APPENDIX

A

OPEN VERIFICATION LIBRARY

The Accellera Open Verification Library [Accellera OVL 2003] provides designers, integrators, and verification engineers with a single, vendor-independent assertion standard interface for design validation using simulation, semi-formal verification, and formal verification techniques. By using a single, well-defined interface, the Open Verification Library bridges the gap between different types of verification, and makes more advanced verification tools and techniques available to non-expert users.

The Open Verification Library is composed of a set of Verilog and VHDL assertion monitors that are defined by the Accellera Open Verification Library committee. This set of monitors enables the designer to check specific properties of a design.

In this section, we discuss thirteen of the most popular OVL monitors. For a complete listing of Accellera Open Verification Library monitors, see *www.verificationlib.org*.

A.1 OVL methodology advantages

The Accellera Open Verification Library (OVL) assertion monitors provide many systematic elements for an effective assertion-based verification methodology, which are typically not addressed by general property languages. For instance, the OVL incorporates a consistent and systematic means of specifying RT-level implementation assertions structurally through a set of concurrent assertion monitors. These monitors provide designers with a module, which guides them to express a broad class of

assertions. In addition, these monitors address methodology considerations by providing uniformity and predictability within an assertion-based verification flow and encapsulating the following features:

- unified and systematic method of reporting that can be customized per project
- common mechanism for enabling and disabling assertions during the verification process
- systematic method of filtering the reporting of a specific assertion violation by limiting the firing report to a configured amount

Finally, the OVL does not require a pre-processor to take advantage of assertion specification in the RTL source. Furthermore, the designer does not have to wait until EDA vendors provide tool support for emerging standards since the library is written in standard IEEE-1364 Verilog and IEEE-1076 VHDL. In other words, it will work *right out of the box* for today's designs. This means that IP containing assertions can be delivered to customers without the need to deliver any addition tools for preprocessing the assertion into simulation monitors.

A.2 OVL standard definition

All assertion monitors defined by the Open Verification Library initiative observe the following BNF format, defined in compliance with Verilog Module instantiation of the IEEE Std 1364-1995 "Verilog Hardware Description Language".[1]

```
assertion_instantiation ::= assert_identifier
   [parameter_value_assignment] module_instance ;

parameter_value_assignment ::= #(severity_level
   {,other parameter expressions}, options, msg)

module_instance ::= name_of_instance
   ([list_of_module_connections])

name_of_instance ::= module_instance_identifier

list_of_module_connections ::=
    ordered_port_connection
```

1. In this appendix, we provide a formal definition for the Verilog version of the OVL. The definition for the VHDL version of the library can be obtained at [Accellera OVL 2003].

```
    {,ordered_port_connection}
  | named_port_connection {,named_port_connection}

ordered_port_connection ::= [expression]

named_port_connection ::= .port_identifier
    ([expression])

assert_identifier ::= assert_[type_identifier]

type_identifier ::= identifier
```

A.2.1 OVL runtime macro controls

The Assertion Monitor Library currently includes four Verilog Macro Global Variables:

- `ASSERT_GLOBAL_RESET
- `ASSERT_MAX_REPORT_ERROR
- `ASSERT_ON
- `ASSERT_INIT_MSG

These four variables are described briefly in the table below and in greater detail in the following paragraphs.

Variable	Definition
ASSERT_GLOBAL_RESET	Overrides individual *reset_n* signals
ASSERT_MAX_REPORT_ERROR	Defines the number of errors required to trigger a report
ASSERT_ON	Enables assertion monitors during verification
ASSERT_INIT_MSG	Prints a report that lists the assertions present in a given simulation environment.

The `list_of_module_connections` has one required parameter, `reset_n`. The signal `reset_n` is an active low signal that indicates to the assertion monitor when the initialization of the circuit being monitored is complete. During the time when *reset_n* is low, the assertion monitor will be disabled and initialized. Alternatively, to specify a reset_n signal or condition for each assertion monitor, you may specify the global macro variable `ASSERT_GLOBAL_RESET. If this variable is defined, all instantiated monitors will disregard their respective *reset_n* signals. Instead, they will be initialized whenever `ASSERT_GLOBAL_RESET is low.

Every assertion monitor maintains an internal register *error_count* that stores the number of times the assertion monitor instance has fired. This internal register can be accessed by the testbench to signal when a given testbench should be aborted. When the global macro variable `ASSERT_MAX_REPORT_ERROR is defined, the assertion instance will stop reporting messages if the number of errors for that instance is greater than the value defined by the `ASSERT_MAX_REPORT_ERROR macro.

To enable the assertion monitors during verification, you must define the macro 'ASSERT_ON (for example, +define+ASSERT_ON). During synthesis, the ASSERT_ON would not be defined. In addition, *//synthesis translate_off* meta-comments are contained within the body of each monitor to prevent accidental synthesis of the monitor logic.

When you define the `ASSERT_INIT_MSG macro, an "initial" block calls a task to report the instantiation of the assertion. This macro is useful for identifying each of the assertions present in a given simulation environment.

Please note: Most assertions are triggered at the positive edge of a triggering signal or expression `clk`. The assertion assert_proposition is an exception, it monitors an expression at all times.

A.2.2 Customizing OVL messages

The OVL, which is available online at *www.verificationlib.org,* includes a file named *ovl_task.h,* as shown in Example A-1. This file contains a set of tasks that allow you to customize the following:

- simulation startup identification of assertions
- error message reporting mechanism
- actions associated with assertion firing (for example, $finish)

You may use this file to call your own PLI user-defined task upon triggering an assertion, as an alternative to the default $display reporting mechanism currently built into the OVL. To take advantage of this feature, simply edit the *ovl task.h* file shown below to reflect your preferences.

Example A-1 Definition for Verilog version of *ovl_task.h*

```
task ovl_error;
    input  [8*63:0]  err_msg;
  begin

`ifdef ASSERT_MAX_REPORT_ERROR
    if (error_count <= `ASSERT_MAX_REPORT_ERROR)
`endif
    if (severity_level == 0) begin
      error_count = error_count + 1;
      $display ("OVL_FATAL : %s : %s : %0s : \ severity %0d : time %0t : %m" ,
        assert_name, msg, err_msg,
        severity_level, $time) ;
      ovl_finish;
    end

  else if (severity_level == 1) begin
      $display ("OVL_ERROR : %s : %s : %0s : \ severity %0d : time %0t : %m" ,
        assert_name, msg, err_msg,
        severity_level, $time) ;
      error_count = error_count + 1;
    end

    else if (severity_level == 2) begin
      $display ("OVL_WARNING: %s : %s : %0s : \ severity %0d : time %0t : %m",
        assert_name, msg, err_msg,
        severity_level, $time) ;
      ovl_warning;
    end

    else begin
      if ((severity_level > 4) ||
          (error_count == 1))
        $display ("OVL_NOTE : %s : %s : %0s : \ severity %0d : time %0t : %m" ,
          assert_name, msg,err_msg,
          severity_level, $time) ;
    end

 end
endtask
task ovl_finish;
  begin
     #100 $finish;
  end
endtask
```

Example A-1 Definition for Verilog version *ovl_task.h*

```
task ovl_warning;
  begin
      // Some user defined Stuff Here, e.g., PLI
  end
endtask

task ovl_init_msg;
  begin
    $display ("OVL_NOTE: %s initialized @ %m \ Severity: %0d, Message: %s",
           assert_name, severity_level, msg);
  end
endtask
```

A.3 Firing OVL monitors

During simulation, an OVL assertion monitor will "fire" (that is, report an error) when the specific check performed by the OVL monitor detects an error, generally on the rising edge of the user-supplied sample clock. The ovl_error() task is called to report the assertion violation.

However, when the expression being asserted (that is, checked) is not synchronized with the assertion sample clock (for example, due to a race condition) either a non-deterministic triggering delay or false assertion firing may occur.

- *Non-deterministic triggering delay* refers to the delay between the time the error condition occurs and the time it is detected.
- *False firing* can occur with more complex assertions if the *test_expr* and assertion sample clock are not synchronized properly.

To avoid these consequences, consider an appropriate assertion monitor sampling clock c*lk* related to the test_expr. Furthermore, using a variable derived from non-blocking assignments within the `test_expr` greatly minimizes the possibility of race conditions. Experience has shown that most false firings of assertions that are a result of race conditions, are due to signals originating from the testbench or from blocking assignments within the RTL code.

A.4 Using OVL assertion monitors

The OVL set of assertion monitors can be used to improve design verification concerns. In general, follow the guidelines listed below when making decisions about placement for assertion monitors in your RTL code:

- Include assertion monitors to capture all design assumptions during RTL coding, or corner case concerns.
- Include assertion monitors when a module has an external interface. In this case, assumptions on the correct input and output behavior should be guarded and verified.
- Include assertion monitors when interfacing with third party modules, since the designer may not be familiar with the module description (as in the case of IP cores) or may not completely understand the module. In these cases, guarding the module with assertion monitors may prevent incorrect use of the module.

Usually, a specific assertion monitor is suited to cover a desired property, which poses a potential problem. In other cases, even though a specific assertion monitor may not exist, a combination of two or three assertion monitors that perhaps include some additional RTL code (for example, bracketed by 'ifdef ASSERT_ON in Verilog) provide the desired coverage.

The number of actual assertions you must add to a specific design varies. A design might require a few or hundreds, depending on the complexity of the design and the complexity of the properties that must be checked.

Writing assertion monitors for a given design requires careful analysis and planning for maximum efficiency. While writing too few assertions may not increase the coverage on a design, writing too many assertions may increase verification time, sometimes without increasing the coverage. In most cases, however, the runtime penalty incurred by adding assertion monitors is relatively small. The significant reduction in debug time provided by an assertion-based methodology more than compensates for the incremental overhead in simulation performance. Hence, the designer should not hesitate to add assertions for all potential corner case concerns within the design. Without capturing these potential problem points or design assumptions within the implementation, there is little chance that these corner case concerns will be validated by the verification team (which is generally focused on verifying the design at a higher level of abstraction).

The following sections provide details related to specific OVL monitors for various classes of properties to be checked.

A.5 Checking invariant properties

The OVL provides a set of monitors that check invariant properties. Invariant properties are conditions that must hold, or not hold, for *all* cycles. This section discusses a few of the more common invariant OVL monitors. A complete list of OVL monitors is located at *www.verificationlib.org* [OVL 2002].

A.5.1 assert_always

checking a property that always holds

The OVL assert_always assertion is used to check that an invariant property *always* holds at every clock boundary. For instance, Example A-2 provides a simple demonstration of an OVL assert_always to monitor a unique counter's specified range (that is, 0 to 8).

Example A-2 Monitor legal range for count variable with assert_always

```
module counter_0_to_8 (reset_n, clk, inc, dec) ;
 input reset_n, clk, inc, dec;
 reg [3:0] count;

 always @ (posedge clk) begin
   if (reset_n) count = 4'b0;
   else count = count + inc - dec;
 end
 // OVL check for valid range
 assert_always #(0, 0, 0, "range 0-8 error")
    valid_count (clk, reset_n,
       (count >= 4'b0000) && (count <= 4'b1000));
endmodule
```

Whenever the `inc` signal is TRUE, the counter increments by one. Whenever the `dec` signal is TRUE, the counter decrements by one. For this unique counter, the user must ensure that the controlling logic, which generates the signal `inc` and `dec`, always maintains a specified range for the variable count (that is, 0 to 8).

The syntax for the OVL assert_always monitor is defined as follows:

Syntax ***assert_always [#(severity_level, options, msg)]***
inst_name (clk, reset_n, test_expr);

severity_level	Severity of the failure with default value of 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
test_expr	Expression being verified at the positive edge of *clk*.

Example A-3 defines the semantics for the OVL assert_always. Note that the *ovl_task.h* definition was previously defined in Example A-1.

Example A-3 Verilog definition for the OVL assert_always

```
module assert_always (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter options = 0;
  parameter msg="VIOLATION";
   input clk, reset_n, test_expr;

//synopsys translate_off
`ifdef ASSERT_ON
  parameter assert_name = "ASSERT_ALWAYS";
   integer error_count;
   initial error_count = 0;

`include "ovl_task.h"

  `ifdef ASSERT_INIT_MSG
     initial
       ovl_init_msg;
   `endif
always @(posedge clk) begin
     `ifdef ASSERT_GLOBAL_RESET
       if (`ASSERT_GLOBAL_RESET != 1'b0) begin
     `else
       if (reset_n != 0) begin  // active low reset
     `endif
        if (test_expr != 1'b1) begin
          ovl_error("");

        end
       end
  end
`endif
//synopsys translate_on
endmodule
```

A.5.2 assert_never

checking a property that never holds

The OVL assert_never monitor allows us to specify an invariant property that should *never* evaluate to TRUE. For instance, if we modify Example A-2 and specify that the `count` variable should

never be greater than 8, then we can use the OVL assert_never monitor as shown in Example A-4 to check this condition.

Example A-4 Monitor legal for count variable with assert_never

```
module counter_0_to_8(reset_n, clk, inc, dec);
 input reset_n, clk, inc, dec;
 reg [3:0] count;

 always @(posedge clk) begin
   if (reset_n) count = 4'b0;
   else count = count + inc - dec;
 end
 // OVL check for valid range
 assert_never #(0, 0, 0, "range 0-8 error")
   valid_count (clk, reset_n, (count > 4'b1000));
endmodule
```

The syntax for the OVL assert_never monitor is defined as follows:

Syntax ***assert_never [#(severity_level, options, msg)]***
inst_name (clk, reset_n, test_expr);

severity_level	Severity of the failure with default value of 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset _n*).
test_expr	Expression being verified at the positive edge of *clk*.

Note that the parameter and argument options specified for the assert_never module are the same as the options used in the assert_always monitor. Also, the semantic definition for assert_never is similar to the definition for assert_always. However, the assert_never monitor checks that *test_expr* is *never* equal to 1 'b1, while the assert_always monitor checks that *test_expr* is *always* equal to 1 'b1.

A.5.3 assert_zero_one_hot

checking for mutually exclusive events

The assert_always and the assert_never monitors are convenient for specifying general invariant properties. However, there are many specific invariant properties where formulating the Boolean expression is cumbersome or awkward. Examples include: checking for one-hot or one-cold conditions, even or odd parity, or valid variable range conditions.

For instance, Example A-5 illustrates how we could check that the individual bits contained within the `cntrl` variable are mutually exclusive. For this assertion, we use the mathematical property:

```
(cntrl & (cntrl-1))
```

which will always equal zero if the `cntrl` bit vector variable is either all zeroes—or only one bit of the variable is active high at a time. We could check this condition using an assert_always monitor and our mathematical equation, as shown in Example A-5.

Example A-5 Check for zero or one hot condition on cntrl variable

```
module pass_mux (clk, reset_n, cntrl, in, out) ;
 input   clk,   reset_n;
 input [3:0] cntrl,
 input [3:0] in;
 output     out;
 reg        out;
 always @ (posedge clk) begin
   case (cntrl)
     4'b0001: out <= in[0];
     4'b0010: out <= in[1];
     4'b0100: out <= in[2];
     4'b1000: out <= in[3];
   endcase
 end
// check for zero or one-hot values for cntrl
assert_always valid_cntrl (clk, reset_n,   ( (cntrl  &
 (cntrl-1) ) == 4'b0) ) ;
endmodule
```

However, as previously stated, checking for a zero or one hot condition as demonstrated in Example A-5 is awkward.

Alternatively, the OVL assert_zero_or_one_hot monitor, as shown in Example A-6, simplifies our coding effort while explicitly stating the property we are asserting.

Example A-6 Using assert_zero_one_hot to check cntrl

```
assert_zero_one_hot # (0, 4) valid_cntrl (clk,
                                  reset_n,  cntrl)  ;
```

The syntax for the OVL assert_zero_one_hot monitor is defined as follows:

Syntax

```
assert_zero_one_hot [#(severity_level, width, options, msg)]
    inst_name (clk, reset_n, test_expr);
```

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*.
options	Vendor options. Currently, the only supported option is *options*=1 , which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
test_expr	Expression being verified at the positive edge of *clk*.

The assert_zero_one_hot assertion is most useful in control circuits. It ensures that the state variable of a finite-state machine (FSM) implemented with zero-one-hot encoding will maintain proper behavior. In data path circuits, assert_zero_one_hot can be used to ensure that the enabling signals of bus-based designs will not generate bus contention. Examples of uses for assert_zero_one_hot include controllers, circuit enabling logic, and arbitration logic. Finally, assert_zero_one_hot is useful for checking mutual exclusivity between multiple events. For instance, if fsm_1, fsm_2, and fsm_3 are not permitted to be in a WR_MODE at the same time, then we could write an assertion as follows using the Verilog concatenation operator to create a bit vector that must either be zero or one-hot:

ExampleA-7 Ensure WR_MODE mutual exclusivity between multiple FSMs

```
assert_zero_one_hot # (0,3) wr_mode (clk, reset_n,
            {fsm_1==`WR_MODE,
            fsm_2==`WR_MODE,
            fsm_3==`WR_MODE});
```

A.5.4 assert_range

checking valid ranges

The OVL assert_range continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It asserts that a specified Verilog expression will always have a value

within a legal *min/max* range, otherwise, a monitor will fire (that is, an error condition will be detected in the Verilog code). The *test_expr* can be either a valid Verilog wire or reg variable, or any valid Verilog expression. The *min* and *max* should be a valid parameter and *min* must be less than or equal to *max.*

Syntax ***assert_range [#(severity_level, width, min, max, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr.*
min	Minimum value allowed for range check. Default to 0.
max	Maximum value allowed for range check. Default to (2**width - 1).
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
test_expr	Expression being verified at the positive edge of *clk.*

The assert_range assertion should be used in circuits to ensure the proper range of values in control structures, such as counters and finite-state machines (FSM). In datapath circuits, this assertion can be used to check whether the variable or expression is evaluated within the allowed range.

For instance, Example A-8 demonstrates a counter whose range must remain between 1 and 9. The assertion ensures that the controlling logic generating the inc and dec commands maintains a valid range for this counter.

Example A-8 Monitor legal range for count variable with assert_never

```
module counter_1_to_9 (reset_n, clk, inc, dec) ;
 input reset_n, clk, inc, dec;
 reg [3:0] count;

 always @ (posedge clk) begin
   if (reset_n) count = 4'b1;
   else count = count + inc - dec;
 end
 // OVL check for valid range 1 thru 9
 assert_range # (0,4,1,9) count_range_check (clk, reset_n, count);
endmodule
```

A.6 Checking cycle relationships

In the previous section, we discussed the OVL invariant class of assertions. While specifying properties that must hold (or never hold) for all cycles is certainly important, there are times when it is necessary to be more specific about the correct cycle relationship between multiple events over time. In this section, we introduce a set of OVL monitors that permit us to assert (that is, validate) correct cycle relationships.

A.6.1 assert_next

The OVL assert_next assertion validates proper cycle timing relationships between two events in the design. For instance, whenever event A occurs, then event B must occur on the following cycle (that is, A -> next B). In this implication, event A is referred to as the *antecedent,* while event B is referred to as the *consequence.* For the OVL monitor, the antecedent is represented by the *start_event* expression, while the consequence is represented by the *test_expr* expression.

In addition to advancing time by one cycle after the occurrence of *start_event* for the check of *test_expr* (that is, event B must occur exactly one cycle after event A) it is possible to specify that the *test_expr* will hold multiple clock cycles after the occurrence of the *start_event* expression. This can be specified through the OVL *num_cks* parameter.

Syntax

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

severity_level	Severity of the failure with default value of 0.
num_cks	The number of clocks *test_expr* must evaluate to TRUE after *start_event* is asserted.
check_overlapping	If 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 to permit overlapping sequences (that is, default is 1).
only_if	If 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 flagged. Default is 0.

options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
start_event	Starting event that triggers monitoring of the *test_expr*.
test_expr	Expression being verified at the positive edge of *clk*.

An important feature of this assertion is that it supports overlapping sequences (as an option). For instance, 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.

The assert_next assertion should be used in circuits to ensure a proper sequence of events. Common uses for assert_next are as follows:

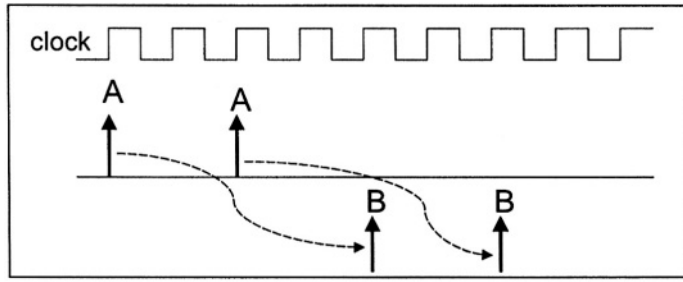
- verification that multicycle operations with enabling conditions will always work with the same data
- verification that single-cycle operations operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require that data is stable after a specified initial triggering event (such as in an asynchronous transaction requiring req/ack signals)

For instance, Example A-9 includes two overlapping sequences that are being verified as shown in [1997]. The assertion claims that when 'A' occurs, 'B' will occur exactly 4 cycles later. Notice how a new 'A' (or *starting event*) does occur in this example prior to the completion of the first sequence (that is, *test_expr* 'B'). The assertion would be coded as follows:

Example A-9 assert_next

```
assert_next # (0,4,1) AB_check (clk, reset_n, A, B);
```

Figure A-1 **Overlapping sequences of A's and B's.**



A.6.2 assert_frame

The OVL assert_next monitor allows us to validate an exact cycle relationship between two events within a design. However, often the cycle relationship between multiple events is specified in terms of a range of possibilities (that is, a min/max frame window). The OVL assert_frame assertion validates proper cycle timing relationships within a frame window between two events (that is, Boolean expressions) 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. If the *test_expr* does not occur within these frame window boundaries, an assertion will fire (that is, an error condition will be detected).

The intent of the minimum and maximum range is to identify legal boundaries in which *test_expr* can occur at or after *start_event.* When you specify both the minimum (equal to or greater than 0) and maximum (greater than 0) ranges, a *test_expr* must occur within the specified frame. However, if you do not provide a maximum range, the checker will ensure that the *test_expr* does not occur until after *min_cks.* The frame is from the time of *start_event* through *max_cks.* Additionally, *max_cks* must be greater than *min_cks.*

Special consideration one. If you do not specify a maximum range, the checker ensures that the *test_expr* does not occur until *min_cks* or later. That is, the *test_expr* must not occur before *min_cks.* However, *test_expr* is not required to occur after *min_cks,* we are simply asserting that *test_expr* does not occur before *min_cks.*

Special consideration two. If you specify that *min_cks* is equal to 0, the checker ensures that the *test_expr* occurs prior to *max_cks.* That is, *test_expr* must occur at some point in any cycle from *start_event* through *max_cks*).

Special consideration three. If you specify that both *min_cks* and *max_cks* equal 0, *test_expr* must be true when there is a 0 to 1 transition for *start_event.* That is, *start_event* implies *test_expr (start_event* g *test_expr).*

Syntax

assert_frame [#(severity_level, min_cks, max_cks, flag, options, msg)] inst_name (clk, reset_n, start_event, test_expr);

severity_level	Severity of the failure.
min_cks	The *test_expr* cannot occur prior to (but not including) the specified minimum number of clock cycles. That is, if *test_expr* occurs at or after *start_event* but before *min_cks,* then an error occurs. **The exception is:** When *min_cks* is set to 0, then there is no minimum check (that is, *test_expr* may occur at start event).
max_cks	The *test_expr* must occur at or prior to the specified number of clock cycles. That is, if the *test_expr* does not occur at or prior to *max_cks,* then an error occurs. **The exception is:** When *max_cks* is set to 0, then there is no maximum check (any value is valid).
flag	0 - Ignores any asserted *start_event* after the first one has been detected (default); 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 cycle while monitoring *test_expr.*
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking sampling event for assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
start_event	Starting event that triggers monitoring of the *test_expr.*
test_expr	Expression being verified at the positive edge of *clk.*

The assert_frame assertion should be used in control circuits to ensure proper synchronization of events. Common uses of assert_frame are as follows:

- verification that multicycle operations with enabling conditions will always work with the same data

- verification of single cycle operations to operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require that data is stable after a specified initial triggering event (such as in an asynchronous transaction requiring req/ack signals)

Example A-10 demonstrates how assert_frame can be used to verify cycle timing relationships between two events. This assertion claims that after the rising edge of `req` is detected, then an `ack` signal must go high within two to four clocks.

Example A-10 assert_frame

```
assert_frame #(0,2,4) check_req_ack (clk, reset_n, req, ack);
```

A.6.3 assert_cycle_sequence

The OVL assert_next and assert_frame enable us to validate the cycle relationship between two events (that is, Boolean expressions that evaluate to TRUE). However, at times we would like to validate a sequence of events. The OVL assert_cycle_sequence can be used to validate the proper occurrence of multiple events within a design.

A contiguous sequence of events is represented by a Verilog (or VHDL) concatenation of multiple Boolean expressions. For instance, in Verilog, the sequence A, followed by B, followed by C, would be expressed as {A,B,C}. This expression is then passed in as the *event_sequence* argument to the assert_cycle_sequence monitor. The size of *event_sequence* is the width of the concatenation expression, which represent the number of clocks required to move forward in time to validate the sequence. Hence, for the *event_sequence* {A,B,C}, the number of clocks (that is, *num_cks*) is 3.

The **assert_cycle_sequence** assertion checks the following:

- If the OVL *necessary_condition* parameter is 0, then this assertion checks to ensure that if all *num_cks-1* prefix (that is, *event_sequence[num_cks-1:1]*) are satisfied, then the last event ("event_sequence[0]") must hold.
- If "necessary_condition" is 1, then this assertion checks that once the first event ("event_sequence[num_cks-1]") holds, all the remaining events must be satisfied.

Syntax *assert_cycle_sequence [#(severity_level, num_cks, necessary_condition, options, msg)] inst_name (clk, reset_n, event_sequence);*

severity_level	Severity of the failure with default value of 0.
num_cks	The width of the *event_sequence* (length of number of clock cycles in the sequence) that must be valid. Otherwise, the assertion will fire; that is, an error occurs.
necessary_condition	Either 1 or 0. The default is 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
event_sequence	A Verilog concatenation expression, where each bit represents an event.

The assert_cycle_sequence assertion should be used in circuits to ensure a proper sequence of events. An event is a Verilog expression that evaluates TRUE. Common uses for assert_cycle_sequence are as follows:

- verification that multicycle operations with enabling conditions will always work with the same data
- verification of single cycle operations to operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require that data is stable after a specified initial triggering event (such as in an asynchronous transaction requiring req/ack signals)

The following example asserts that when write cycle starts, followed by one wait statement, then the next opcode will have the value `'DONE`.

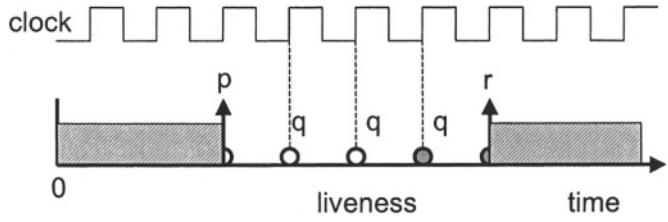
Example A-11 assert_cycle_sequence

```
assert_cycle_ sequence #(0,3) init_test (clk, reset_n,
        {r_opcode == 'WRITE, r_opcode == 'WAIT,
         r_opcode == 'DONE}) ;
```

A.7 Checking event bounded windows

Many temporal properties are bounded by events (that is, Boolean expressions). For instance, often the designer would like to check that after a *start_event* occurs, then a specific Boolean expression must change values prior to the occurrence of an *end_event.* Figure A-2 illustrates this idea. For this example, the *start_event* is p, while the *end_event* is r. Hence, the Boolean expression q must change values (represented by the color change) after p and before r.

Figure A-2 Event bounded window for a liveness property



Alternatively, the designer might check that the specified Boolean expression must not change values in an event bounded window.

A.7.1 assert_win_change

The assert_win_change assertion continuously monitors the *start_event* at every positive edge of the triggering event or clock *clk.* When this signal (or expression) evaluates TRUE, the assertion monitor ensures that the *test_expr* changes values prior to the *end_event.*

When the *start_event* evaluates TRUE, the assertion monitor ensures that the *test_expr* changes values on a clock edge at some point up to and including the next *end_event.* Once the *test_expr* evaluates TRUE, it is not necessary for it to remain TRUE throughout the remainder of the test (up to and including *end_event*). Hence, the *test_expr,* does not have to be TRUE at the *end_event,* provided it was true at some point during the test (up to and including *end_event*).

Syntax ***assert_win_change [#(severity_level, width, options, msg)] inst_name (clk, reset_n, start_event, test_expr, end_event);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
start_event	Starting event that triggers monitoring of the *check_expr*
test_expr	Expression being verified at the positive edge of *clk*.
end_event	Event that terminates monitoring of a *start_event*.

The assert_win_change assertion should be used in circuits to ensure that after a specified initial event, a particular variable or expression will change after the *start_event* and before the *end_event*. Common uses for assert_win_change include:

- verification that synchronization circuits respond after a specified initial stimuli. For example, that a bus transaction will occur without any bus interrupts. Another example is that a memory write command will not occur while we are in a memory read cycle.
- verification that finite-state machines (FSM) change state or will go to a specific state after a specified initial stimuli and before another specified event occurs

In Example A-12, an assertion checks whether *data_bus* is asserted before an asynchronous read operation is finished.

Example A-12 assert_win_change

```
module processor (clk, reset_n, …, rd, rd_ack, … ) ;
  input  clk;
  input  reset_n;

  output  rd
  input rd_ack;

  inout  [31:0]  data_bus;

  ...

  assert_win_change #(0,32) sync_bus_with_rd
    (clk, reset_n, rd, data_bus, rd_ack) ;

endmodule
```

A.7.2 assert_win_unchange

The assert_win_unchange assertion continuously monitors the *start_event* at every positive edge of the triggering event or clock *clk*. When this signal (or expression) evaluates TRUE, the assertion monitor ensures that the *test_expr* will not change in value up to and including the *end_event.*

Syntax

assert_win_unchange[#(severity_level,width, options, msg)] inst_name (clk, reset_n, start_event, test_expr, end_event);

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
start_event	Starting event that triggers monitoring of the *check_expr*
test_expr	Expression being verified at the positive edge of *clk.*
end_event	Event that terminates monitoring of the *start_event.*

The assert_win_unchange assertion should be used in circuits to ensure that after a specified initial event, a particular variable or expression will remain unchanged after the *start_event* and before the *end_event.* Common uses for assert_win_unchange include:

- verification that non-deterministic multicycle operations with enabling conditions will always work with the same data
- verification of synchronizing conditions requiring that data is stable after a specified initial triggering event and before an ending condition takes place.

For instance, a bus transaction will occur without any bus interrupts; or a memory write command will not occur if we are in a memory read cycle.

In Example A-13, assume that for the multi-cycle divider operation to work properly, the value of signal *a* must remain unchanged for the duration of the operation. Completion is signaled by the signal *done.*

Example A-13 assert_win_unchange

```
module division_with_check (clk,reset_n,a,b,start,done);
   input clk, reset_n;
   input [15:0] a,b;
   input start;
   output done;

   wire [15:0] q,r;

   div16 div01 (clk, reset_n, start, a, b, q, r, done);

  assert_win_unchange #(0,16) div_win_unchange_a (clk, reset_n, start,
a, done);

endmodule
```

A.8 Checking time bounded windows

In addition to specifying temporal properties within event bounded windows, many properties are bounded with cycle related time bounded windows. In this section we introduce a set of OVL monitors that allow the design to specify time bounded window assertions.

A.8.1 assert_change

The assert_change assertion continuously monitors the *start_event* at every positive edge of the triggering event or clock *clk*. When the *start_event* evaluates TRUE, the assertion monitor ensures that the *test_expr* changes values (as sampled on a clock edge) at some point within the next *num_cks* number of clocks. As soon as the *test_expr* evaluates TRUE, this assertion is satisfied; and checking is discontinued for the remaining *num_cks* number of clocks.

Syntax ***assert_change [#(severity_level, width, num_cks, flag, options, msg)] inst_name (clk, reset_n, start_event, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
num_cks	The number of clocks for *test_expr* to change its value before an error is triggered after *start_event* is asserted.
flag	0 - Ignores any asserted *start_event* after the first one has been detected (default); 1 - Re-start monitoring *test_expr* if *start_event* is asserted in any subsequent clock while monitoring *test_expr;* 2 - Issues an error if an asserted *start_event* occurs in any clock cycle while monitoring *test_expr.*
options	Vendor options. Currently, the only supported option is *options*=1 , which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
start_event	Starting event that triggers monitoring of the *test_expr.*
test_expr	Expression being verified at the positive edge of *clk.*

The assert_change assertion should be used in circuits to ensure that after a specified initial event, a particular variable or expression will change. Common uses for assert_change include:

- verification that synchronization circuits respond after a specified initial stimuli. For example, in protocol verification, this assertion may be used to check that after a *request* an *acknowledge* will occur within a specified number of cycles.

- verification that finite-state machines (FSM) change state, or will go to a specific state, after a specified initial stimuli.

In Example A-14, the module *synchronizer_with_bug* is designed to respond by asserting *out* after *count_max* cycles *sync* was asserted. Note in the accompanying figure, however, that *out* is not asserted until after the trigger of *clk*. In this figure, the waveform is shown until the error is triggered by the assertion library. At this point, the simulation is aborted.

Example A-14 assert_change

```
module synchronizer_with_bug (clk, reset_n, sync,
  count_max, out);

  input clk, reset_n, sync;
  input [3:0] count_max;
  output out;

  reg out;
  reg [3:0] count;

  always @(posedge clk) begin
    if (reset_n == 0) begin
      out <= 0;
      count <= 0;
    end
    else if (count != 0) begin
      count <= count - 1;
      if (count == 1) out <= 1;
    end
    else if (sync == 1) count <= count_max;
    else if (out == 1) out <= 0;
  end

  // out must change values 3 cycles after sync
  assert_change #(0,1,3,0) synch_test (clk,reset_n,(sync == 1), out);
endmodule
```

A.8.2 assert_unchange

The assert_unchange assertion continuously monitors the *start_event* at every positive edge of the triggering event or clock *clk*. When *start_event* evaluates TRUE, the assertion monitor ensures that the *test_expr* will not change values within the next *num_cks* number of clocks.

Syntax ***assert_unchange [#(severity_level, width, num_cks, flag, options, msg)] inst_name (clk,reset_n, start_event, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
num_cks	For this number of clocks after *start_event* is asserted, *test_expr* must remain unchanged. Otherwise, an error is triggered.
flag	0 - Ignores any asserted *start_event* after the first one has been detected (default); 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 cycle while monitoring *test_expr*.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
start_event	Starting event that triggers monitoring of the *check_expr*
test_expr	Expression being verified at the positive edge of *clk*.

The assert_unchange assertion should be used in circuits to ensure that after a specified initial event, a particular variable or expression will remain unchanged for some number of clocks. Common uses for assert_unchange include:

- verification that multicycle operations with enabling conditions will always work with the same data
- verification that single cycle operations will operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require data is stable after a specified initial triggering event

In Example A-15, we assume that for the multi-cycle divider operation to work properly, the value of signal *a* must remain unchanged for the duration of the operation that, in this example, is 16 cycles.

Example A-15 assert_unchange

```
module division_with_check (clk,reset_n,a,b,start,done);
  input clk, reset_n;
  input [15:0] a,b;
  input start;
  output done;

  wire [15:0] q,r;

  div16 #(16) div01 (clk, reset_n, start, a, b, q, r, done);

  assert_unchange #(0,16,16) div_unchange_a                    (clk,
reset_n,start == 1, a);
endmodule
```

A.9 Checking state transitions

The OVL provides a set of monitors that can be used to insure proper sequencing of finite state machines (FSM). For example, if an FSM is in a specific state, then the OVL assert_no_transition can be used to monitor an illegal transition.

A.9.1 assert_no_transition

The assert_no_transition assertion continuously monitors the *test_expr* variable at every positive edge of the triggering event or clock *clk*. When this variable evaluates to the value of *start_state,* the monitor ensures that *test_expr* will never transition to the value of *next_state*. The *width* parameter defines the size (that is, number of bits) of the *test_expr*.

Syntax

assert_no_transition [#(severity_level, width, options, msg)]
inst_name (clk, reset_n, test_expr, start_state, end_state);

severity_level	Severity of the failure with default value of 0.
width	Width of state variable *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.

clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
test_expr	State variable representing finite-state machine (FSM) being checked at the positive edge of *clk*.
start_state	Triggering state of *test_expr*.
end_state	Invalid state for the machine represented by *test_expr* when traversed from state *start_state*.

The assert_no_transition assertion should be used in control circuits, especially FSMs, to ensure that invalid transitions are never triggered.

Please note: *start_state* and *end_state* include any valid Verilog expression. As a result, multiple transitions can be specified by encoding the transitions in these variables. Please refer to the following example.

Example A-16 assert_no_transition

```
module counter_09_or_0F              (reset_n,clk,count,sel_09);
 input reset_n,clk, sel_09;
 output [3:0] count;
 reg [3:0] count;

 always @ (posedge clk)
  if (reset_n==0 || count==4'd9 && sel_09==1'b1)
    count <= 4'd0;
  else
    count <= count + 1;
 assert_no_transition #(0,4) valid_count
          (clk, reset_n, count, 4'd9,                (sel_09 == 1)
? 4'd10 : 4'd0);
endmodule
```

A.9.2 assert_transition

The assert_transition assertion continuously monitors the *test_expr* variable at every positive edge of the triggering event or clock *clk*. When *test_expr* evaluates to the value of *start_state,* the assertion monitor ensures that if *test_expr* changes value, then it will change to the value of *next_state*. The *width* parameter defines the size (that is, number of bits) of the *test_expr*.

Syntax *assert_transition [#(severity_level, width, options, msg)]*
inst_name (clk, reset_n, test_expr, start_state, end_state);

severity_level	Severity of the failure with default value of 0.
width	Width of state variable *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of reset_n of a global reference to reset_n).
test_expr	State variable representing finite-state machine (FSM) being checked at the positive edge of *clk*.
start_state	Triggering state of *test_expr*.
end_state	Next valid state for machine represented by *test_expr* when traversed from state *start_state*.

The assert_transition assertion should be used in control circuits, especially FSMs, to ensure that required transitions are triggered.

Please note *start_state* and *end_state* are verification events that can be represented by any valid Verilog expression. As a result, multiple transitions can be specified by encoding the transitions in these variables.

Example A-17 assert_transition

```
module counter_09_or_0F                (reset_n,clk,count,sel_09);
input reset_n, clk, sel_09;
output [3:0] count;
reg [3:0] count;

always @(posedge clk)
  if (reset_n==0 || count==4'd9 && sel_09==1'b1)
    count <= 4'd0;
  else
    count <= count + 1;

assert_no_transition #(0,4) valid_count
    (clk, reset_n, count, 4'd9,
        (sel_09 == 1'b0)? 4'd10 : 4'd0);
endmodule
```

APPENDIX

B

PSL PROPERTY SPECIFICATION LANGUAGE

B.1 Introduction to PSL

PSL (Property Specification Language) is a formal property language created by Accellera that is compatible with any HDL. This language was based on the Sugar property language originally developed by IBM. A specification written in PSL is both easy to read and mathematically precise, which makes it ideal for both documentation and verification. Thus, it is ideal for specifying hardware properties. An important use of PSL is as input to verification tools for *informal* dynamic verification (for example, simulation), as well as *formal* static verification (for example, property checkers).

Unlike the OVL set of monitors and the SystemVerilog assertion construct, which are used predominantly *during* RTL implementation, the PSL property language is suited for specifying architectural properties prior to *and* during RTL implementation. In addition, as a declarative form of specification, PSL is also suited for specifying interface properties during block-level refinement from an architectural model. Finally, the expressiveness of PSL makes it excellent for capturing implementation assertions and boundary assumptions during RTL implementation.

PSL is a large and expressive property language. However, while every PSL property can be checked in formal verification, not every PSL property can be checked in simulation. A subset of the PSL language lends itself to automatically generating simulation checkers (also referred to as monitors), which then can be used to

check specific design properties during simulation. Hence, we are limiting our PSL introduction to a subset of the language that can be checked in both simulation *and* formal verification. A full description of the PSL language can be found in the Accellera PSL Formal Property Language Reference Manual [Accellera PSL-1.1 2004].

B.2 Operators and keywords

Table B.1 shows the PSL keywords, which are case-sensitive. This appendix discusses the keywords marked in **bold** type.

Table B.1 PSL Keywords

A AF AG AX **abort** **always** **and**[1] **assert** **assume** assume_guarante e **before** **before!** **before!_** **before_** **boolean** **clock** const **countones** **cover** default	E EF EG EX **endpoint** **eventually!** F fairness **fell** **forall** G in inf inherit is[2] **isunknown**	**never** **next** **next!** next_a next_a! next_e next_e! next_event next_event! next_event_a next_event_a! next_event_e next_event_e! **not**[3] **onehot** **onehot0** or[4] **property** **prev** **report** **rose**	**sequence** **stable** strong to[5] **U** union **until** **until!** **until!_** **until_** vmode vprop vunit W **within** X X!

1.and is a keyword only in the VHDL flavor.
2.is is a keyword only in the VHDL flavor.
3.not is a keyword only in the VHDL flavor.
4.or is a keyword only in the VHDL flavor.
5.to is a keyword only in the VHDL flavor.

Table B.2 lists the operators available in PSL from highest to lowest precedence. This appendix addresses the operators in bold. Those in standard weight are beyond the scope of this appendix.

Additional information on these operators can be found in the PSL Language Reference Manual [Accellera PSL-1.1 2004].

Table B.2 PSL operator precedence

Operator class	/ Associativity /	Operators
HDL operators *(highest precedence)*		
Union operator	left	`union`
Clocking	left	`@`
SERE repetition	left	`[* ]  [+ ]  [= ]  [-> ]`
Sequence AND	left	`&  &&`
Sequence OR	left	`|`
Sequence within	left	`within`
Sequence fusion	left	`:`
Sequence concatenation	left	`;`
FL termination	left	`abort`
FL occurrence	right	`always never eventually! next*` `X  G  F  X!`
FL bounding	right	`U  W` `until*  before*`
Sequence implication	right	`|->  |=>`
Boolean implication *(lowest precedence)*	right	`->  <->`

PSL also defines a set of Options Branching Extension (OBE) operators, which are beyond the scope of this appendix. Please refer to the PSL LRM [Accellera PSL-1.1 2004] for details.

B.3 PSL Boolean layer

The *Boolean layer* of PSL provides for any Boolean expression valid within the language flavor of PSL being used (that is, Verilog or VHDL Boolean expressions). The result of the Boolean expression is a singular value of TRUE or FALSE. This is equivalent to a condition being evaluated within an `if` statement within Verilog or VHDL. Additionally, PSL provides a number of predefined functions that return Boolean values.

B.4 PSL Temporal Layer

The *temporal layer* is the heart of the PSL language; it describes properties of the design that have complex temporal relationships. Thus, unlike simple properties such as "signals a and b are mutually exclusive", the temporal layer allows PSL to describe relationships between signals, such as *"if signal c is asserted, then signal d must be asserted before signal e is asserted, but no more than 8 clock cycles later"*.

Sequence Extended Regular Expressions (SERE)

PSL's *temporal layer* is based on an extension of regular expressions, called *Sequence Extended Regular Expressions* (SEREs), which is in many cases more concise and easier to read and write. The simplest SERE is a Boolean expression describing a Boolean event. More complicated SEREs are built from Boolean expressions using various SERE composition operators. A SERE is not a property on its own; it is a building block of a property; that is, properties are built from temporal operators applied to SEREs and Boolean expressions.

a sequence *holds*

Within this section, we refer to a sequence *holding*. This term indicates that the behavior described by the sequence or property is actually seen.

This section is organized by describing the composition operators first, followed by the temporal operators.

B.4.1 SERE

Sequential Extended Regular Expressions (*SERE*s), shown in Syntax B-1, describe single- or multi-cycle behavior built from a series of Boolean expressions.

The most basic SERE is a Boolean expression.

More complex sequential expressions are built from Boolean expressions using various SERE operators. These operators are described in the subsections that follow.

Syntax B-1 Sequential Extended Regular Expression (SERE)

```
SERE ::=
      Boolean
    | Sequence
```

B.4.2 Sequence

A sequence is a SERE that can appear at the top level of a declaration, directive, or property.

Syntax B-2 Sequence

```
Sequence  ::=
     Repeated_SERE
   | Braced_SERE
   | Clocked_SERE
```

B.4.3 Braced SERE

A SERE enclosed in braces is another form of sequence..

Syntax B-3 Sequence

```
Braced_SERE  ::=
     { SERE }
```

B.4.4 SERE concatenation (;) operator

The SERE concatenation operator (;) describes a sequence of events by specifying two sequences of events that must hold one after the other. That is, the second SERE starts one cycle after the first SERE completes.

The right operand of '*;*' is a SERE that is required to hold after the left operand completes. If either operand describes the empty sequence, then the concatenation holds if and only if the non-empty sequence holds.

Syntax B-4 Concatenation of sequences

```
SERE ::=
          SERE ; SERE
```

B.4.5 Consecutive repetition ([*]) operator

The SERE *consecutive repetition* operator ([*]) describes repeated consecutive concatenation of the same SERE.

Note the RANGE_SYM is ':' for Verilog and 'to' for VHDL.

Syntax B-5 SERE **consecutive repetition**

```
SERE ::=
        SERE [ * [ Count ] ]
      | [*[ Count ] ]
      | SERE [ + ]
      | [+]

Count ::=
        Number | Range
Range ::=
        LowBound RANGE_SYM HighBound
LowBound ::=
        Number | MIN_VAL
HighBound ::=
        Number | MAX_VAL
```

The first operand of consecutive repetition is a SERE that is required to hold several consecutive times. The second operator is a number (or a range of numbers) that describes the number of times the SERE is repeated.

If the high value of the range is not specified (or is specified as `inf`), then the SERE must hold for at least the low value of the range. If the low value of the range is not specified (or is specified as '0') then the SERE must hold no more than the high value of the range times. If neither of the range values is defined, then the SERE is allowed to hold any number of times including zero, that is, the empty sequence is allowed.

When there is no SERE operand, and only a number (or a range), then the resulting SERE describes any sequence whose length is described by the second operand as above.

The notation `'+'` is a shortcut for a repetition of one or more times.

Syntax B-6 SERE **consecutive repetition**

```
SERE_A [ Number_n]              // SERE_A repeats exactly Number_n times

SERE_A [Number_n : Number_m]    // SERE_A repeats at least Number_n times
                                // no more than Number_m times

SERE_A [0 : Number_m]           // SERE_A is either empty or repeats no
                                // more than Number_m times

SERE_A [Number_n : inf]         // SERE_A repeats at least Number_n
                                // times

SERE_A [0 : inf]                // SERE_A is either empty or repeats some
                                // undefined number of times

SERE_A [+]                      // SERE_A evaluates one or more times

[* Number_n]                    // the sequence is of length Number_n

[* Number_n : Number_m]         // the length of the sequence is a number
                                // between Number_n and Number_m

[* 0 : Number_m]                // an empty sequence or a sequence of
                                // length Number_m at most

[* Number_n : inf]              // a sequence is of length Number_n at
                                // least

[* 0 : inf]                     // any sequence of events

[*]                             // any sequence of events (including the
                                // empty sequence)

[+]                             // any sequence of events of length one
                                // at least
```

B.4.6 Nonconsecutive repetition ([=]) operator

Nonconsecutive repetition allows for space between the repetition terms. The syntax for nonconsecutive repetition is the same as for consecutive repetition except the '*' operator is replaced with the '=' operator.

Note the `RANGE_SYM` is ':' for Verilog and '`to`' for VHDL.

Syntax B-7 SERE nonconsecutive repetition

```
SERE ::=
        Boolean [ = Count ]

Count ::=
        Number | Range
Range ::=
        LowBound RANGE_SYM HighBound
LowBound ::=
        Number | MIN_VAL
HighBound ::=
        Number | MAX_VAL
```

B.4.7 Goto repetition ([->]) operator

Goto repetition allows for space between the repetition of the terms. The repetition ends on the Boolean expression being found. This facilitates searching for a particular expression and then continuing the sequence at the point it is found.

Note the `RANGE_SYM` is ':' for Verilog and '`to`' for VHDL.

Syntax B-8 Goto repetition of a sequence

```
SERE ::=
        Boolean [ -> [ positive Count ] ]

Count ::=
        Number | Range
Range ::=
        LowBound RANGE_SYM HighBound
LowBound ::=
        Number | MIN_VAL
HighBound ::=
        Number | MAX_VAL
```

B.4.8 Sequence fusion (:) operator

The *sequence fusion* operator specifies that two sequences overlap by one cycle. In this case, the second sequence starts the cycle that the first sequence ends.

Syntax B-9 Sequence fusion operator

```
SERE ::= SERE : SERE
```

B.4.9 Sequence non-length-matching (&) operator

The *sequence non-length-matching and* operator specifies that two sequences must hold and complete in different cycles (that is, they may be of different lengths).

Syntax B-10 Sequence non-length-matching and operator

```
SERE ::= SERE & SERE
```

B.4.10 Sequence length-matching (&&) operator

The *sequence length-matching and* operator specifies that two sequences must hold and complete in the same cycle (that is, they must be of the same length).

Syntax B-11 Sequence length-matching and operator

```
SERE ::= SERE && SERE
```

B.4.11 Sequence or (|) operator

The *sequence or* operator specifies that one of two alternative sequences must hold.

Syntax B-12 Sequence or operator

```
SERE ::= SERE | SERE
```

B.4.12 until* sequence operators

The until* operators (until, until!, until!_, and until_) specify that a property holds until a second property holds. The until* operators provide another way to move forward, this time while putting a requirement on the cycles in which we are moving.

Syntax B-13 until* operators

```
FL_Property ::=
        FL_Property until! FL_Property
      | FL_Property until FL_Property
      | FL_Property until!_ FL_Property
      | FL_Property until_ FL_Property
```

weak versus strong operators

The different flavors of this operator specify strong (until! and until!_) or weak (until and until_) operators. *Strong operators* require the terminating property to eventually occur, while *weak operators* do not. The inclusive operators (until_ and until!_) specify that the property must hold up to and including the cycle in which the terminating property holds, whereas the non-inclusive operators (until and until!) require the property to hold up to, but not necessarily including, the cycle in which the terminating property holds.

B.4.13 within sequence operators

The SERE within operator (**within**), shown in Syntax B-14, constructs a SERE in which the second SERE holds at the current cycle, and the first SERE starts at or after the cycle in which the second starts, and completes at or before the cycle in which the second completes.

Syntax B-14 within* operators

```
SERE ::=
   SERE within SERE
```

B.4.14 next operator

The next operators allow us to be more specific about the timing; it takes us forward one clock cycle. The `next` operator comes in both weak (next) and strong (next!) forms. If the number parameter is present, it indicates the cycle at which the property on the right hand side holds.

For further information on the remaining family of next* operators, please refer to the PSL LRM.

Syntax B-15 next* operators

```
FL_Property ::=
        next FL_Property
      | next! FL_Property
      | next [ Number ] ( FL_Property )
      | next! [ Number ] ( FL_Property )
```

B.4.15 eventually! operator

While the next operator moves us forward exactly one cycle, the eventually! operator allows us to move forward without specifying exactly when to stop. This operator is a strong operator, which requires that the ending property or sequence actually occur.

Syntax B-16 eventually! operators

```
FL_Property ::=
        eventually! FL_Property
      | eventually! Sequence
```

B.4.16 before* operators

The before* operators provide an easy way to state that some signal must be asserted before some other signal.

Syntax B-17 before operator

```
FL_Property ::=
        FL_Property before! FL_Property
      | FL_Property before FL_Property
      | FL_Property before!_ FL_Property
      | FL_Property before_ FL_Property
```

weak versus strong operators

The different flavors of this operator specify strong (before! and before!_) or weak (before and before_) operators. Strong operators require the ending condition to eventually occur, while weak operators do not. If the ending condition overlaps with the rightmost operand of the sequence, use the inclusive operators (before_ and before!_). Use the non-inclusive operators (before

and before!) to require the rightmost operand of the sequence to complete the cycle before the terminating condition.

B.4.17 abort operator

The abort operator provides a way to lift any future obligations of a property when some Boolean condition is observed.

Syntax B-18 abort operator

FL_Property ::=
FL_Property **abort** Boolean

The `abort` operator is reminiscent of the `until` operator, but there is an important difference. Both "`f abort b`" and "`f until b`" specify that we should stop checking when `b` occurs. However, the abort operator removes future obligations of `f`, while the until operator does not.

B.4.18 Endpoint declaration

An endpoint for a sequence is defined in PSL with a named endpoint declaration. The name of an endpoint cannot be the same name as other named PSL declarations.

Syntax B-19 Endpoint declaration

Endpoint_Declaration ::=
endpoint Name [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;**

B.4.19 Suffix implication operators

A SERE is not a PSL property in and of itself. In order to use a SERE to build a PSL property, we link a SERE with another PSL property or with another SERE using an implication operator. An implication operator can be read as "whenever we have a sequenceA, we expect to see sequenceB."

Syntax B-20 Suffix impliciaton

```
FL_Property ::=
        Sequence ( FL_Property )
      | Sequence |-> Sequence [ ! ]
      | Sequence |=> Sequence [ ! ]
```

weak versus strong operators

The strong implication operators specify that the rightmost sequence must complete, whereas the weak implication operators do not. The suffix implication specifies that the rightmost sequence begins on the cycle in which the leftmost sequence ends. The suffix next implication specifies that the rightmost sequence begins on the cycle after the leftmost sequence ends.

B.4.20 Logical implication operator

The logical `if` implication operator specifies that if the leftmost property holds, then the rightmost property must hold.

Syntax B-21 Logical implication

```
FL_Property ::=
        FL_Property -> FL_Property
```

B.4.21 always temporal operator

The always operator specifies one of the simplest temporal properties, which states that some Boolean expression must hold at all times.

Syntax B-22 always

```
FL_Property ::=
        always FL_Property
```

B.4.22 never temporal operator

The never operator allows us to specify an invariant property, which specifies conditions that must *never* hold.

Syntax B-23 never

FL_Property ::=
never FL_Property

B.5 PSL properties

B.5.1 Property declaration

The building blocks of Boolean expressions and sequences described in previous sections create PSL properties. Properties capture the temporal relationships between these building blocks. Properties are grouped using parentheses (( )).

B.5.2 Named properties

PSL allows us to name property definitions as shown in Syntax B-24. Note that DEF_SYM is '=' for Verilog and '`is`' for VHDL.

Syntax B-24 Named property

Property_Declaration ::=
property Name [**(** Formal_Parameter_List **)**] DEF_SYM Property **;**

B.5.3 Property clocking

PSL allows us to declare a default clock, explicitly declare a clock associated with a property, or declare that "clock cycle" and "next

point in time" are equivalent. A clock expression is any Boolean expression. A PSL property may refer to multiple clocks.

Syntax B-25 Default clock declaration

```
PSL_Declaration ::=
        Clock_Declaration
Clock_Declaration ::=
        default clock DEF_SYM Clock_Expression ;(see B.8.3.2)
```

Syntax B-26 Clocked property or SERE

```
SERE ::=
        SERE @ Clock_Expression

FL_Property ::=
        FL_Property @ Clock_Expression
```

B.5.4 forall property replication

PSL provides an easy way to replicate properties that are the same except for specific parameters. Example B-1 shows the syntax for the `forall` operator.

Example B-1 forall property replicatation syntax

```
Property ::=
    Replicator Property
Replicator ::=
    forall Name [ IndexRange ] in ValueSet :
IndexRange ::=
    LEFT_SYM finite_Range RIGHT_SYM
Flavor Macro LEFT_SYM =
    Verilog: [ / VHDL: ( / GDL: (
Flavor Macro RIGHT_SYM =
    Verilog: ] / VHDL: ) / GDL: )
ValueSet ::=
      { ValueRange { , ValueRange } }
    | boolean
ValueRange ::=
      Value
    | finite_Range
Range ::=
    LowBound RANGE_SYM HighBound
```

B.6 The verification layer

The *verification layer* tells the verification tools what to do with the properties described by the temporal layer. For example, the Verification layer contains directives that tell a tool to assert a property (that is, to verify that it holds) or to check that a specified sequence is covered by some test case.

B.6.1 assert directive

The assert directive verifies that a property holds. If the property does not hold, an error is raised.

Syntax B-27 assert directive

Assert_Statement ::=
assert Property **;** (see B.8.3.3)

B.6.2 assume directive

The assume directive defines constraints to guide a verification tool.

Syntax B-28 assume directive

Assume_Statement ::=
assume Property **;** (see B.8.3.3)

B.6.3 cover directive

The cover directive instructs the tool to indicate if a property has been exercised by the test suite or given constraints.

Syntax B-29 cover directive

Cover_Statement ::=
cover Sequence **;** (see B.8.3.2)

B.7 The modeling layer

The *modeling layer* models the behavior of design inputs (for tools such as formal verification tools that do not use test cases), and auxiliary hardware that is not part of the design but is needed for verification. This layer is, for the most part, outside the scope of this appendix. However, in this section we discuss a few useful functions that are defined by the modeling layer..

Syntax B-30 Built in functions

Built_In_Function_Call ::=
  prev (AnyType [, Number])
 | **next** (AnyType)
 | **stable** (AnyType)
 | **rose** (Bit)
 | **fell** (Bit)
 | **isunknown** (BitVector)
 | **countones** (BitVector)
 | **onehot** (BitVector)
 | **onehot0** (BitVector)

B.7.1 prev()

The built-in function `prev()` takes an expression of any type as argument and returns a previous value of that expression. With a single argument, the built-in function `prev()` gives the value of the expression in the previous cycle, with respect to the clock of its context. If a second argument is specified and has the value `i`, the built-in function `prev()` gives the value of the expression in the i^{th} previous cycle, with respect to the clock of its context.

The clock context may be provided by the PSL property in which the function call is nested, or by a relevant default clock declaration. If the context does not specify a clock, the relevant clock is that corresponding to the granularity of time as seen by the verification tool.

B.7.2 next()

The built-in function `next()` gives the value of a signal of any type at the next cycle, with respect to the finest granularity of time as seen by the verification tool. In contrast to the built-in functions `prev()`, `stable()`, `rose()`, and `fell()`, the function `next()` is not affected by the clock of its context.

B.7.3 stable()

The built-in function `stable()` takes an expression of any type as argument and produces a Boolean result that is true if the argument's value is the same as it was at the previous cycle, with respect to the clock of its context.

The clock context may be provided by the PSL property in which the function call is nested, or by a relevant default clock declaration. If the context does not specify a clock, the relevant clock is that corresponding to the granularity of time as seen by the verification tool.

The function `stable()` can be expressed in terms of the built-in function `prev()` as follows: `stable(e)` is equivalent to the Verilog or SystemVerilog expression (`prev(e) == e`), and is equivalent to the VHDL expression (`prev (e) = e`), where `e` is any expression. The function `stable()` can be used anywhere a Boolean is required.

B.7.4 rose()

The built-in function `rose()` takes a Bit expression as argument and produces a Boolean result that is true if the argument's value is 1 at the current cycle and 0 at the previous cycle, with respect to the clock of its context, otherwise it is false.

The clock context may be provided by the PSL property in which the function call is nested, or by a relevant default clock declaration. If the context does not specify a clock, the relevant clock is that corresponding to the granularity of time as seen by the verification tool.

The function `rose()` can be expressed in terms of the built-in function `prev()` as follows: `rose(b)` is equivalent to the Verilog or SystemVerilog expression (`prev(b)==1'b0 && b==1'b1`), and is equivalent to the VHDL expression (`prev(b)='0' and b='1'`), where `b` is a Bit signal. The function `rose(b)` can be used anywhere a Boolean is required.

B.7.5 fell()

The built-in function `fell()` takes a Bit expression as argument and produces a Boolean result that is true if the argument's value is 0 at the current cycle and 1 at the previous cycle, with respect to the clock of its context, otherwise it is false.

The clock context may be provided by the PSL property in which the function call is nested, or by a relevant default clock declaration. If the context does not specify a clock, the relevant clock is that corresponding to the granularity of time as seen by the verification tool.

The function `fell()` can be expressed in terms of the built-in function `prev()` as follows: `fell(b)` is equivalent to the Verilog or SystemVerilog expression (`prev(b) ==1'b1 && b==1'b0`), and is equivalent to the VHDL expression (`prev(b)='1' and b='0'`), where `b` is a Bit signal. The function `fell(b)` can be used anywhere a Boolean is required.

B.7.6 isunknown()

The built-in function `isunknown()` takes a BitVector as argument. It returns True if the argument contains any bits that have "unknown" values (i.e., values other than 0 or 1); otherwise it returns False.

Function `isunknown()` can be used anywhere a Boolean is required.

B.7.7 countones()

The built-in function `countones()` takes a BitVector as argument. It returns a count of the number of bits in the argument that have the value 1.

Bits that have unknown values are ignored.

B.7.8 onehot(), onehot0()

The built-in function `onehot()` takes a BitVector as argument. It returns True if the argument contains exactly one bit with the value 1; otherwise it returns False.

The built-in function `onehot0()` takes a BitVector as argument. It returns True if the argument contains at most one bit with the value 1; otherwise it returns False.

For either function, bits that have unknown values are ignored.

Functions `onehot()` and `onehot0()` can be used anywhere a Boolean is required

B.8 BNF

The appendix summarizes the syntax .

B.8.1 Meta-syntax

The formal syntax described in this standard uses the following extended Backus-Naur Form (BNF).

a) The initial character of each word in a nonterminal is capitalized. For example:

    PSL_Statement

    A nonterminal can be either a single word or multiple words separated by underscores. When a multiple-word nonterminal containing underscores is referenced within the text (e.g., in a statement that describes the semantics of the corresponding syntax), the underscores are replaced with spaces.

b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. These words appear in a larger font for distinction. For example:

    vunit (;

c) The : : = operator separates the two parts of a BNF syntax definition. The syntax category appears to the left of this operator and the syntax description appears to the right of the operator. For example, item d) shows three options for a *VUnitType.*

d) A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself. For example:

VUnitType ::= **vunit | vprop | vmode**

e) Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

Sequence_Declaration ::= **sequence** Name [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;**

indicates *Formal_Parameter_List* is an optional syntax item for *Sequence_Declaration,* whereas

| SERE **[** * [Range] **]**

indicates that (the outer) square brackets are part of the syntax for this SERE, while *Range* is optional.

f) Braces enclose a repeated item unless it appears in boldface, in which case it stands for itself. A repeated item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

Formal_Parameter_List ::= Formal_Parameter { **;** Formal_Parameter }
Formal_Parameter_List ::= Formal_Parameter | Formal_Parameter_List **;** Formal_Parameter

g) A comment in a production is preceded by a colon (:) unless it appears in boldface, in which case it stands for itself.

h) If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *vunit*_Name is equivalent to Name.

i) Flavor macros, containing embedded underscores, are shown in uppercase. These reflect the various HDLs that can be used within the PSL syntax and show the definition for each HDL. The general format is the term `Flavor`

`Macro`, then the actual *macro name,* followed by the = operator, and, finally, the definition for each of the HDLs. For example:

> Flavor Macro PATH_SYM = Verilog: **.** / VHDL: **:** / GDL: **/**

shows the *path symbol* macro. See 4.3.2 for further details about *flavor macros.*

The main text uses *italicized* type when a term is being defined, and `monospace` font for examples and references to constants such as `0`, `1`, or `x` values.

B.8.2 HDL Dependencies

PSL depends upon the syntax and semantics of an underlying hardware description language. In particular, PSL syntax includes productions that refer to nonterminals in SystemVerilog, Verilog, VHDL, or GDL. PSL syntax also includes Flavor Macros that cause each flavor of PSL to match that of the underlying HDL for that flavor.

For SystemVerilog, the PSL syntax refers to the following nonterminals in the Accellera SystemVerilog version 3.1 a syntax:

- module_or_generate_item_declaration
- module_or_generate_item
- list_of_variable_identifiers
- identifier
- expression
- constant_expression

For Verilog, the PSL syntax refers to the following nonterminals in the IEEE 1364-2001 Verilog syntax:

- module_or_generate_item_declaration
- module_or_generate_item
- list_of_variable_identifiers
- identifier
- expression
- constant_expression
- net_declaration
- reg_declaration
- integer_declaration

For VHDL, the PSL syntax refers to the following nonterminals in the IEEE 1076-1993 VHDL syntax:

- block_declarative_item
- concurrent_statement
- design_unit
- identifer
- expression

For GDL, the PSL syntax refers to the following nonterminals in the GDL syntax:

- module_item_declaration
- module_item
- module_declaration
- identifer
- expression

B.8.2.1 Verilog Extensions

For the Verilog flavor, PSL extends the forms of declaration that can be used in the modeling layer by defining two additional forms of type declaration. PSL also adds an additional form of expression for both Verilog and VHDL flavors.

```
Extended_Verilog_Declaration ::=
        Verilog_module_or_generate_item_declaration
      | Extended_Verilog_Type_Declaration

Extended_Verilog_Type_Declaration ::=
        integer Integer_Range list_of_variable_identifiers ;
      | struct { Declaration_List } list_of_variable_identifiers
        ;

Integer_Range ::=
        ( constant_expression : constant_expression )

Declaration_List ::=
        HDL_Variable_or_Net_Declaration {
        HDL_Variable_or_Net_Declaration }

HDL_Variable_or_Net_Declaration ::=
        net_declaration
      | reg_declaration
```

| integer_declaration

B.8.2.2 Flavor macros

Flavor Macro DEF_SYM =
SystemVerilog: = / Verilog: = / VHDL: **is** / GDL: **:=**
Flavor Macro RANGE_SYM =
SystemVerilog: **:** / Verilog: **:** / VHDL: **to** / GDL: **..**
Flavor Macro AND_OP =
System Verilog: **&&** / Verilog: **&&** / VHDL: **and** /
GDL: **&**
Flavor Macro OR_OP =
SystemVerilog: || / Verilog: || / VHDL: **or** / GDL: |
Flavor Macro NOT_OP =
SystemVerilog: **!** / Verilog: **!** / VHDL: **not** / GDL: **!**
Flavor Macro MIN_VAL =
SystemVerilog: **0** / Verilog: **0** / VHDL: **0** / GDL: *null*
Flavor Macro MAX_VAL =
SystemVerilog: **$** / Verilog: **inf** / VHDL: **inf** / GDL: *null*
Flavor Macro HDL_EXPR =
SystemVerilog: SystemVerilog_Expression
/ Verilog: Verilog_Expression
/ VHDL: Extended_VHDL_Expression
/ GDL: GDL_Expression
Flavor Macro HDL_CLK_EXPR =
SystemVerilog: SystemVerilog_Event_Expression /
Verilog: Verilog_Event_Expression / VHDL:
VHDL_Expression / GDL: GDL_Expression
Extended_VHDL_Expression
/ GDL: *GDL*_Expression
Flavor Macro HDL_UNIT =
SystemVerilog: SystemVerilog_module_declaration /
Verilog: *Verilog*_module_declaration / VHDL:
*VHDL*_design_unit / GDL: *GDL*_module_declaration
Flavor Macro HDL_DECL =
SystemVerilog:
SystemVerilog_module_or_generate_item_declaration
/ Verilog: Extended_Verilog_Declaration
/ VHDL: *VHDL*_block_declarative_item
/ GDL: *GDL*_module_item_declaration
Flavor Macro HDL_STMT =
SystemVerilog:
SystemVerilog_module_or_generate_item
/ Verilog: *Verilog*_module_or_generate_item
/ VHDL: *VHDL*_concurrent_statement
/ GDL: *GDL*_module_item
Flavor Macro HDL_RANGE=

VHDL: range_attribute_name
Flavor Macro LEFT_SYM =
SystemVerilog: **[** / Verilog: **[** / VHDL: **(** / GDL: **(**
Flavor Macro RIGHT_SYM =
SystemVerilog: **]** / Verilog: **]** /VHDL: **)** / GDL: **)**

B.8.3 Syntax productions

The rest of this section defines the PSL syntax.

B.8.3.1 Verification units

PSL_Specification ::=
{ Verification_Item }
Verification_Item ::=
HDL_UNIT | Verification_Unit
Verification_Unit ::=
VUnitType Name [**(** Hierarchical_HDL_Name **)**] **{**
{ Inherit_Spec }
{ VUnit_Item }
}
VUnitType ::=
vunit | vprop | vmode
Name ::=
HDL_or_PSL_Identifier
Hierarchical_HDL_Name ::=
*module*_Name {Path_Separator *instance*_Name }
Path_Separator ::=
. | /
Inherit_Spec ::=
inherit *vunit*_Name {**,** *vunit*_Name } **;**
VUnit_Item ::=
HDL_DECL
| HDL_STMT
| PSL_Declaration (see B.8.3.2)
| PSL_Directive (see B.8.3.3)

B.8.3.2 PSL declarations

PSL_Declaration ::=
Property_Declaration
| Sequence_Declaration
| Endpoint_Declaration
| Clock_Declaration
Property_Declaration ::=

property Name [**(** Formal_Parameter_List **)**]
DEF_SYM Property**;**

Formal_Parameter_List ::=
Formal_Parameter { **;** Formal_Parameter }

Formal_Parameter ::=
ParamType Name {**,** Name }

ParamType ::=
const | boolean | property | sequence

Sequence_Declaration ::=
sequence Name [**(** Formal_Parameter_List **)**]
DEF_SYM Sequence **;**(see B.8.3.5)

Endpoint_Declaration ::=
endpoint Name [**(** Formal_Parameter_List **)**]
DEF_SYM Sequence **;**(see B.8.3.5)

Clock_Declaration ::=
default clock DEF_SYM Clock_Expression **;**(see B.8.3.7)

Clock_Expression ::=
Boolean
| '(' HDL_CLK_EXPR ')'

Actual_Parameter_List ::=
Actual_Parameter {**,** Actual_Parameter }

Actual_Parameter ::=
Number | Boolean | Property | Sequence (see B.8.3.7) (see B.8.3.7) (see B.8.3.4) (see B.8.3.5)

B.8.3.3 PSL directives

PSL_Directive ::=
[Label:] Verification_Directive

Label ::=
HDL_or_PSL_Identifier

HDL_or_PSL_Identifier ::=
System Verilog_Identifier
| Verilog_Identifier
| VHDL_Identifier
| GDL_Identifier
| PSL_Identifier

Verification_Directive ::=
Assert_Statement
| Assume_Statement
| Assume_Guarantee_Statement
| Restrict_Statement
| Restrict_Guarantee_Statement
| Cover_Statement
| Fairness_Statement

Assert_Statement ::=
assert Property [**report** String] **;**(see B.8.3.4)
Assume_Statement ::=
assume Property **;**(see B.8.3.4)
Assume_Guarantee_Statement ::=
assume_guarantee Property [**report** String] **;**(see B.8.3.4)
Restrict_Statement ::=
restrict Sequence ; (see A.3.5)
Restrict_Guarantee_Statement ::=
restrict_guarantee Sequence [**report** String]; (see A.3.5)
Cover_Statement ::=
cover Sequence [**report** String] **;**(see B.8.3.5)
Fairness_Statement ::=
fairness Boolean**;**
| **Strong fairness** Boolean **,** Boolean **;**(see B.8.3.7)

B.8.3.4 PSL properties

Property ::=
Replicator Property
| FL_Property
| OBE_Property
Replicator ::=
forall Name [IndexRange] **in** ValueSet **:**
IndexRange ::=
LEFT_SYM *finite*_Range RIGHT_SYM
|'(' HDL_RANGE')'
ValueSet ::=
{ ValueRange {, ValueRange } **}**
| **boolean**
ValueRange ::=
Value (see B.8.3.7)
| FiniteRange(see B.8.3.5)
FL_Property ::=
Boolean (see B.8.3.7)
| **(** FL_Property **)**
| Sequence [**!**]
| *property*_Name [**(** Actual_Parameter_List **)**]

| FL_Property @ CLOCK_EXPRESSION
| FL_Property **abort** Boolean
: Logical Operators :

| NOT_OP FL_Property
| FL_Property AND_OP FL_Property
| FL_Property OR_OP FL_Property
:
| FL_Property -> FL_Property
| FL_Property <-> FL_Property
: Primitive LTL Operators :
| **X** FL_Property
| **X!** FL_Property
| **F** FL_Property
| **G** FL_Property
| **[** FL_Property **U** FL_Property **]**
| **[** FL_Property **W** FL_Property **]**
: Simple Temporal Operators:
| **always** FL_Property
| **never** FL_Property
| **next** FL_Property
| **next!** FL_Property
| **eventually!** FL_Property
:
| FL_Property **until!** FL_Property
| FL_Property **until** FL_Property
| FL_Property **until!**_ FL_Property
| FL_Property **until**_ FL_Property
:
| FL_Property **before!** FL_Property
| FL_Property **before** FL_Property
| FL_Property **before!**_ FL_Property
| FL_Property **before**_ FL_Property
: Extended Next (Event) Operators :(see B.8.3.7)
| **X [** Number **] (** FL_Property **)**
| **X! [** Number **] (** FL_Property **)**
| **next [** Number **] (** FL_Property **)**
| **next! [** Number **] (** FL_Property **)**
:(see B.8.3.5)
| **next_a [***finite*_Range **] (** FL_Property **)**
| **next_a! [***finite*_Range **] (** FL_Property **)**
| **next_e [***finite*_Range **] (** FL_Property **)**
| **next_e! [***finite*_Range **] (** FL_Property **)**
:
| **next_event! (** Boolean **) (** FL_Property **)**
| **next_event (** Boolean **) (** FL_Property **)**
| **next_event! (**Boolean**) [***positive*_Number**] (**
FL_Property **)**
| **next_event (** Boolean **) [***positive*_Number **] (**

FL_Property)
:
| **next_event_a!** (Boolean) [*finite_positive*_Range]
(FL_Property)
| **next_event_a** (Boolean) [*finite_positive*_Range] (
FL_Property)
| **next_event_e!** (Boolean) [*finite_positive*_Range]
(FL_Property)
| **next_event_e** (Boolean) [*finite_positive*_Range] (
FL_Property)

: Operators on SEREs :(see B.8.3.5)
| Sequence (FL_Property)
| Sequence |-> FL_Property
| Sequence |=> FL_Property

B.8.3.5 Sequences

Sequence ::=
{ SERE_or_SequenceInstance } [@clock_Boolean]
| [SERE_Element] ContiguousRepeat
| Boolean IndependentRepeat

SEREorSequenceInstance ::=
SERE
| *sequence*_Name [(Actual_Parameter_List)]

ContiguousRepeat ::=
[* [Count]]
| [+]

IndependentRepeat ::=
[= Count]
| [-> [positive_Count]]

Count ::=
Number | Range

Range ::=
LowBound RANGE_SYM HighBound

LowBound ::=
Number | MIN_VAL

HighBound ::=
Number | MAX_VAL

B.8.3.6 Sequential extended regular expressions

SERE ::=
SERE_Element
| SERE SEREOp SERE_Element

SERE_Element ::=
Boolean (see A.3.7)
| Sequence

SEREOp ::= **&&** | **&** | | | **within** | **:** | **;**

B.8.3.7 Forms of expression

Value ::=
Boolean | Number

AnyType ::=
HDL_or_PSL_Expression

Bit ::=
*bit*_HDL_or_PSL_Expression

Boolean ::=
*boolean*_HDL_or_PSL_Expression

BitVector ::=
*bitvector*_HDL_or_PSL_Expression
Number ::=
*numeric*_HDL_or_PSL_Expression

String ::=
*string*_HDL_or_PSL_Expression

HDL_or_PSL_Expression ::=
HDL_Expression
| PSL_Expression
| Built_In_Function_Call
| Union_Expression
| Endpoint_Instance

HDL_Expression ::=
HDL_EXPR

PSL_Expression ::=
Boolean '->' Boolean

| Boolean '<->' Boolean

Built_In_Function_Call ::=
prev (AnyType [, Number])
| **next** (AnyType)
| **Stable** (AnyType)
| **rose** (Bit)
| **fell** (Bit)
| **isunknown** (Bit Vector)
| **countones** (Bit Vector)
| **onehot** (BitVector)
| **onehot0** (BitVector)

Union_Expression ::=
AnyType **union** AnyType

Endpoint_Instance ::=
*endpoint*_Name [(Actual_Parameter_List)]

B.8.3.8 Optional branching extension

OBE_Property ::=
Boolean
| **(** OBE_Property **)**
| *property*_Name [**(** Actual_Parameter_List **)**]
: Logical Operators:
| **!** OBE_Property
| OBE_Property **&** OBE_Property
| OBE_Property | OBE_Property
| OBE_Property -> OBE_Property
| OBE_Property **<->** OBE_Property
: Universal Operators:
| **AX** OBE_Property
| **AG** OBE_Property
| **AF** OBE_Property
| **A** [OBE_Property **U** OBE_Property]
: Existential Operators:
| **EX** OBE_Property
| **EG** OBE_Property
| **EF** OBE_Property
| **E [** OBE_Property **U** OBE_Property **]**

APPENDIX

C

SYSTEMVERILOG ASSERTIONS

C.1 . Introduction to SystemVerilog

SystemVerilog is incorporating the concepts discussed in Chapter 3, “Specifying RTL Properties” on page 57. See the Accellera SystemVerilog 3.1a specification for the full standard of the language.

The BNF described here follows a few conventions.

- Keywords are in bold
- Required syntax characters are marked with single quotes

Production names not found in this text are part of the remainder of SystemVerilog 3.1a BNF.

C.2 Operator and keywords

SystemVerilog is introducing the following operators and keywords. These directly relate to sequences and properties. The SystemVerilog operators for data types are available for relating Boolean and vector expressions within sequence and property definitions.

Keyword Table

Keywords specific to assertions sequences, properties and templates.			
`property`	`endproperty`	`sequence`	`endsequence`
`and`	`or`	`intersect`	`within`
`throughout`	`first_match`	`ended`	`matched`
`assert`	`assume`	`cover`	

Operator table

Name	Operator	Description
Consecutive repetition	`s1 [* N:M]`	Repetition of s1 N, or between N to M times.
Goto repetition	`s1 [-> N:M]`	Repetition of s1 (N to M times) in nonconsecutive cycles, ending on s1 .
Nonconsecutive repetition	`s1 [= N:M]`	Repetition of s1 N to M times in nonconsecutive cycles, maybe ending on s1.
Temporal delay	`# # N` `## [N:M]`	Concatenation of two sequence elements.
Not	**`not`** `p1`	Invert result of evaluation of the property.
And	`s1` **`and`** `s1` `p1` **`and`** `p2`	Both sequence/properties occur at some time.
Intersection	`s1` **`intersect`** `s2`	Both sequences occur at the same time.
Or	`s1` **`or`** `s2` `p1` **`or`** `p2`	Either sequence/property occurs.
Condition	**`if`** `(expr) p1` **`if`** `(expr) p1` **`else`** `p2`	Based on evaluation of expr, evaluate property p1 if true, or p2 if false.
Boolean until	`b` **`throughout`** `s1`	B must be true until sequence s1 completes (results in a match).
Within	`s1` **`within`** `s2`	s1 and s2 must occur. Lengths of s1 and s2 must follow s1 <= s2.
Ended	`s1.`**`ended`**	Sequence s1 matched (ended) at this time.
Matched (from different clock domain)	`s1.`**`matched`**	Sequence s1 (on another clock) ended at this time.
First match	**`first_match`** `(s1)`	First occurrence of sequence, rest ignored.

Name	Operator	Description
Overlapping implication	s1 |-> p1	If s1 occurs, p1 (starting this cycle) must occur, else true.
Non-overlapping implication	s1 |=> p1	If s1 occurs, p1 (starting next cycle) must occur, else true.

System Function table

$rose	$fell	$stable
$countones	$onehot	$onehot0
$isunknown	$past	$sampled

Operator precedence table

Operator (sequence or property)	Associativity
Repetition (consecutive, nonconsecutive, goto)	
Delay (##)	Left
Boolean until (Throughout)	Right
Within	Left
Intersection	Left
Not (property)	
And (sequence and property)	Left
Or (sequence and property)	Left
if..else	
Implication (overlapping, nonoverlapping)	Right

C.3 Sequence and property operations

The new sequence operators defined for SystemVerilog allow us to compose expressions into temporal sequences. These sequences are the building blocks of properties and concurrent assertions. The first four allow us to construct sequences, while the remaining operators allow us to perform operations on a sequence as well as compose a complex sequence from simpler sequences.

C.3.1 Temporal delay

The temporal delay operator "##" constructs larger sequences by combining smaller sequences and expressions.

Example C-1 Construction of sequences with temporal delay

```
sequence_expr ::= [ cycle_delay_range ]
      sequence_expr { sequence_expr cycle_delay_range }

cycle_delay_range ::=
    ##   constant_expression
   ## '[' constant_expression ':' constant_expression ']'
   ## '[' constant_expression ':' '$' ']' -- Infinite range.
```

A temporal delay may begin a sequence. The range may be a single constant amount or a range of time. All times may be used to match the following sequence. The range is interpreted as follows:

- ##0 a - same as (a)
- ##1 a - same as (1 ##1 a)
- ##[0:1] a - same as a or (1 ##1 a)

When the symbol **$** is used, the range is infinite.

The right side sequence of a concatenation is expected on the following cycle unless the delay has the value 0. Each subsequent element of a concatenation further advances time one cycle. Examples:

- 1 ##1 a - means a must be true in cycle 1.
- 1 ##1 a ##1 b - means a must be true in cycle 1, b must be true in cycle 2.
- a ##[0:2] b same as a & b or (a ##1 b) or (a ##1 1 ##1 b)
- a ##[2:3] b same as (a ##1 1 ##1 b) or (a ##1 1 ##1 1 ##1 b)

Note:

When a range is used, it is possible for the sequence to match with each value within the range. Thus, we must take into account that there may be multiple possible matches when we write sequences with a range. See C.3.12, "First match" for more information.

C.3.2 Consecutive repetition

Consecutive repetition of a sequence applies to a single element or a sequence expression.

Example C-2 Consecutive repetition of a sequence

```
sequence_expr ::=
     sequence_expr       '[' '*' const_range_expression ']'
   | expression_or_dist  '[' '*' const_range_expression ']'

const_range_expression ::=
     constant_range_expression
   | constant_range_expression ':' constant_range_expression
   | constant_range_expression ':' '$'
```

When using a repetition, the element or sequence must occur starting in the next cycle for each repetition expected. Examples are:

- a[* 0] ##1 b same as (b)
- a[* 2] ##1 b same as (a ##1 a ##1 b)
- a [* 1:2] ##1 bsame as (a ##1 b) or (a ##1 a ##1 b)
- (a ##1 b) [* 2] same as (a ##1 b ##1 a ##1 b)

The first example comes from the fact that a repetition of zero is equivalent to the following:

```
a[*0] ##N b === ## (N-1)  b
```

Note:

When a range is used, it is possible for the sequence to match with each value within the range. Thus, we must take into account that there may be multiple possible matches when we write sequences with a range. See C.3.12, “First match” for more information.

C.3.3 Goto repetition

goto repetition allows for space (absense of the term) between the repetition of the terms. However, the repetition must end on the Boolean expression being found. This facilitates searching for

a particular expression and then continuing the sequence at the point it is found.

Example C-3 Goto repetition of a sequence

```
sequence_expr : :=
    expression_or_dist '[' '->' const_range_expression ']'

const_range_expression ::=constant_expression
  | constant_expression ':' constant_expression
  | constant_expression ':' '$'
```

Goto repetition is defined in terms of the other operators as:

```
s [-> min:max] ::= (!s[*0:$] ##1 s) [* min:max]
```

Examples are:

- a[->0] ##1 b same as (b)
- a[->1] ##1 b same as (!a [*0:$] ##1 a ##1 b)
- a[->2] ##1 b same as (!a [*0:$] ##1 a ##1 !a [*0:$] ##1 a ##1 b)

C.3.4 Nonconsecutive repetition

Nonconsecutive repetition, like goto repetition allows for space between the repetition of the expression. At the end of the repetition, however, there can be additional space after the repeated expression.

Example C-4 Nonconsecutive repetition of a sequence

```
sequence_expr ::=
    expression_or_dist '[' '=' const_range_expression ']'

const_range_expression ::= constant_range_expression
  | constant_range_expression ':' constant_range_expression
  | constant_range_expression ':' '$'
```

Nonconsecutive repetition is defined in terms of the other operators as:

```
s [= min:max] ::= ((!s[*0:$] ##1 s )
                   [* min:max ]) ##1 !s[*0:$]
```

Examples are:

- a [=0] ##1 b same as (b)

- a[=1] ##1 b same as (!a [*0:$] ##1 a ##1 !a [*0:$] ##1b)
- a [=2] ##1 b same as (!a [*0:$] ##1 a ##1 !a [*0:$] ##1 a ##1 !a [*0:$] ##1 b)

C.3.5 Sequence and Property AND

Both properties and sequences can be operands of the and operator. The operation on two sequences produces a match once both sequences produce a match (the end point may not match). A match occurs until the endpoint of the longer sequence (provided the shorter sequence produces one match).

Example C-5 Sequence and (non-matching length)

```
sequence_expr ::= sequence_expr and sequence_expr
```

This operation, for sequences, is defined in terms of the next operator (intersect) as:

```
s and t ::= ( (s ##1 1 [*0:$ ]) intersect t)
            or (s intersect (t ##1 1 [*0:$])).
```

Examples of sequence and'ing are (() mean no match):

- (a ##1 b) **and** () same as ()
- (a ##1 b) **and** (c ##l d) same as (a&c ##l b&d)
- (a ##[1:2] b) **and** (c ##3 d)same as (a & c ##1 b ##1 1 ##1d)
  or (a & c##1 1##1 b##1 d)

Note:

It is possible for this sequence operation to match with each value of a range or repeat in the operands. Thus, we must take into account that there may be multiple possible matches when we use this operator. See C.3.12, "First match" for more information.

Properties can be combined using this operator to produce a match if both properties evaluate to true. Only one match will be produced when the operands are properties. A sequence is implicitly transformed into a property through the application of first_match to the sequence. See C.3.13, "Property Implication" for addition information.

C.3.6 Sequence intersection

An intersection of two sequences is like an and of two sequences (both sequences produce a match). This operator also requires the length of the sequences to match. That is, the match point of both sequences must be the same time. With multiple matches of each sequence, a match occurs each time both sequences produce a match.

Example C-6 Sequence and (matching length)

```
sequence_expr ::= sequence_expr intersect sequence_expr
```

Examples of a sequence intersection are (() means no match):

- (1) **intersect** () same as ()
- ##1 a **intersect** ##2 b same as ()
- ##2 a **intersect** ##2 b match if (##2 (a & b))
- ##[1:2] a **intersect** ##[2:3] b match if (1 ##1 a&b)
  or (1 ##1 a&b ##1 b)
- ##[1:3] a **intersect** ## [1:3] bmatch if (##1 a&b)
  or (##2 a&b)
  or (##3 a&b)

Note:

It is possible for this operation to match with each value of a range or repeat in the operands. Thus, we must take into account that there may be multiple possible matches when we use this operator. See C.3.12, “First match” for more information.

C.3.7 Sequence and Property OR

For sequences, a match on either sequence results in a match for the operation.

Example C-7 Sequence or

```
sequence_expr ::= sequence_expr or sequence_expr
```

Examples of sequence or are:

- () **or** () same as ()
- (## 2 a **or** ## [1:2] b) match if
  (b) or (##1 b) or (## 2 a) or (##2 b)

Note:

It is possible for this operation to match with each value of a range or repeat in the operands. Thus, we must take into account that there may be multiple possible matches when we use this operator. See C.3.12, “First match” for more information.

Properties can be combined using this operator to produce a match if one property evaluate to true. Only one match will be produced when the operands are properties. A sequence is implicitly transformed into a property through the application of first_match to the sequence. See C.3.13, “Property Implication” for addition information.

C.3.8 Boolean until (throughout)

This operator matches a Boolean value throughout a sequence. The operator produces a match if the sequence matches and the Boolean expression is true until the end of the sequence.

Example C-8 (Boolean) throughout sequence

```
sequence_expr ::= expression_or_dist throughout sequence_expr
```

The throughout operator is defined in terms of the intersect sequence operators. Its definition is:

```
(b throughout s) ::= ( b [*0:$] intersect s)
```

Examples include:

- 0 **throughout** (1) same as()
- 1 **throughout** ##1 a same as##1 a
- a **throughout** ##2 b same as(a ##1 a & b)
- a **throughout** (b ##[1:3] c) same as (a&b ##1 a ##1 a &c)
  or (a&b ##1 a ##1 a ##1 a &c)
  or (a&b ##1 a ##1 a ##1 a ##1 a &c)

Note:

It is possible for this operation to match with each value of a range or repeat in the operands. Thus, we must take into account that there may be multiple possible matches when we use this operator. See C.3.12, “First match” for more information.

C.3.9 Within sequence

The **within** operator determines if one sequence matches within the length of another sequence.

Example C-9 Within sequence

```
sequence_expr ::= sequence_expr within sequence_expr
```

The within operator is defined in terms of the intersect sequence operators. Its definition is:

```
s1 within s2 ::= ((1 [*0:$] ##1 s1 ##1
        1 [*0:$]) intersect s2)
```

This means that s1 may start the same time as s2 or later and may end earlier or at the same time as s2. Examples are (() means no match):

- () **within** (1) same as ()
- (1)**within** () same as ()
- (a) **within** ## [1:2] b same as (a&b) or (b ##1 a) or (a##1 b) or (##1 a&b)

Note:

It is possible for this operation to match with each value of a range or repeat in the operands. Thus, we must take into account that there may be multiple possible matches when we use this operator. See C.3.12, “First match” for more information.

C.3.10 Ended

The method ended returns true at the end of the sequence. This is in contrast to matching a sequence from the beginning timepoint, which is obtained when we use only the sequence name.

Example C-10 Ended sequence

```
expression ::= sequence_identifier. ended

sequence_identifier ::= (identifier of type sequence)
```

Examples include:

```
sequence a1;
   @(posedge clk) (c ##1 b ##1 d);
```

endsequence

- `(a ##1 a1.ended)` same as `(c ##1 b & a ##1 d)`
- `(a ##2 a1.ended)` same as `(c &a ## b ##1 d)`

Note the position of 'a' relative to the end of sequence 'a1' (term 'd')- Compare this with the following sequence where 'a' occurs before 'a1'

- `(a ##1 a1)` same as `(a ##1 c ##1 b ##1 d)`

C.3.11 Matched

The method matched operates similarly to the ended method; however, this method is used when the sequence, of the method call, uses a different clock than the sequence being called.

Example C-11 Matched sequence

```
expression ::= sequence_identifier '.' matched

sequence_identifier ::= (identifier of type sequence)
```

C.3.12 First match

The first match operator returns only the first match from a sequence. The remaining are suppressed.

Example C-12 First match

```
sequence_expr ::= first_match sequence_expr
```

Examples of first_match are:

- **first_match** (1 [*1:5]) same as (1)
- **first_match** (##[0:4] 1)same as (1)
- **first_match** (##[0:1] a)same as (a) or (!a ##1 a)
- **first_match (**s1 **intersect** s2) same as (s1 **intersect first_match**(s2)
- **first_match (**b **throughout** s1) same as (b **throughout first_match**(s1))
- **first_match(**s1 **within** s2) same as (s1 **within first_match** (s2))

- **first_match**(expression) same as (expression [->1])

Note:

Use of a range on the delay operators or a range on the repetition operators can cause multiple matches. Use of first_match can be helpful to suppress the subsequent matches. These additional matches may cause a false firing that is solved with this operator.

C.3.13 Property Implication

As a convenience, there are two forms of implication, overlapping and non-overlapping. The overlap occurs between the final cycle of the left-hand side (the antecedent) and the starting cycle of the right-hand side (the consequent) operands. For the overlapping form, the consequent starts on the current cycle (that the antecedent matched). The non-overlapping form has the consequent start the subsequent cycle. Implication is similar in concept to an if() statement.

Implication uses the antecedent as a test condition to determine whether the consequent should be evaluated or the operation should (vacuously) return a true result. The consequent is a property, allowing for negations, implication, and other property operations.

Example C-13 Overlapping implication

```
property_expr ::= sequence_expr '|->' property_expr
```

Example C-14 Non-overlapping (suffix) Implication

```
property_expr ::= sequence_expr '|=>' property_expr
```

Examples include:

- a |->b same as a ? b : 1'b1
- (a##1 b) |-> (c) same as (a##1 b)?c : 1'b1
- (a ##1 b) => (c ##1 d) same as (a ##1 b) |-> ##1 c ##1 d)

When the consequent property[1] is simply a sequence, the first match of the sequence is sufficient for the property to return true. This is equivalent to writing:

- antecedent |-> first_match(property_is_a_sequence)

Thus, a property will produce a single match or no match, regardless of the operators used.

C.3.14 Conditional property selection

Properties can be conditionall selected using the if...else operator.

Example C-15 Conditional selection using if...else

```
property_expr ::= if (expression) property_expr1
    |     if (expression) property_expr1 else property_expr2
```

The *conditional property selection* operator operates similarly to the procedural `if()` statement. The property evaluates to true if the (Boolean) expression is true and `property_expr1` evaluates to true, or expression is false and `property_expr2` evaluates to true.

Examples include:

- A |=> **first_match**(##[1:10] B | C)
  |-> **if**(B) D##[1:10] F
  else G [*2]

The first line of the property identifies that sequence **B** after **A** or **C** after **A**.[2] The second line uses the conditional selection operator to choose the next property based on which sequence was detected. If **B** was found the property `D ##[1:10] F` is expected to match. If **B** was not found (implying **C** was found) the property `G [*2]` is expected to match.

1. Actually any property that is only a sequence (does not contain the property operations) is satisfied on the first match, equivalent to implicitly using the first_match() operator.

2. The first_match operator is used to prevent a subsequent shorter match from incorrectly completing the property. We could have used the sequence `(b|c) [->1]` if a time limit was not required.

C.4 Property declarations

SystemVerilog allows for declarations of both sequences and properties. A property differs from a sequence in that it contains a clock specification for the entire property, an asynchronous term to disable property evaluations and additional operators that can be used. Properties allow the operators negation, if..else, and implication operations.

Example C-16 Property declaration

```
property_declaration ::=
      property property_identifier [ property_formal_list ] ';'
           { property_decl_item }
           property_spec ';'
      endproperty [ ':' property_identifier ]

property_formal_list ::= '(' formal_list_item { ',' formal_list_item } ')'
formal_list_item    ::=  formal_identifier [ '=' actual_arg_expr ]
actual_arg_expr   ::= expression | identifier | event_control | '$'

property_decl_item ::=  sequence_declaration
       |  list_of_variable_identifiers_or_assignments

property_spec ::=  [ event_control ]
   [ disable iff '(' expression ')' ]  property_expr

property_expr ::=
     sequence_expr
   | event_control  property_expr
   | '(' property_expr ')'
   | not  property_expr
   | property_expr  or   property_expr
   | property_expr  and  property_expr
   | sequence_expr  |-> property_expr
   | sequence_expr  |=> property_expr
   | if '(' expression ')' property_expr
       [ else property_expr ]
   | property_instance

event_control   ::= '@' event_identifier
   | '@' '(' event_expression ')'
```

Properties can be complete definitions useful with other properties, or they can be used for verification as an assertion, assumption or as a coverage point. Properties can also contain parameters to be specified when they are used in these other contexts. Examples include:

```
property req_t1_start;
        @(posedge clk) req && req_tag == t1;
endproperty

property illegal_op;
        @(posedge clk) not req && cmd == 4;
```

```
*** not @(posedge clk)  req && cmd == 4;
endproperty
property starts_at(start, grant, reset_n);
   disable iff (~reset_n)  // asynch reset of property.
   @(posedge clk) (grant[->1] ##1 grant & start);
endproperty
```

Properties may reference other properties in their definition. They may even reference themselves, recursively. Properties may also be written utilizing multiple clocks to define a sequence that transitions from one clock to another as it matches the elements of the sequence. See the SystemVerilog LRM for additional details.

Sequences are also declared. They use syntax similar to properties.

Example C-17 Sequence declaration

```
sequence_declaration ::=
  sequence sequence_identifier [sequence_formal_list ] ';'
    { assertion_variable_declaration }
    sequence_expr ';'
  endsequence [ ':' sequence_identifier ]

sequence_formal_list ::=
  '(' formal_list_item { ',' formal_list_item } ')'

assertion_variable_declaration ::=
  data_type_list_of_variable_declarations ';'
```

They can be defined within properties and as separate elements. They also can be declared with parameters that are specified when used in other contexts. These elements, coupled with the following directives, allow one to define and utilize properties to follow an assertion-based design methodology. Examples include:

```
sequence op_retry;
   (req ##1 retry);
endsequence
sequence cache_fill(req, done, fill);
    (req ##1 done [-> 1] ##1 fill [->1]);
endsequence
```

C.4.1 Sequence composition

Sequence creation can use all the operators defined above, except the property operators implication, conditional selection (if...else) and negation. Example C-18 shows the BNF.

Example C-18 Sequence specification

```
cycle_delay_range ::=
       '##' constant_expression
    |  '##' '[' const_range_expression ']'

const_range_expression ::=
      constant_expression : constant_expression
    |  constant_expression : '$'

sequence_expr ::= [ cycle_delay_range ]
         sequence_expr { cycle_delay_range sequence_expr }
   | expression_or_dist [boolean_repeat]
   | expression_or_dist { ',' sequence_match_item } [boolean_repeat]
   | sequence instance [ consecutive_repeat ]
   | sequence_expr { ',' sequence_match_item } [consecutive_repeat]
   | event_control sequence_expr
   | '(' sequence_expr ')'
   | sequence_expr and      sequence_expr
   | sequence_expr or       sequence_expr
   | sequence_expr intersect  sequence_expr
   | sequence_expr within   sequence_expr
   | expression    throughout sequence_expr

sequence_match_item ::= variable_assignment | subroutine_call

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

boolean_repeat ::= consecutive_repeat
   |  nonconsecutive_repeat
   |  goto_repeat

consecutive_repeat    ::=
           '[' '*' const_range_expression ']'

nonconsecutive_repeat ::=
           '[' '=' const_range_expression ']'

goto_repeat         ::=
           '[' '->' const_range_expression ']'
```

C.5 Assert, Assume and Cover statements.

Property directives define how to use properties (and sequences) for specific works. These statements utilize all the elements above to define how they are to be used for a given design.

Example C-19 Property directives

```
immediate_assert_statement ::=
    assert ( expression ) action_block

concurrent_assert_statement ::=
  assert property `(` property_spec `)` action_block

concurrent_assume_statement ::=
  assume property `(` property_spec `)` `;`

concurrent_cover_statement ::=
  cover property `(` property_spec `)` statement_or_null

action_block ::=
    statement   [ else statement_or_null ]
  | [statement_or_null] else statement_or_null

statement_or_null ::=
    statement   |   `;`
```

As an assertion, properties are evaluated, and when they do not match the desirable sequence, they fail and produce an error message by default (or they execute the else statement set). When they match a sequence, the first (pass) statement set is executed (like an **if()** statement).

As an assumption, the properties are expected to hold true during simulation. For formal property tools, the assumptions are also expected to hold true during a proof. The formal tool is not required to verify all assumptions are true though, allowing one to specify assumptions restricting the logic to a subset of all legal input values. This restriction may be to simplify proofs during development (see Section 2.5.4, “Gradual exhaustive formal verification” on page 56).

As a cover directive, properties are evaluated, and when they succeed, they execute the first (pass) statement set. If they fail to match a sequence they execute the else statement set, if any. Examples include:

- **assert property** (illegal_op) **else** $error;
- **assert property** (req => done[->1] ##1 fill[->1])
  else
  $warning(“Fill did not occur for completed mem read. Why?”);

- **assert property** (illegal_op)
  else $error("Illegal operation occurred on bus B.");
- **assume property** (legal_bus_cmds);
- **cover property** (req_t1_start)
  begin
  $display("Starting Request t1.");
  end

C.6 Dynamic data within sequences

In addition to matching sequences of events, SystemVerilog includes the ability to call a routine or sample data within the sequence for later comparison or matching. The ability to call a routine (tasks, void functions, methods and system tasks) allows one to record or alter information. System tasks are very useful as a debug aid to track property and sequence progression. For example, the $display system task is used to display information relative to the state of the sequence.

```
sequence track_it;
   (req[->1], $display("Found the request at cycle %0d\n", 'cycle))
   ##1 (sent[->1], $display("Sent occurred at cycle %0d\n", 'cycle))
  ##3 end;
endsequence
assert property (start |=> track_it);
```

Once start occurred, the following output could be produced:

```
Found the request at cycle 1101
Sent occurred at cycle 1109
```

Error messages could also be produced when used on a condition the inverse of what was expected. These error messages would directly reflect the values of the signals used in the property (the sampled values.) This also provides the ability to display local variable state as computed during the execution of the sequence.

Both properties and sequences allow us to declare variables that will be assigned data at some point within a sequence. Note, these variables are separate for each property or sequence start. That is, sequences or properties that can start on each subsequent cycle have independent variables. If shared variables are desired, they should be declared like other RTL variables with code provided to

set their values. Here we show the property declaration for variables.

Example c-20 Dynamic data

```
property_declaration ::=
   property property_identifier [ property_formal_list ] ';'
    { assertion_variable_declaration }
    property_spec ';'
    endproperty [ ':' property identifier ]

assertion_variable_declaration ::=
    data_type list_of_variable_identifiers ';'
```

Examples include:

```
property addr_sent;
   reg [31:0] addr;
   // Find start and save address, then find the next
   // request and compare the address with the saved one
    (start, addr = new_addr |=> req[->1]
           ##0 req_addr == addr);
endproperty
```

C.7 System Functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is one-hot. The following system functions are included to facilitate such common assertion functionality:

- $onehot (<expression>) returns true if one and only one bit of expression is high
- $onehot0 (<expression>) returns true if at most one bit of expression is high
- $isunknown (<expression>) returns true if any bit of the expression is 'X' or 'Z'. This is equivalent to:
  "^ <expression> === 1'bx"
- $stable (<expression>, <clocking event>) returns true if the previous value of the expression is the same as the current value of the expression.
- $rose (<expression>, <clocking event>) returns true if the expression was previously zero and the current value is nonzero. If the expression length is more than one bit, then only bit 0 is used to determine a positive edge.

- $fell (<expression>, <clocking event>) returns true if the expression was previously nonzero and the current value is zero. If the expression length is more than one bit, then only bit 0 is used to determine a negative edge.

All of the above system functions have a return type of bit. A return value of 1'b1 indicates true, and a return value of 1'b0 indicates false. These following system functions return a value equivalent to the length of the first (or only) expression.

- $past (<expression>, <num cycles>, <clock gate>, <clocking event>) returns the value of the expression from *num cycles* ago. If the value did not exist, 'bx is returned. The clock gate expression, if specified, causes the evaluation of the clock to be dependent on its asserted value. The clocking event, if specified, explicitly defines the sampling clock. For example the `$past` call is equivalent to the following code:

Example C-21 $past equivalent code

```
assign old_data =
         $past(data, 2, ld_done, posedge clk);

always @ (posedge clk)
  if (ld_done)
    {old_data, prev_data} <= { prev_data, data};
```

- $sampled (<expression>, <clocking event>) returns the sampled value of the expression. If the value did not exist, `'bx` is returned. The clocking event is implicitly inferred unless explicitly specified as the second expression.
- $countones (<expression>) returns the number of bits asserted in the expression.

C.7.1 New operators

SystemVerilog introduced new operators that can make specifying expression clearer to understand. These operators are the wilcard comparison operator =?= and the inside operator.

The wildcard comparison operator (=?=) allows one to compare a signal or another expression against a constant with don't care bits expressed as X or Z. The evaluation semantics match the casex procedural statement. For example

```
wire match = {valid, cmd[4:0]} =?= 6'b1_?1111;
```

The signal match will be asserted when `valid` is asserted and `cmd[4:0]` is either `5'h1f` or `5'hf`. This can simplify expressions into comparisons instead of equations of specific bits.

The inside operator is similar to the wildcard comparison operator. It compares a signal or expression against a list of constants or expressions. The comparison matches the casex procedural statement comparisons with one exception. Any bits of the left hand side operand that are X or Z are not considered to match the bits on the right hand side operand. This prevents unknown bits from causing a match. For example:

```
wire inset =  {valid, attr[2:0], attr1[1:0]}
  inside {6'h0,          // invalid
          6'b1_110_00,    // strobe 0
          6'b1_111_x1,    // strobe 1y
          6'b1_011_z1};   // strobe 3y
```

This expression allows don't care bits to be expressed in the list of expressions, but will not allow any of the signals valid, attr or attr1 to be 'bX and match a value from the right side. This is expressed as:

```
    1'bx inside 1'b0, 1'b1) === 1'b0
```

C.8 SystemTasks

SystemVerilog has defined several severity system tasks for use in reporting messages with a common message. These system tasks are defined as follows:

```
$fatal (finish_num [, message
       {, message argument } ] );
```

This system task reports the error message provided and terminates the simulation with the `finish_num` error code. This system task is best used to report fatal errors related to testbench/ OS system failures (for example, can't open, read, or write to files) The message and argument format is the same as the **$display()** system task.

$error(message {, message_argument});

This system task reports the error message as a run-time error in a tool-specific manner. However, it provides the following information:

- severity of the message

- file and line number of the system task call
- hierarchical name of the scope or assertion or property
- simulation time of the failure

$warning(message {, message_argument});

This system task reports the warning message as a run-time warning in format similar to $error and with similar information.

$info(message {, message_argument});

This system task reports the informational message in a format similar to $error and with similar information.

$asserton(levels, [list_of_modules_or_assertions])

This system task will reenable assertion and coverage statements. This allows sequences and properties to match elements. If a `level` of 0 is given, all statements in the design are affected. If a list of *modules* is given, then that module and modules instantiated to a depth of the `level` parameter are affected. If specifically named assertion statements are listed, then they are affected.

$assertkill(levels, [list_of_modules_or_assertions])

This system task stops the execution of all assertion and cover statements. These statements will not begin matching until reenabled with **$asserton()**. Use the arguments in the same way as $asserton uses them.

$assertoff(levels, [list_of_modules_or_assertions])

This system task prevents matching of assertion and cover statements. Sequences and properties in the progress of matching sequences will continue. Assertion and cover statements will not begin matching again until reenabled with **$assserton()**. Use the arguments in the same way as $asserton uses them.

C.9 BNF

The SystemVerilog BNF represented here is the property specification subset. This subset resides within a module context. It also utilizes the expression BNF subset as part of its expressions.

C.9.1 Use of Assertions BNF:

```
concurrent_assertion_item::=
        | concurrent_assert_statement
        | concurrent_assume_statement
        | concurrent_cover_statement
procedural_assertion_item::=
        immediate_assert_statement
        | concurrent_assert_statement
        | concurrent_cover_statement
```

C.9.2 Assertion statements

```
immediate_assert_statement ::=
    assert ( expression ) action_block
concurrent_assert_statement ::=
    assert property ‘(‘ property_spec ‘)’ action_block
concurrent_assume_statement ::=
    assume property ‘(‘ property_spec ‘)’
concurrent_cover_statement ::=
    cover property ‘(‘ property_spec ‘)’ statement_or_null
action_block      ::=
      statement          [ else statement_or_null ]
   | [statement_or_null] else statement_or_null
statement_or_null ::=
      statement | ‘;’
```

C.9.3 Property and sequence declarations

```
property_declaration ::=
  property property_identifier [ property_formal_list ] ';'
    { assertion_variable_declaration }
    property_spec ';'
  endproperty [ ':' property_identifier ]

property_formal_list ::=
          '(' formal_list_item {',' formal_list_item } ')'
formal_list_item ::= formal_identifier [ '=' actual_arg_expr]
actual_arg_expr ::= expression | identifier | event_control

property_decl_item ::=  sequence_declaration
        | list_of_variable_identifiers_or_assignments

sequence_declaration  ::=
  sequence sequence_identifier [sequence_formal_list ]';'
    {assertion_variable_declaration }
    sequence_expr';'
  endsequence [':' sequence_identifier ]
sequence_formal_list ::=
       '(' formal_list_item {',' formal_list_item } ')'
assertion_variable_declaration ::=
          data_type   list_of variable_identifiers.
```

C.9.4 Property construction

```
property_spec ::= [ event_control ]
    [ disable iff '(' expression ')' ] multi_clock_property_expr

property_expr ::=
    sequence_expr
  | event_control property_expr
  | '(' property_expr ')'
  | not property_expr
  | property_expr or property_expr
  | property_expr and property_expr
  | sequence_expr |-> property_expr
  | sequence_expr |=> property_expr
  | if '(' expression ')' property_expr
      [ else property_expr ]
  | property_instance

event_control ::= '@' event_identifier
          | '@' '(' event_expression ')'
```

C.9.5 Sequence construction

```
cycle_delay_range ::=
      '##' constant_expression
   | '##' '[' const_range_expression ']'

const_range_expression ::= constant_expression : '$'
   |  constant_expression : constant_expression

sequence_expr ::= [ cycle_delay_range ]
            sequence_expr  { cycle_delay_range sequence_expr }
   | expression_or_dist [boolean_repeat]
   | expression_or_dist { ',' sequence_match_item } [boolean_repeat]
   | sequence_instance [ consecutive_repeat ]
   | sequence_expr { ',' sequence_match_item }  [consecutive_repeat]
   | event_control  sequence_expr
   | '(' sequence_expr ')'
   | sequence_expr and        sequence_expr
   | sequence_expr or         sequence_expr
   | sequence_expr intersect  sequence_expr
   | sequence_expr within     sequence_expr
   | expression_or_dist   throughout sequence_expr

sequence_match_item ::= variable_assignment | subroutine_call
expression_or_dist ::= expression
            | expression  dist '{' dist_list '}'

boolean_repeat ::= consecutive_repeat
  | nonconsecutive_repeat
  | goto_repeat'

consecutive_repeat    ::=
    '[' '*'  const_range_expression ']'
nonconsecutive_repeat ::=
    '[' '=' const_range_expression  ']'
goto_repeat           ::=
    '[' '->' const_range_expression ']'
```

BIBLIOGRAPHY

[Abarbanel et al. 2000] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, Y. Wolfsthal, “FoCs--Automatic Generation of Simulation Checkers from Formal Specifications”, *Proc. Computer Aided Verification, 12th International Conference, CAV 2000,* pp. 414-427, July 15-19, 2000.

[Accellera OVL 2003] Accellera proposed standard Open Verification Library Users Reference Manual, 2003.

[Accellera PSL-1.1 2004] Accellera proposed standard Property Specification Language (PSL) 1.1, March 2004.

[Accellera SystemVerilog-3.1 a 2004] Accellera proposed standard SystemVerilog 3.1a, March 2004.

[Adir et al., 2002a] A. Adir, R. Emek, E. Marcus. "Adaptive Test Program Generation: Planning for the Unplanned." *Proc. IEEE Int'l High Level Design Validation and Test Workshop,* 2002.

[Adir et al., 2002b] ?A. Adir, R. Emek, E. Marcus. "Adaptive Test Generation." *Proc. Microprocessor Test and Verification Conference,* 2002.

[Alexander 1977] C. Alexander, *A Pattern Language,* New York: Oxford University Press, 1977.

[Alexander 1979] C. Alexander, *The Timeless Way of Building,* New York: Oxford University Press, 1979.

[AMBA-2.0 1999] AMBA™ Specification, Revision 2.0, ARM Limited, 1999.

[Appleton 2000] B. Appleton, “Patterns and Software: Essential Concepts and Terminology”, Retrived March 31, 2003, from the World Wide Web: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

[Beizer 1990] B. Beizer, *Software Testing Techniques,* Van Nostrand Rheinhold, New York, second edition, 1990.

[Benjamin et al., 1999] Benjamin, M., D. Geist, A. Hartman, G. Mas, and Y. Wolfsthal. "A Study in Coverage-Directed Test Generation." *Proc. Design Automation Conference,* 1999.

[Bentley 2001] B. Bentley, “Validating the Intel Pentium 4 Microprocessor”, *Proc. Design Automation Conference,* pp. 244-248, 2001.

[Bening and Foster 2001] L. Bening, H. Foster, *Principles of Verifiable RTL Design,* Kluwer Academic Publishers, 2001.

[Ben-Ari et al. 1983] M. Ben-Ari, Z. Manna, and A, Pnueli, “The temporal logic of branching time", *Acta Informatica 20,* 1983.

[Bergeron 2003] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models, Second Edition,* Kluwer Academic Publishers, 2003.

[Betts et al. 2002] J. A. Betts, F. Delguste, S. Brefort, C. Clause, A. Salas, T.Langswert, "The Role of Functional Coverage in Verifying the C166S IP", *Proc. Intn'l Workshop on IP-based System-On-Chip Design,* 2002.

[Borland 2002] "Borland developer network", Article ID: 28432 Publish Date: February 21, 2002 Last Modified: March 06,2002

[Clarke and Emerson 1981] [E. Clarke, E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic". *Logic of Programs: Workshop,* LNCS 407, Springer 1981.

[Clarke et al. 2000] [E. Clarke, O. Grumberg, D. Peled, *Model Checking,* The MIT Press, 2000.

[Coonan 2000] J. T. Coonan, "ASIC Design & Verification Top-10 List", Retrived March 31, 2003, from the World Wide Web: http://www.mindspring.com/~tcoonan/asicdv.html.

[Coplien 2000] J. Coplien, *Software Patterns,* SIGS Books and Bultimedia, New York, 2000.

[Devadas et al. 1996] S. Devadas, A. Ghosh, K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," *Proc. Intn'l Conf. on Computer-Aided Design,* pp. 418-425, 1996.

[Drako and Cohen 1998] D. Drako and P. Cohen, "HDL Verification Coverage", *Integrated System Design,* June, 1998.

[Dwyer et al. 1998] *M. Dwyer, G. Avrunin, J. Corbett,* 2nd Workshop on Formal Methods in Software Practice, March, 1998, Retrived January 30, 2003, from the World Wide Web: http://www.cis.ksu.edu/~dwyer/papers/spatterns.ps.

[e Language Reference Manual] *e Language Reference Manual* is available at http://www.ieee1647.org/

[Fallah et al. 1998] F. Fallah, S. Devadas, K. Keutzer, "OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification," *Proc. Design Automation Conference,* pp. 152-157, 1998.

[Fitzpatrick et al. 2002] T. Fitzpatrick, H. Foster, E. Marschner, P. Narain, "Introduction to Accellera's assertion efforts", EEDesign, Retrieved June 2, 2002, from the World Wide Web: http://www.eedesign.com/story/OEG20020602S0001

[Floyd 1967] Robert Floyd, "Assigning meanings to programs" *Proceedings, Symposium on Applied Mathematics,* Volume 19, American Mathematical Society, Providence, RI (1967), pp. 19– 32

[Foster and Coelho 2001] H. Foster, C. Coelho, "Assertions Targeting A Diverse Set of Verification Tools", *Proc. Intn'l HDL Conference,* March, 2001.

[Foster et al. 2002] H. Foster, P. Flake, and T. Fitzpatrick, “Adding Design Assertion Extensions to SystemVerilog”, *Proc. Intn’l HDL Conference,* March, 2002

[Foster et al. 2004] H. Foster, H. Wong-Toi, C. N. Ip, D. Perry, “Formal Verification of Block-Level Requirements”, *DesignCon,* 2004

[Gamma et al. 1995] E. Gamma, R. Helm, R Johnson, J Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software,* 1995.

[Geist et al.., 1996] Geist, D., M. Farkash, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. "Coveraged Directed Generation Using Symbolic Techniques." *Proc. Int'l Conference on Formal Methods in Computer-Aided Design,* 1996.

[Grinwald et al. 1998] R. Grinwald, E. Harel, M. Orgad, S. Ur, A. Ziv, “User Defined Coverage - A Tool Supported Methodology for Design Verification”, *Proc. Design Automation Conference,* 1998.

[Grumberg and Long 1994] O. Grumberg and D. Long. Model checking and modular verification. ACM Transaction on Programming Languages and Systems 16: 843-872, 1994.

[Hoare 1969] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM* Vol 12, No. 10, October 1969.

[Horgan et al. 1994] J. Horgan, S. London, M. Lyu, “Achieving Software Quality with Testing Coverage Measures,” *Computer,* 27(9), pp. 60-69, September 1994.

[IEEE 1076-1993] IEEE Standard 1076-1993 *VHDL Language Reference Manual,* IEEE, Inc., New York, NY, USA, June 6, 1994.

[IEEE 1364-2001] IEEE Standard 1364-2001 *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language,* IEEE, Inc., New York, NY, USA, March 2001.

[Kernighan 1978] B Kernighan, *Elements of Programming Style,* McGraw-Hill, 1998

[Keating and Bricaud 2002] M. Keating and P. Bricaud, *Reuse Methodology Manual,* Kluwer Academic Publishers, 2002.

[Kantrowitz and Noack 1996] M. Kantrowitz, L. Noack, “I’m done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha microprocessor”, *Proc. Design Automation Conference,* pp. 325-330, 1996.

[Kripke 1963] S. Kripke, Semantic Considerations on Model Logic. *Proceedings of a Colloquium: Modal and Many valued Logics,* volume 16 of *Acta Philosophica Fennica,* pp. 83-94, August 1963.

[Krolnik 1998] A. Krolnik, Cyrix M3 Phase 1 Report, Cyrix Inc. internal document, 1998

[Krolnik 1999] A. Krolnik, Cyrix M3 Phase 2 Report, Cyrix Inc. internal document, 1999.

[Kroph 1998] T. Kroph, Introduction to Formal Hardware Verification, Springer, 1998.

[Lachish et al. 2002] O. Lachish, E. marcus, S. Ur, A. Ziv, "Hole Analysis for Functional Coverage Data", *Proc. Design Automaction Conference,* 2002.

[Loh et al. 2004] L. Loh, H. Wong-Toi, C. N. Ip, H. Foster, D. Perry, "Overcoming the Verification Hurdle for PCI Express", *DesignCon,* 2004

Ludden et al. 2002] A. J. M. Ludden, W. Roesner, G. M. Heiling, J. R. Reysa, J. R. Jackson, B.-L. Chu, M. L. Behm, J. R. Baumgartner, R. D. Peterson, J. Abdulhafiz, W. E. Bucy, J. H. Klaus, D. J. Klema, T. N. Le, F. D. Lewis, P. E. Milling, L. A. McConville, B. S. Nelson, V. Paruthi, T. W. Pouarz, A. D. Romonosky, J. Stuecheli, K. D. Thompson, D. W. Victor, and B. Wile, "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems", *IBM Journal of Research and Development,* Vol. 46 Num. 1, 2002.

[Meyer 1992] B. Meyer,B. Meyer, Eiffel The Language, Prentice Hall, 1992

[Marschner et al. 2002] E. Marschner, B. Deadman, G. Martin, "IP Reuse Hardening via Embedded Sugar Assertions", *International Workshop on IP SoC Design,* October 30, 2002.

[Moorby et al. 2003] P. Moorby, A. Salz, P. Flake, S. Dudani, T.Fitzpatrick "Achieving Determinism in SystemVerilog 3.1 Scheduling Semantics", *Proceedings of DVCon 2003,* 2003. Retrived March 31, 2003, from the World Wide Web: http://www.eda.org/sv-ec/sv31schedsemantics-dvcon03.pdf

[Nacif et al. 2003] J. Nacif, F. de Paula, H. Foster, C. Coelho Jr., F. Sica, D. da Silva Jr., A. Fernandes, "An Assertion Library for On-Chip White-Box Verification at Run-Time", IEEE Latin American Test Workshop, 2003.

[OpenVera Language Reference Manual 2003] OpenVera Language Reference Manual. Retried December 22, 2003 from the World Wide Web:http:// www.open-vera.com.

[PCI-2.2 1998] PCI Local Bus Specification, Revision 2.2, PCI Special Interest Group, December 18, 1998.

[Piziali 2004] A. Piziali, *Verification Coverage Measurement and Analysis,* Kluwer Academic Publishers, 2004.

[Piziali and Wilcox 2002] A. Piziali, T. Wilcox, *Design Intent Diagram: Reasoning About Sources Of Bugs,* Unpublished presentation, 2002.

[Pnueli 1977] A. Pnueli, "The temporal logic of programs". *18th IEEE Symposium on foundation of Computer Science,* IEEE Computer Society Press, 1877.

[Richards 2003] J. Richards, "Creative assertion and constraint methods for formal design verification", *Proceedings of DVCon,* 2003.

[Riehle and Zullighoven 1996] D. Riehle, H. Zullighoven, *Understanding and Using Patterns in Software Development,* Retrived March 31, 2003, from the World Wide Web:http://citeseer.nj.nec.com/riehle96understanding.html.

[Shimizu et al. 2000] K. Shimizu, D. Dill, A. Hu, "Monitor-Based Formal Specification of PCI," *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design,* pp. 335-353, November 2000.

[Shimizu et al. 2001] K. Shimizu, D. Dill, C-T Chou, "A Specification Methodology by a Collection of Compact Properties as Applied to the Intel Itanium Processor Bus Protocol," *Proceedings of CHARME 2001,* Springer Verlag, pp. 340-354, 2001.

[Shimizu and Dill 2002] K. Shimizu, D. Dill, "Deriving a Simulation Input Generator and a Coverage Metric From a Formal Specification," *Proceedings of the 39-th Design Automation Conference,* June, 2002.

[Smith 2002] S. Smith, "Synergy between Open Verification Library and Specman Elite", *2002 Proceedings of Club Verification: Verisity Users' Group Meeting,* http://www.verisity.com/resources/vault/index.html

[Sutherland 2002] S. Sutherland, *The Verilog PLI Handbook,* Kluwer Academic Publishers, 2002.

[Tasiran and Keutzer 2001] S. S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs", *Design and Test of Computers,* pp. 36-45, July/August, 2001.

[Taylor et al. 1998] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, J. and C. Ramey, "Functional Verification of a Multiple-issue Out-of-order, Superscalar Alpha Processor—the DEC Alpha 21264 microprocessor", *Proc. Design Automation Conference,* pp. 638-643, June, 1998.

[Turing 1949] On checking a large routine. In "Report of a conference on high speed automatic calculating machines", pages 67-69, Univ. Math. Laboratory, Cambridge, 1949.

[Ur and Yadin, 1999] Ur, S. and Y. Yadin. "Micro Architecture Coverage Directed Generation of Test Programs." *Proc. Design Automation Conference,* 1999.

[Yuan, et al. 1999] J. Yuan, K. Shultz, C. Pixley, H. Miller, A. Aziz, "Modeling Design Constraints and Biasing in Simulation Using BDDs," *Proceedings of the IEEE International Conference on Computer Aided Design,* pp. 584-589, November 1999.

[Ziv 2002a] L. A. Ziv, "Using Temporal Checkers for Functional Coverage", *Proc. Intn'l Workshop on Microprocessor Test and Verification,* 2002.

[Ziv 2002b] L. A. Ziv, "Full Cycle Functional Coverage: Coverage Success Stories", *IBM Verification Seminar,* 2002.

[Ziv et al., 2001] Ziv, A, G. Nativ, S. Mittermaier, S. Ur. "Cost Evaluation of Coverage-Directed Test Generation for the IBM Mainframe." *Proc. Int'l Test Conference,* 2001.

Index

Symbols

$assert_always 105, 109
$assertkill 374
$assertoff 374
$asserton 374
$countones 95, 172, 355
$display 6, 108, 154, 373
$error 85, 373
$fatal 85, 373
$fell 355
$first_match 363
$info 86, 374
$isunknown 95, 166, 355
$onehot 355
$onehot0 355
$past 95, 185, 355
$rose 157, 355
$stable 355
$warning 85, 374
&& 79, 102
/*PSL*/ 83
//PSL 83
->80
|=> 78, 79, 94
|-> 77, 89, 94, 95

A

AHB Advanced High-Performance Bus 158
AMBA Advanced Microcontroller Bus Architecture 158
arbiter 38, 263
architect phase 127, 132
architect/design 14
assert_always 156
assertion knowledge 134
assertion pattern 162
assertions 3
  adding 12
  benefits 7, 30, 41, 48
    communication 20
    debug 20, 26
  capturing 36
  density 37, 39
  disabling 42
  enable 42
  error detection 4
  error isolation 4
  error notification 4
  false firing 104, 108, 111, 116
  groups 39
  history 4
  IEEE 1076-1993 VHDL 5
  infrastructure 26
  interfaces 9
  methodology 21
    existing designs 40
    new designs 30
  myths 11
  overconstrained 105
  performance 12, 26
  procedural 40
  results 6
  reviews 36
  RTL 5
  specify 37, 38, 39
  vs functiona coverage 131
assume-guarantee reasoning 51, 53
assumption 51

B

black-box 126
Boolean layer 62, 323

C

Cisco Systems, Inc. 6
clocks 40, 42
code coverage 125
code reviews 149
comments 35, 41
comparing captured data 194
complex multiplexer 248
complexity 49
compositional reasoning 49, 50
conditional expression 179, 180
consistency 18
constraint 50, 67
controllability 128, 129, 149
corner cases 139
counters 240
coverage metrics 125, 128
    ad-hoc 129
    arc 130
    functional 130
    programming code 129, 130
    state machine 130
coverage model 131, 132
CPU 143

D

deadlock 238
debugging 10, 12, 13, 37, 44
    time 8
decoded multiplexer 245
decoder 134
density 37, 39
design 127
design by contract 5
design intent 48
design knowledge 133
design phase 127, 132
design process 14
    regions of intent 15
Digital Equipment Corporation 6
direct mapped cache 226
directed tests 129, 148
documentation 11
documents
    design 28

E

e 150
ECC 53
Eiffel 5
encoder 249
event 65
event-bounded window 193

F

FIFO 34, 38, 43, 69, 70, 86, 125, 126, 132, 144, 156, 157, 167, 200, 202, 213, 214, 215, 216, 217, 218, 220
fixed depth pipeline register 219
forbidden 261
forbidden sequence 194, 195, 261, 262
forbidden sequences 279
formal 44, 48, 128
    exhaustive 51
    semi-exhaustive 50
formal specification 10
FSM 141, 237
functional correctness 129, 130
functional coverage 65, 72, 86, 125, 130, 131, 211
    analysis 138, 142, 144
    benefits 134, 135
    best practices 145
    bypass logic 140
    correctness 148
    cost 137
    cross 131
    debug logic 140
    density 138, 139, 141
    enable 155
    error detection 140
    forms 138, 150
    groups 143, 144, 146, 147, 155
    implementation level 126, 131
    internal sequences 140
    libraries 151
    logging 146
    methodology 137
    model 14, 131
        building 132
    organization 142
    ownership 132
    point 131
    post processing 154
    process 138
    reporting 151
    reports 147
    results 138
    sources 133
        assertions 134
        implementation 133

specification 133
testplan 133
specification level 126
specify 145
specifying 139
state machine 141
success stories 135
test generation 149
testplan 135
tracking 144
traffic patterns 140
usage 136
vs assertions 131
functional coverage points 126

G

global clock 171
gray-box 127
gray-code 172

H

Hewlett-Packard 6, 136, 138
high-level requirements 267, 284, 285
holes 129
HVL Hardware Verification Language 149, 150

I

IBM 6, 136
image computation 47
implementation assertions 283, 284
implementation phase 14, 15, 18, 133
implication
antecedent 73, 179
consequence 73, 179
in-order multiple request protocol 254
integration 9
Intel 136
Intel Corporation 6
interfaces 37, 39, 41
internal monitors 131

L

last in first out 222
libraries 36
linear-time temporal logic 46
linting 148
LSI Logic Corporation 6

M

memories 259
messages 43
methodology
assertion 21
coverage 126
modeling layer 62
Motorola, Inc 6
multi-cycle behavior 324

O

observability xviii, xx, 7, 8, 20, 127, 128, 129, 135, 149
one-hot 3, 7, 38, 41, 55, 63, 127, 164, 170, 172, 180, 217, 218, 245, 249, 250
opcode 51, 80, 173, 174, 238
Open Verification Library 2, 62, 150, 151, 152
customizing messages 294
firing 296
invariance 298
methodology 291
modules
cover_monitor 152
monitors 156
assert_always 37, 69, 73, 153, 158, 298, 303, 305, 307, 309, 311
assert_change 192
assert_cycle_sequence 79, 184, 189, 195
assert_frame 191,192
assert_never 69, 73, 165, 166, 300, 302
assert_next 73, 74, 188, 192
assert_one_hot 170
assert_quiescent_state 43, 119
assert_range 168
assert_time 192
assert_unchange 192
assert_width 192
assert_win_change 194
assert_win_unchange 194
assert_window 81, 82, 194
assert_zero_one_hot 172
task
ovl task.h 294
ovl_error 152
ovl_finish 296
ovl_init_msg 296
out-of-order protocol 197, 257

overflow 69, 70, 71, 213, 214

P

patterns 161
  assertions 162
  classification
    conceptual 163
    design 163
    programming 163
  definition 162
  elements 163
    considerations 164
    context 164
    motivation 163
    pattern name 163
    problem 163
    solution 164
  form
    Alexander 163
    assertions 163
    Gang of Four 163
  property structure 162, 210
  rationale 162, 210
  signals 164
  software 161
PCI 96
PCI Express 285, 286, 288
Pentium 4 136
pipeline protocol 202
PLI 154, 294
  callback 110
  calltf 105, 107, 113, 116
  checktf 105, 118
  consumer 113
  end of a simulation time slot 109
  end of compilation 109
  end of simulation 109, 119, 123
  expr_value_p 121
  mistf 109, 113, 121
  PLI 1.0 103
  PLI 2.0 103
  reason 109, 121
  REASON_FINISH 121
  REASON_ROSYNCH 109, 110
  routine
    acc_close 116
    acc_handle_tfarg 116
    acc_initialize 113, 116
    acc_vcl_add 116
    acc_vcl_delete 116, 117
    io_printf 107
    tf_dofinish 107
    tf_error 106
    tf_exprinfo 121
    tf_getinstance 116
    tf_getlongp 121
    tf_getp 107, 116
    tf_getworkarea 115
    tf_igetp 116
    tf_ispname 116
    tf_nump 106
    tf_rosynchronize 109
    tf_setworkarea 113
    tf_sizep 106
    tf_spname 107
    tf_strgettime 107
  s_tf_exprinfo 121
  s_vecval 121
  TF_STRING 118
  Value Change Link (VCL) 112
  verisuser.h 109, 121
PLI-based assertion 103
PowerPC 136
pragma-based assertions 82
prerequisite sequence 181
printf 107
priority encoder 251
priority multiplexer 246
procedural assertion 105, 108
processor-to-memory interface 175
progression testing 147, 148
proof algorithm 46
property 45, 62
  decomposition 49
  invariance 298
  liveness 46, 66
  safety 46
property checking 1
property specification 48
propositional temporal logics 46
PSL Property Specification Language 2, 46, 126, 154
  built-in function
    fell 82
    prev 82
    rose 82
  consecutive repetition operator 77
  coverage 138
  goto repetition 194, 272
  inclusive operator 81
  keyword
    abort 72, 74, 212, 332
    always 158, 333

assert 64, 70, 158, 336
assume 64, 336
before* 331
cover 65, 134, 154, 157, 160, 336
default clock 75, 154, 212
endpoint 332
eventually 80
if 333
never 70, 334
next 73, 330, 331
property 72
restrict 64
sequence 78, 154, 157, 160
until 80
until* 330
until_81
non-inclusive operator 81
sequence fusion 79
sequence length-matching AND 79, 102
strong operator 81
suffix next implication 78
terminating property 80
weak operator 81

Q

qualitative 129
quantitative 129
queue 38, 139, 150, 151, 213
quiescent state 43

R

random
effectiveness 135
random test 129
reachability analysis 47
reactive testbench 149
reasoning
assume guarantee 51
assume-guarantee 53
compositional 49, 50
recurring problem 162
recurring solution 162
regression 135
regression testing 147
reporting 42, 43
requirements 16
design 27
requirements model 282
reset 155
restriction 50
reuse 34
reviews
design 29
roles 19

S

S/390 processor 136
safety property 46
Sequential Extended Regular Expression 324
SERE 324
set associative cache 231
set patterns 173
severity level 42, 43, 153, 166 383
Silicon Graphics, Inc. 6
simple single request protocol 252
simulation 42, 128, 149, 155
cycle-based semantics 84, 111
event-based semantics 84
preponed region 86
time slots 85, 86, 104, 108
specification 138
ambiguity 268, 270, 271, 273, 280
architect 14, 15, 18, 19
Boolean layer 283
design 14, 15, 18, 19, 37
edge-sensitive active 272
end-to-end 282
eventual after 270
eventual next 269
formal 45
guidelines 278
higher-level 281
immediate after 270
immediate next 268
implement 14
implicit assumption 276
language 27
level-sensitive active 272
modeling layer 283
natural language 267, 273
nonoverlapping between 273
overlapping between 273, 274
partial 277
phases 14
specify 14, 15, 18
strong interpretation 274
temporal layer 283
verification layer 283

specification knowledge 133
specification phase 127, 132
specify once 150
SRAM controller 202
stack 222
standards
Accellera 6
IEEE 1076-1993 VHDL 5
IEEE 1364-2001 Verilog 6
state machine 38, 236
suffix sequence 184
sx1000 136
synthesis 35
SystemVerilog 2, 150, 153
BNF 374
concurrent assertion 84, 86
consecutive repetition 91
continuous invariant 84
coverage 138
cycle delay operator 88
dynamic variable 216
dynamic variables 94
immediate assertion 84
implication 364
keyword
and 359, 360
assert 85, 87, 369, 375
cover 87, 154, 157
disable iff 87, 212
ended 354, 362
endproperty 88, 354
endsequence 90, 376
first match 363
first_match 92, 354
intersect 360
matched 354, 363
not 87
or 360
property 87, 88, 157, 354
sequence 90, 376
throughout 93, 361
within 362
naming assertions 85
non-consecutive count repetitions 91
non-consecutive exact repetition 92
non-overlapped 94
overlapped implication 94
pass statement 85
property declaration 86
regular expression 88
repetition
consecutive 357
goto 272, 357
nonconsecutive 358
sequence expression 88
sequence intersection 360
severity task 85

T

table coverage 139
tagged transaction 194, 196, 197
temporal delay 356
temporal layer 62, 324
temporal logics
branching-time 46
linear time 46
propositional 46
testplan 133, 135
threshold 43
time slots 108
time-bounded window 190
tools
formal 10
semi-formal 10

U

underflow 69, 70, 71, 86, 213, 215
user-defined error messages 212

V

validation 30
Vera 150
verifiable contracts 9
verification
black-box 2, 126
efficiency 10
engineers 126
gray-box 127
testplan 19
white-box 3, 127
verification knowledge 133
verification layer 62

W

waterfall refinement 14
white-box 127

X

X detection 164, 167