions during initialization. The control system tasks are also useful for turning on/off assertions and cover statements based on the testbench, versus complex *ifdefs* or *generates* that evolve over time.

📖 **Rule:** The assertion system tasks must be in a procedural block, such as an **initial** or **always** block.

IEEE 1800-2009 provided global assert controls that affected all assertions in the design. It did not provide a way to control assertions based on their type. There was no way for the users to only disable **cover** directives but keep the **assert** and **assume** directives as enabled. Similarly, there was no way to only enable concurrent assertions and switch off immediate assertions. IEEE 1800-2012 made enhancements to allow those tight control features. The syntax for the assertion control syntax is as follows:

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

Compatibility with 1800-2009

1800-2012

```
assert_task ::=
    $asserton
  | $assertoff
  | $assertkill
list_of_scopes_or_assertions ::=
   scope_or_assertion { ,  scope_or_assertion }
scope_or_assertion ::=
   hierarchical_identifier
```

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

4.2.4.1 Assert control

📖 **Rule**: The **$assertcontrol** provides finer granularity in how and which types of assertions are controlled. The syntax is:

```
$assertcontrol ( control_type [ , [ assertion_type ] [ , [ directive_type ]
                        [ , [ levels ]  [ , list_of_scopes_or_assertions ] ] ] ] ) ;
```

[1] The arguments for the **$assertcontrol** *system task are described below:*

– control_type*: This argument controls the effect of the* **$assertcontrol** *system task. This argument shall be an integer expression.* Section 4.2.4.1.1 describes the valid values of this argument.

– assertion_type*: This argument selects the assertion types that are affected by the* **$assertcontrol** *system task. This argument shall be an integer expression.* Section 4.2.4.1.2 describes the valid values for this argument.