CHAPTER

3

SPECIFYING RTL PROPERTIES

In this chapter, our goal is to introduce general concepts related to property specification. Then, we will apply these concepts as we introduce emerging RTL specification standards (that is, assertion libraries and languages). Initially, we compare and contrast the Accellera PSL 1.1 property specification language proposal [Accellera PSL-1.1 2004] with the Open Verification Library [Accellera OVL 2003]. We then introduce the Accellera SystemVerilog 3.1a assertion constructs [Accellera SystemVerilog-3.1a 2004]. Each of the assertion standards we discuss has its own merits. Our objective is to help the engineer understand the advantages (and limitations) of the various assertion forms and their usage model, while building a foundation of principles that we have found useful when specifying RTL properties. This will prepare readers to select appropriate specification forms that suit their needs (or preferences).

Current users might argue in favor of the subtle advantages they perceive their favorite assertion language possesses (possibly even a different language than what we present). Ultimately, what matters is that you simply choose an assertion language and use it. When you incorporate any form of RTL assertion in your design and verification process, you will significantly improve the overall quality of your design and dramatically simplify its verification debug effort.

Some readers might skip this chapter entirely, and jump directly to the *good stuff* (that is, the examples in Chapters 6, 7, and 8). However, at some point, these readers may find it helpful to return to this chapter to broaden their understanding of basic property language concepts and common definitions. These are covered in the beginning sections of this chapter. Our goal is to build a basic foundation of property specification and language concepts that is *useful for the RTL designer*, without delving too deeply into a theoretical discussion of automata theory.

try our assertion-based design concepts using the OVL

Many of the examples in this book use the OVL, which can be used with today's simulators. Our goal is to ensure that readers can implement and explore the various concepts we present using their existing IEEE-1364 Verilog or IEEE-1076 VHDL simulators. It is important for the reader to understand the basic assertion concepts we present in this book through working examples, such as the OVL. After mastering the general concepts we present, the reader is in position to quickly learn any property language or emerging standard for assertion specification.

3.1 Definitions and concepts

Before we introduce various forms for expressing assertions, it is helpful to consider definitions for two fundamentals: *property* and *event.* The reader should focus on the concepts presented in this section and not any specific syntax used to express these ideas. Details related to various assertion language syntax and semantics are discussed near the end of this chapter, as well as in the appendices.

3.1.1 Property

a property consists of a Boolean and temporal layer

Informally, a *property* is a general behavioral attribute used to characterize a design. More formally, we can define a property as:

> *A collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behavior* [Accellera PSL-1.1 2004].

When studying properties, it is generally easier to view their composition as four distinct layers:

- the *Boolean layer,* which is comprised of Boolean expressions (for example, Verilog or VHDL expressions)
- the *temporal layer,* which describes the relationship of Boolean expressions over time
- the *modeling layer,* which provides a means to model complex behavior of design inputs and outputs—as well as auxiliary logic that is not part of the design but often necessary for capturing higher-level requirements
- the *verification layer,* which describes how to use a property during verification

Defining (or partitioning) a property in terms of the abstract layer view enables us to dissect and discuss various aspects of properties. However, you will find that it is quite simple to express design properties. Thus, the four-layer view is merely a way to explain concepts and should not convey a sense that the actual language syntax is complex.[1]

To aid in studying property concepts, all examples in the following sections are presented using the Accellera PSL property specification language, unless otherwise noted.

Boolean layer

A property's Boolean layer is comprised of Boolean expressions composed of variables within the design model. For example, if we state that "*signal en1 and signal en2 are mutually exclusive*" (that is, a *zero-or-one-hot* condition in which only one signal can be active high at a time), then the Boolean layer description representing this property could be expressed in Verilog as shown in Example 3-1.

Example 3-1 A property's Boolean layer expressed in Verilog

```
!(en1 & en2) // enables are mutually exclusive
```

time ambiguity

Notice that we have not associated any time relationship to the statement: "*signal en1 and signal en2 are mutually exclusive*". In fact, the statement by itself is ambiguous. Is this statement *true* only at time 0 (as many formal tools infer), or is it *true* for all time?

Temporal layer

together, the Boolean and temporal layers form the basis of a property

A property's temporal layer permits us to describe the Boolean expressions' relationships to each other over time. Thus, all time ambiguities associated with a property are removed. For example, if *signal en1 and signal en2 are always mutually exclusive* (that is, for all time), then a temporal operator could be added to the Boolean expression to state precisely this. Temporal operators allow us to specify precisely when the Boolean expression must hold.[2] Example 3-2 demonstrates this point using the PSL temporal operator *always* combined with a Verilog Boolean experssion.[3]

1. The details of the *modeling layer* are discussed in Chapter 8 "Specifying Correct Behavior".

2. The term *hold* in this context means that the design exhibits behavior described by a specific Boolean expression, when the Boolean expression evaluates *true.*

Example 3-2 A property's temporal layer expressed in PSL

```
always (!(en1 & en2)) // enables are always mutually exclusive
```

We discuss additional PSL temporal operators used to specify relationships of multiple Boolean expressions over time in detail later in this chapter. And, we provide examples throughout the remainder of the book.

Verification layer

the verification layer for a property defines how to use it during verification

While a property's Boolean and temporal layers describe general behavior, they do not state how the property should be used during verification. In other words, should the property be asserted, and thus checked? Or, should the property be assumed as a constraint? Or, should the property be used to specify an event used to gather functional coverage information? Hence, the third layer of a property, which is the *verification layer,* states how the property is to be used.

Consider the following definitions for an assertion and a constraint.

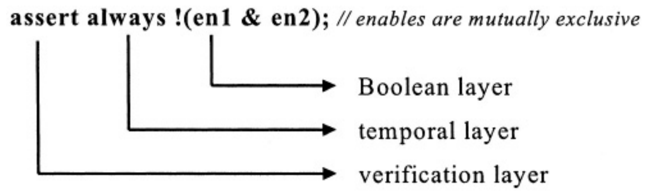
- *Assertion* - A given property that is expected to hold within a specific design. The PSL assert directive would be associated with the property to specify an assertion.
- *Constraint* - A condition (usually on the input signals) which limits the set of behavior to be considered during verification. A constraint may represent real requirements (e.g., clocking requirements) on the environment in which the design is used, or it may represent artificial limitations (e.g., mode settings) imposed in order to partition the verification. In this case, the PSL assume or restrict directives would be associated with the property to specify a constraint.

Constraints and assertions are described in further detail in Section 3.2.2.

Look again at the property *signal en1 and signal en2 are mutually exclusive.* Example 3-3 shows this property with the PSL assert directive. This states that the property is to be treated as an assertion during verification.

3. Note that a PSL *property* definition does not end with a semicolon (;), while *assertions* (which are built on top of properties) do end in a semicolon.

Example 3-3 Specifying a PSL property as an assertion for verification


assert always !(en1 & en2); // enables are mutually exclusive
Boolean layer
temporal layer
verification layer

3.1.2 Events

When discussing design properties in the context of verification (and in particular, simulation), it is helpful to understand the concept of a verification event. An *event* is any user-specified property that is satisfied at a specific point in time during the course of verification. A *Boolean event* occurs when a Boolean expression evaluates *true* in relation to a specified sample clock. A *sequential event* is satisfied at the end of a sequence of Boolean events.

In Example 3-4, if the sequence `c_mem_access` followed by `c_write` is satisfied during simulation (for example, at time unit 100), then we can claim that an event has occurred in our verification environment at that specific point in time. However, if the event is never satisfied, then our verification test was unable to verify some key aspect or functionality of our design. In other words, our testing and input stimulus was insufficient.

Example 3-4 A PSL functional coverage point

```
cover {c_mem_access; c_write};
```

The PSL cover directive permits the designer to designate the property as a *functional coverage point.* Chapter 5, "Functional Coverage" discusses additional aspects of creating functional coverage models through property specification.

3.2 Property classification

Properties are often classified in the context of their *temporal* and *verification* layers. Furthermore, properties can be classified by their *evaluation* method (that is, concurrent or sequential activation). This section describes the various classifications of properties.

3.2.1 Safety versus liveness

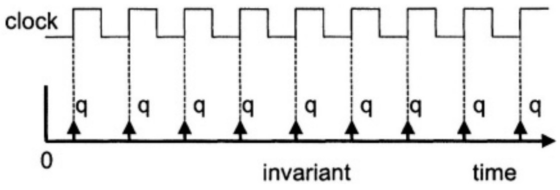
As previously defined, a property is a general behavioral attribute that is used to characterize a design. It is generally expressed in a format that enables us to reason about sequences of Boolean expressions over time. Hence, a property is often classified by its temporal layer. This section defines the two property classifications that are based on the temporal layer: *safety* and *liveness.*

invariant property

A *safety* property is also known as an invariant; which informally states that, for all time, nothing bad should happen. Thus, it is a property that must evaluate to *true* for all sample points of time. The sample point could be defined by either an explicit clock associated with the property or an inferred clock.

Figure 3-1 illustrates a safety property. The Boolean expression q in this example must always evaluate to *true* at every clock cycle.

Figure 3-1 **invariant property**



The arrows along the time axis represents the sampling of the Boolean expression q at every positive edge of signal clock (for this example). For a safety property, the sampled Boolean expression must always evaluate to *true,* which is represented by the up arrow.

liveness property

A liveness property is a property that specifies an eventuality that is unbounded in time. Loosely speaking, a *liveness* property claims that something good *eventually* happens. For example, the property *"whenever signal `req` is asserted, signal `ack` is asserted sometime in the future"* is a liveness property.

3.2.2 Constraint versus assertion

classifying properties as constraints or assertions

In addition to safety and liveness, a property can be classified according to its verification layer as either a *constraint* or an *assertion.*

One example of a *constraint* is a property that specifies the range of allowable values (or sequences of values) permitted on an input. The design cannot be guaranteed to operate correctly if its input value (or sequence of values) violates a specified constraint.

Alternatively, a property that describes that the expected design output behavior must remain valid or *true,* is an example of an *assertion.* For any permissible sequence of input values applied to a design (which means that all input constraints are satisfied), all assertions will evaluate to *true* if the design is functioning correctly.

The functions of constraints and assertions are dependent on the verification tool and environment. During *simulation,* both constraints and assertions can be treated as monitors (that is, dynamic property checkers) that check for compliance. During *formal verification,* constraints bound the static formal search engine to the design's legal input space, while assertions are treated as targets (that is, properties that must be proved) for formal analysis.

3.2.3 Declarative versus procedural

A *declarative property* describes the expected behavior of the design independent of its RTL procedural details. Hence, it is not necessary to understand the procedural code to understand the required expected behavior. On the other hand, a *procedural property* describes the expected behavior of the design in the current context (or frame of reference) at a particular line within the procedural code. Hence, it is necessary to understand the details of the procedural code to fully understand the expected behavior. Expressing interface properties declaratively is generally more natural than expressing these properties procedurally, since interface requirements are typically independent of the details of the block's implementation. However, capturing internal RTL implementation's design intent procedurally generally reduces the amount of extra code required to express these properties (particularly if the assertion is deeply nested with `case` and `if` statements).

concurrent versus sequential

A design model typically consists of a static, hierarchical structure, in which primitive elements interact through a network of interconnections. The primitives may be built-in simple functions (for example, gates) or larger, more complex procedural or algorithmic descriptions (for example, VHDL processes or Verilog always procedural blocks). Within a procedural description, statements execute in sequence. However, within the design as a whole, the primitives and communication interact concurrently.

Just as the design model itself involves a collection of concurrent elements (represented in a *declarative* fashion) and sequential elements (represented in a *procedural* fashion), properties may also be represented either as declarative or procedural. Hence, a *declarative assertion* is a statement (outside of a procedural context) that is always active and is evaluated concurrently with other layers or primitives in the design. A *procedural assertion,* on the other hand, is a statement within the context of a process (for example, a Verilog procedural block) that executes sequentially, in its turn, as the procedural code executes.

3.3 RTL assertion specification techniques

The system architect and verification engineer are instrumental in specifying global design assertions that must be verified (that is, independent of lower-level implementation details). However, quite often the verification engineer lacks sufficient in-depth knowledge of the implementation details to provide effective white-box assertion density. On the other hand, during the course of RTL development, the design engineer makes many lower-level assumptions about the design's environment as well as other implementation assumptions. Experience has shown that if design assumptions or concerns are not captured during the process of RTL implementation, then many lower-level implementation decisions, details, and properties are lost (that is, they will not be verified).

In this section, we demonstrate various forms for RT-level assertion specification. This includes the OVL [Accellera OVL 2003], PSL formal property language [Accellera PSL-1.1 2004], and SystemVerilog 3.1a assertion constructs [Accellera SystemVerilog-3.1a 2004]. Note that a more in depth discussion of these proposed standards is presented in Appendices A, B, and C.

3.3.1 RTL invariant assertions

The most basic form of RTL assertions is simple invariant (safety) properties, as discussed in Section 3.2.1. Examples of safety properties in RTL code include:

- it is never possible to overflow a specific FIFO
- it is never possible to read and write to the same memory address simultaneously
- it is never possible to generate an address out of range

In this section, we present examples of both an OVL and a PSL invariant assertion.

OVL invariant

Example 3-6 demonstrates the coding of invariant assertions directly into the RTL using the OVL for a simple FIFO circuit. This example is based on the UART 16550 core designed by Jacob and Mohor [2001]. The RTL assertion specifies that the blocks interfacing with the FIFO should never overflow or underflow the FIFO buffer. In other words, attempting to perform a *push* operation when the FIFO is full will result in an assertion violation. Similarly, it is a violation to perform a *pop* operation when the FIFO is empty.

OVL assert_never invariant

The Accellera OVL assert_never monitor, demonstrated in Example 3-6, accepts three arguments:

1. a clock expression
2. a reset expression
3. a user-specified Boolean expression

The Boolean expression is evaluated on every rising edge of the sample clock (when the reset signal is not active), and the monitor asserts that the Verilog Boolean expression will never evaluate to *true*. If the overflow assertion in Example 3-6 is violated, the following (default) error message is logged during simulation:

```
OVL_ERROR : ASSERT_NEVER : VIOLATION : : severity 0 : time 105 :top.my_FIFO.no_overflow
```

In addition to checking that a Verilog Boolean expression *never* evaluates to *true,* OVL provides the assert_always monitor to check that a Boolean expression *always* holds. For additional OVL feature details, such as customizing error message or severity levels, as well as an overview of additional monitors contained within the library, see Appendix A.

PSL invariant

specifying safety properties with PSL

A formal property language offers an alternative to instantiating assertion monitors directly in the RTL source (as demonstrated in Example 3-6). The expected behavior could be specified using a formal property language, such as the Accellera PSL formal property language. We could express the same overflow and underflow assertions using PSL, as demonstrated below in the Verilog Example 3-5.

Example 3-5 *PSL* overflow and underflow assertion

```
assert never (reset_n && {push,pop}==2'b10 && cnt==FIFO_depth)
  @(posedge clk);

assert never (reset_n && pop && cnt==0)
  @(posedge clk);
```

Example 3-6 Verilog FIFO overflow and underflow assertion example

```
module FIFO (data_out, data_in,
              clk, FIFO_clr_n, FIFO_reset_n,
              push, // push strobe, active high
              pop   // pop strobe, active high
             );
// FIFO parameters
   parameter FIFO_width = `FIFO_WIDTH;
   parameter FIFO_depth = `FIFO_DEPTH;
   parameter FIFO_pntr_w = `FIFO_PNTR_W;
   parameter FIFO_cntr_w = `FIFO_CNTR_W;
   output  [FIFO_width-1:0] data_out;
   input   [FIFO_width-1:0] data_in;
   input                    clk,FIFO_clr_n,
                            FIFO_reset_n, push, pop;
// FIFO buffer declaration
   reg [FIFO_width-1:0] FIFO[FIFO_depth-1:0];
// FIFO pointers
   reg [FIFO_pntr_w-1:0] top; // top
   reg [FIFO_pntr_w-1:0] btm; // bottom
   reg [FIFO_cntr_w-1:0] cnt; // count

`ifdef ASSERT_ON
   wire reset_n = FIFO_reset_n & FIFO_clr_n;

// OVL assert that the FIFO cannot overflow
   assert_never no_overflow (clk, reset_n,
        ({push,pop}= =2'b10 && cnt==FIFO_depth-1));

// OVL assert that the FIFO cannot underflow
   assert_never no_underflow (clk, reset_n,
        (pop && cnt= =0));
`endif
```

Example 3-6 Verilog FIFO overflow and underflow assertion example

```
  always @ (posedge clk or negedge FIFO_clr_n)
 // Clear FIFO content and reset control
   if (!FIFO_clr_n) begin
       top <= 0;
       btm <= 0;
       cnt <= 0;
       for (i=0; i<FIFO_depth; i=i+1)
          FIFO[i] <= 0;
   end
 // reset FIFO control
   else if (!FIFO_reset_n) begin
       top <= 0;
       btm <= 0;
       cnt <= 0;
   end
   else
     case ({push, pop})
       2'b10  : // WRITE
         begin // assertion checks for overflow
           FIFO[top] <= data_in;
           top <= top + 1;
           cnt <= cnt + 1;
         end
       2'b01 : //READ
         begin // assertion checks for underflow
           btm <= btm + 1;
           cnt <= cnt + 1;
         end
       2'b11  : // WRITE & READ
         begin
           FIFO [top]  <= data_in;
           btm <= btm + 1;
           top <= top + 1;
         end
       endcase
 // end always

   assign data_out = FIFO[btm];

 endmodule
```

Note that the PSL assertion is a declarative form of specification, which is independent of any hardware description language. In other words, at the Boolean layer PSL supports Verilog, VHDL or technically any HDL. The basic syntax for expressing a PSL never assertion (related to a Boolean expression) is shown in Example 3-7.[4]

Example 3-7 *PSL* assertion syntax

```
 assert never (<Boolean expression>)  [@<clock expression>];
```

4. For a more in-depth description of the PSL never operator, particularly related to specifying sequences and properties, see Appendix B.

Note that in Example 3-5, we demonstrated this assertion using a Verilog Boolean expression. For VHDL code, the appropriate VHDL Boolean expression syntax would be used.

In addition to checking that a Boolean expression never holds, PSL provides the means for checking that a Boolean expression *always* holds with the assert always keywords. For additional details, see Appendix B.

3.3.2 Declaring properties with PSL

As previously stated in this chapter, a *property* specifies a behavior of the design. Once defined, a property can be used in verification as an *assertion* (a property that is checked), a *functional coverage* specification (a property the must evaluate to true during verification), or a *constraint* (a property that limits the verification input space).

PSL allows you to define named property declarations with optional arguments, which facilitates property reuse. These parameterized properties can then be instantiated in multiple places in your design with unique argument values. A property can be referenced by its name.

For example, we could specify that a and b are mutually exclusive whenever reset_n is not active as follows:

Example 3-8 *PSL* property declaration example

```
property mutex (boolean clk, reset, a, b) =
     always (!(a & b )) @(posedge clk) abort !reset_n;
```

reset condition

The abort clause allows you to specify a reset condition. If the abort Boolean expression becomes true at any time during the evaluation of the assertion expression, then the property holds regardless of the assertion expression evaluation.

In Example 3-9, we now create a PSL assertion for a design property where write_en cannot occur at the same time as a read_en:

Example 3-9 *PSL* assertion for mutex write_en & read_en

```
property mutex (boolean clk, reset, a, b) =
     always (!(a & b )) @(posedge clk) abort !reset_n;

assert mutex(clk_a, master_rst_n, write_en, read_en);
```

3.3.3 RTL cycle related assertions

In this section, we demonstrate how to express assertions involving multiple Boolean expressions using the PSL next temporal operator and the OVL assert_next monitor.

PSL next operator

The OVL assert_always and assert_never monitors, and the PSL always and never temporal operators, allow us to specify an invariant (that is, a condition that must hold or must not hold for all cycles). Additional OVL monitors and PSL operators allow us to be more specific about specifying cycle timing relationships. For instance, the PSL next operator allows us to specify the cycle relationship between consecutive events (that is, the relationship between Boolean or temporal expressions). Thus, the PSL assertion in Example 3-10 states that whenever the signal `req` holds, then the signal `ack` must hold on the next cycle.

Example 3-10 ***PSL* next operator**

```
assert always (req -> next ack) @(posedge clk);
```

PSL Boolean implication

Note the use of the PSL Boolean implication operator `->`. In math, the implication operator consist of an *antecedent* that implies a *consequence* (for example, `A -> C,` which reads `A` implies `C`). If the antecedent is true, then the consequence must be true for the implication to pass. If the antecedent is false, then the implication passes regardless of the value of the consequence.

Continuing our example, if the `ack` is expected to hold on the 3rd cycle after the `req`, then the assertion would have to be coded in a more complicated form, as shown in Example 3-11.

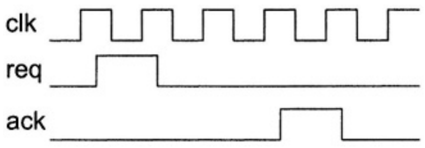
Exlample 3-11 ***PSL* multiple next**

```
assert always (req -> next (next (next ack))) @(posedge clk);
```

PSL next repetition operator

Although the specification for multiple next cycles shown above is valid, PSL provides a more succinct mechanism that utilizes the repetition operator [i], where *i* is a constant value. As shown in Figure 3-2, next[3] states that the operand is required to hold at the 3rd next cycle (rather than at the very next cycle).

Figure 3-2 **assert always (req -> next[3] ack) @(posedge clk);**



OVL assert_next

The Accellera OVL assert_next monitor has semantics that are similar to the PSL next operators, as shown in the following PSL assertion.[5]

Example 3-12 *PSL* abort operator

```
assert always (req -> next ack) @(posedge clk) abort !reset_n;
```

However, this same example could be coded in the RTL by instantiating a Verilog OVL assert_next monitor as shown in Example 3-13.

Example 3-13 *OVL* assert_next

```
assert_next my_req_ack (clk, reset_n, req, ack);
```

The OVL equivalent of the PSL assertion demonstrated in Figure 3-2 is demonstrated below in Example 3-14. Recall that this specifies that an `ack` must occur exactly three cycles after a `req`.

Example 3-14 *OVL* assert_next with 3 *number of clocks* parameter

```
assert_next #(0,3) my_req_ack (clk, reset_n, req, ack);
```

In Example 3-14, the #(0,3) parameters represent the *severity level* (0) and *number of clocks* (3) required for the sequence. A severity level of 0 is the highest severity, which will cause simulation to halt. And, the `ack` signal must be satisfied three clocks after `req`, as specified by the *number of clocks* parameter. For additional details on the OVL assert_next parameter options, see Appendix A.

3.3.4 PSL and default clock declaration

PSL provides a means for specifying a *default clock* expression, which enables you to define a property or sequence without explicitly specifying a clock. For example, we could re-write Example 3-12 using a default clock declaration as shown in Example 3-15:

5. The PSL abort operator specifies a condition that removes any obligation for a property to hold. The left operand of the abort operator is the property to be aborted. The right operand of the abort operator is the Boolean condition that causes the abort to occur.

Example 3-15 *PSL* default clock

```
default clock = (posedge clk);
assert always (req -> next ack) abort !reset_n;
```

For additional details on the default clock syntax, see Appendix B.

For simplicity, we have coded many of the PSL examples throughout the book without an explicit clock. For these examples, you can assume that a default clock was previously defined—just like we did in Example 3-15.

3.3.5 Specifying sequences

In this section, we discuss specifying sequences with PSL and OVL. First, we explore the power of PSL to support a concise coding style. Then, we demonstrate how the OVL is implemented to check sequences.

Checking sequences with PSL

sequences of Boolean expressions

The basic PSL temporal operators described in the previous sections (that is, always, never, and next) can be combined to create complicated assertions. However, writing such assertions is sometimes cumbersome, and reading (and understanding) complicated assertions can be equally difficult.

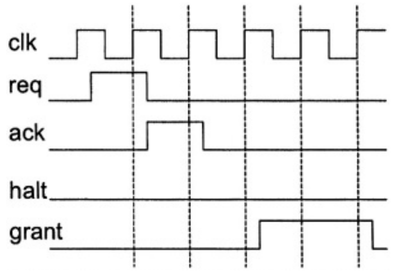
The assertion shown in Example 3-16 states that the following sequence must occur:

- if signal `req` is asserted
- and then in the next cycle, signal `ack` is asserted
- and then in the following cycle signal `halt` is not asserted
- then, starting at that cycle, signal `grant` is asserted for two consecutive cycles

Example 3-16 *PSL* sequence specified with the next operator

```
assert always
   (req -> next(ack -> next (!halt -> (grant & next grant))));
```

Figure 3-3 **Sequence.**



sugar extended regular expressions (SEREs)

PSL provides an alternative way to reason about sequences of Boolean expression that is more concise and easier to read and write. It is based on an extension of regular expressions, called *sugar extended regular expressions,* or SEREs.

SEREs describe series of Boolean events by specifying a sequence for which each Boolean expression in the series must hold over contiguous cycles, A rudimentary SERE is a single Boolean expression describing a Boolean event at a single cycle of time. More complex sequences of Boolean expressions can be constructed using the SERE concatenation operator (;). Example 3-17 shows the specification that three Verilog Boolean expressions `A&B`, `C|D`, and `E^F` must hold consecutively.

Example 3-17 ***PSL*** **SERE - sequence of Boolean expressions**

```
{A&B; C|D; E^F}
```

The sequence is matched if the following three assertions hold:

- on the first cycle, the Boolean expression `A&B` holds
- on the second cycle, the Boolean expression `C | D` holds
- on the last cycle, the Boolean expression `E^F` holds

Often, an implication operator is used to start the sequence. Thus, if our specification states: *if signal* `req` *is asserted, then in the next cycle, signal* `ack` *must be asserted, and in the following cycle, signal* `halt` *must not be asserted,* then the property could be written using a SERE as shown in Example 3-18.

Example 3-18 ***PSL*** **SERE for req, ack, !halt sequence**

```
assert always (req -> next {ack; !halt});
```

Note that if the req signal does not hold, then the sequences (defined by the SERE) that start in the cycle immediately after req are not required to hold.

SERE [*] and [+] consecutive repetition operators

Repetitions within sequences. Like regular expressions found in most scripting languages (such as Perl or TCL), PSL allows the user to specify repetitions when specifying sequences of Boolean expressions (SEREs). For instance, the SERE *consecutive repetition operator* `[*m:n]` describes repeated consecutive concatenation of the same Boolean expression (or SERE) that is expected to hold between m and n cycles, where m and n are constants.

If neither of the range values are defined (that is, `[*]`) then the SERE is allowed to hold any number of cycles, including zero. Hence, the empty sequence is allowed. Also note that the repetition operator `[+]` is shorthand for the repletion `[*1:inf]`, where the inf keyword means infinity.

For example, the SERE `{a; b[*]; c[3:5]; d[+]; e}` describes the following sequences:

- the Boolean variable `a` holds on the first cycle of the sequence
- and then, on the following cycle, there must be zero or more `b`'s that must hold
- this is followed by three to five `c`'s that must hold
- which is followed by one or more `d`'s that must hold
- which is finally followed by `e` that must hold.

Sequence implication. In Example 3-18, we demonstrated a simple Boolean implication, in which a Boolean expression implied a sequence. Often, it is desirable for the completion of one sequence (that is, a prerequisite sequence) to imply either a property or another sequence. Hence, the *suffix implication* family of PSL operators enables us to specify this type of behavior.

The PSL *suffix implication* operator |-> can be read as: *If the left hand side prerequisite sequence holds, then the right hand side sequence (or property) must hold.* The | character symbolizes the completion of the prefix sequence, which is then followed by the implication operator ->, implying the suffix sequence.

Let us reconsider Example 3-16. Suppose that the two cycles of `grant` should start the cycle after `!halt`. We could code this as shown in Example 3-19.

Example 3-19 ***PSL*** **suffix implication**

```
assert always ({req; ack; !halt} |-> {1; grant, grant});
```

Or, we can simplify the code by using the repetition operator as shown in Example 3-20.

Example 3-20 *PSL* repetition operator

```
assert always ({req; ack; !halt) |-> 1; grant[2]});
```

Note that the last Boolean expression in the prerequisite sequence overlaps (occurs at the same time as) the first Boolean expression in the suffix sequence. In other words, the `!halt` Boolean expression in the prerequisite SERE overlaps with the first item in the suffix SERE. Hence, we add the 1 *(true)* Boolean expression for this overlap, which moves time forward by one cycle. An alternative way to code Example 3-19 is shown in Example 3-20.

Example 3-21 *PSL* suffix implication with next operator

```
assert always ({req; ack; !halt} |-> next {grant[2]});
```

However, PSL provides a simpler way to do this using the *suffix next implication* operator |=>. The |=> operator takes us forward in time by one clock cycle, which permits us to specify the property in Example 3-19 as shown in Example 3-22.

Example 3-22 *PSL* suffix next implication

```
assert always ({req; ack; !halt} |=> {grant[2]});
```

Declaring sequences within PSL

In PSL, sequences can be declared and then reused with optional parameters.

For example, we could define a *request-acknowledge* sequence with parameters that allow redefining the `req` and `ack` variables as follows:

Example 3-23 *PSL* sequence declaration

```
sequence req_ack (req, ack) = {req; [*0:2]; ack};
```

Once defined, a sequence can be reused and referenced by name within various PSL properties.

Sequence operators within PSL

PSL provides a number of sequence operators, useful for composing sequences. In this section we introduce the *sequence fusion* operator and the *sequence length matching AND* operation. The additional operators are described in Appendix B.

Sequence fusion (:). The *sequence fusion* operator (**:**), constructs a SERE in which two sequences overlap by one cycle. That is, the second sequence starts at the cycle in which the first sequence completes. For example, to specify that an active `grant` overlaps with the last active `req` in a set of sequences of one, two, three, or four consecutive `req` signals, we would code:

Example 3-24 ***PSL*** `grant` **overlapping very last** `req`

```
{req[*1:4}:{grant}
```

Sequence length matching AND (&&). The *sequence length-matching AND* operator (&&) constructs a SERE, in which two sequences both hold at the current cycle, and furthermore both complete in the same cycle. An example of the sequence length matching AND operator is demonstrated in Example 3-51 PSL PCI basic read transaction on page 102.

Checking sequences with the OVL

The OVL assert_cycle_sequence monitor is a useful way to check a contiguous sequence of Boolean expressions. In the Verilog OVL, the sequence `A`, followed by `B`, followed by `C`, followed by `D`, would be expressed as a concatenation operation `{A,B,C,D}`. This expression is then passed on as the *event_sequence* argument to the assert_cycle_sequence monitor. Note that the monitor can be configured (as an option) to check either of the analogous PSL properties shown in Example 3-25 and Example 3-26.

Example 3-25 ***PSL*** **Boolean expression implies sequence**

```
 always ({A} |=> {B,C,D})
```

Example 3-25 states that the occurrence of the Boolean expression `A` implies the sequence `{B,C,D}` starting on the next cycle, or

Example 3-26 PSL Sequence implies Boolean expression

```
 always ({A,B,C} |=> {D})
```

Example 3-26 states that the occurrence of sequence `{A,B,C}` implies the Boolean expression `D` on the next cycle (see Appendix B for details).

Another example, Example 3-27, asserts that when a `'WRITE` cycle starts, which is then followed by a `'WAIT` statement, then the next opcode must have the value `'READ`.

Example 3-27 ***OVL*** **assert_cycle_sequence**

```
assert_cycle_sequence #(0,3) init_test (clk,1,
          {r_opcode == 'WRITE,
           r_opcode == 'WAIT,
           r_opcode == 'READ});
```

This is analogous to the PSL assertions shown in Example 3-28.

Example 3-28 ***PSL*** **opcode sequence**

```
assert always
   ((r_opcode=='WRITE) -> next {r_opcode=='WAIT; r_opcode=='READ});
```

3.3.6 Specifying eventualities

PSL eventually! operator

In section 3.3.3, "RTL cycle related assertions", we introduced the next operator, which allows us to move forward exactly one cycle. However, we might not want to explicitly specify (or we may not know) the exact timing relationship between multiple events. Hence, the PSL eventually operator allows us to move forward without specifying exactly when to stop. The assertion that *whenever a* `req` *is asserted, then an* `ack` *will eventually be asserted* would be coded in PSL as shown in Example 3-29.

Example 3-29 ***PSL*** **eventually operator**

```
assert always (req -> eventually ack);
```

PSL until operator

The PSL until operator provides another way to reason about a future point in time, while specifying a requirement on a Boolean expression that must hold for the current cycles moving forward (that is, until a *terminating property* holds). See Example 3-30.

Example 3-30 ***PSL*** **until operator**

```
assert always (req -> next (!req until ack));
```

This assertion states that whenever signal `req` is asserted, then starting at the next cycle, signal `req` will be de-asserted until signal `ack` is asserted. For this example, Boolean expression (that is, signal) `ack` is the terminating property.

up to but not necessarily including

The until operator is a *non-inclusive* operator; that is, it specifies the left operand holds up to, but not necessarily including, the cycle where the right operand terminating property holds. As such, the sub-property `(!req until ack)` specifies that `req` will be de-asserted up to, but not including, the cycle where `ack` is asserted. Thus, if signal `ack` is asserted immediately after the cycle in which the signal `req` is asserted, then the de-assertion of `req` is not required.

up to and including

Alternatively, the until_operator is an *inclusive* operator; that is, it specifies the left operand holds up to and including the cycle where the right operand terminating property holds. Thus, if the req signal is required to be de-asserted (that is, `!req`) at least one cycle after the initial `req`, then until_ would be used to specify this property as shown in Example 3-31.

Example 3-31 ***PSL*** **event bounded window pattern**

```
assert always (req -> next (!req until_ ack));
```

Example 3-31 states that whenever signal `req` is asserted, then `!req` will be asserted during the next cycle (whether or not `ack` is asserted), and it will remain asserted through (and including) the cycle where `ack` is asserted.

weak versus strong operators

One additional note concerning the PSL eventually, until and until_ operators: these are known as *weak* operators. A *weak* operator makes no requirements about the terminating condition, while a *strong* operator requires that the terminating condition eventually occur. For example, the `ack` signal in Example 3-31 is not required occur prior to the end of verification (for example, at the end of simulation) for the weak until_ operator. The eventually!, until!, and until!_ are all strong operators. For additional details concerning *strong* (!) and *weak* operators, see Appendix B.

OVL event bounded window checkers

OVL assert_window

The OVL contains a set of *event bounded* window checkers that permit us to specify an eventuality class of assertions. This allows an assertion similar to Example 3-30 to be captured in the RTL using an OVL assert_window monitor, as shown in Example 3-32.

Example 3-32 *OVL* event bounded window

```
assert_window req_ack (clk, reset_n, req, !req, ack) ;
```

For additional details concerning the assert_window, as well as other OVL event bounded (and time bounded) window checkers, see Appendix A.

3.3.7 PSL built-in functions

PSL contains a number of built-in functions that are useful for modeling complex behavior. In this section, we describe the prev(), rose(), and fell(), which are used throughout various example in the book.

- prev (bit_vector_expr [, number_of_ticks])

  returns the previous value of the `bit_vector_expr`. The `number_of_ticks` argument specifies the number clock ticks used to retrieve the previous value of `bit_vector_expr`. If `number_of_ticks` is not specified, then it defaults to one.
- rose (boolean_expr)

  The built-in function rose() is similar to the posedge event control in Verilog. It takes a Boolean signal as an argument and returns a 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.
- fell (boolean_expr)

  The built-in function fell() is similar to negedge in Verilog. It takes a Boolean signal as an argument and returns a true if the argument's value is 0 at the current cycle and 1 at the previous cycle, with respect the clock of its context, otherwise it is false.

3.4 Pragma-based assertions

A number of proprietary, vendor- and tool-specific approaches have been developed in recent years that provide designers the ability to embed verification assertions in their RTL design code. Because of a lack of standardization within this area, many of these approaches must rely on text-macros [Bening and Foster 2001], or meta-comment mechanisms to specify design assertions. These are followed by a pre-processing step that attempts to model the assertion semantics in HDL (or via PLI) in a way that is transparent to the user. One attractive feature with either the text-macro or meta-comment approach is that they have no side effect on existing tools that read the RTL code (that is, they can be ignored by existing tools as comments). Furthermore, this approach permits the designer to embed new assertion languages within an existing HDL standard.

The Accellera PSL formal property language permits us to specify assertions of the design independent of the implementation code; that is, the HDL code. However, it is often desirable to capture assertions directly in the HDL source code during RTL implementation. To achieve inter-operability, many companies have coordinated an effort for embedding assertions within RTL code using the common pragma approach. Example 3-33 illustrates a technique that embeds PSL assertions directly within Verilog code.

Example 3-33 *PSL* embedded pragma

```
// PSL assert always (req -> next ack) ;
```

For PSL properties and assertions spanning multiple lines, Example 3-34 demonstrates a recommended technique.[6]

Example 3-34 *PSL* embedded pragma spanning multiple lines

```
/* PSL
  property req_ack = always (req -> next ack);
  assert req_ack;
*/
```

It is encouraging to see that multiple EDA vendors are working together to ensure inter-operability for PSL embedded assertions. However, we recommend that you consult your specific tool's documentation prior to embedding a PSL assertion into your RTL code. Please note that the syntax for a PSL property and assertion is fixed through the Accellera standard. However, the embedding mechanism across multiple design languages (for example, a pragma syntax) is outside the scope of the PSL formal property language.

6. Note in this example we declared a PSL *named property* (req_ack) and then *assert* the property in a separate statement.

3.5 SystemVerilog assertions

SystemVerilog 3.1a [Accellera System Verilog-3.1a 2004] recommends extensions to the IEEE-1364 Verilog language that permit the user to specify assertions declaratively (that is, outside of any procedural context) or directly embedded within procedural code. In addition, SystemVerilog supports two forms of assertion specification: *immediate* and *concurrent.*

In this section, we focus on a small set of common SystemVerilog operators that we use in examples throughout the book. Details for all the SystemVerilog operators are covered in Appendix C, "SystemVerilog Assertions" on page 353.

immediate event-based semantics

Immediate assertions evaluate using simulation *event-based semantics,* similar to other procedural block statements in Verilog. There is a danger of semantic inconsistency between the evaluation of immediate assertions in simulation versus formal property checkers, since formal tools generally evaluate assertions using *cycle-based semantics* (that is, sampled off of a clock or signals) versus simulation event-based semantics. In addition, there is a risk of false firing associated with immediate assertions in simulation, which is discussed later.

concurrent cycle-based semantics

Concurrent assertions are based on clock semantics and use sampled values of variables (note, this is similar to the OVL clock semantics). All timing glitches (real or artificial due to delay modeling and transient behavior within the simulator) are abstracted away.

For a detailed discussion of SystemVerilog scheduling and semantics related to assertion evaluation, we recommend Moorby et al. [2003].

3.5.1 Immediate assertions

use immediate assertions with caution

Immediate assertions (also referred to as *continuous invariant assertions*) derive their name from the way they are evaluated in simulation. In a procedural context, the test of the assertion expression is evaluated *immediately,* instead of waiting until a sample clock occurs. When the variables in the assertion expression change values in the same simulation time slot, due to transient scheduling of events within a zero-delay simulation model, a *false firing* may occur if standard Verilog event scheduling is used. To prevent this class of false firings, evaluation of the assertion expression must wait until all potential value changes on the variables have completed (that is, the transient behavior of events in the simulation has reached a steady state). Hence, SystemVerilog 3.1a has proposed a new region within the simulation scheduler's time slot called the *observe region,* which evaluates after the non-blocking assignment (NBA) region—ensuring that assertion expression variable values have reached a steady state [Moorby et al. 2003]. Note that this is similar to performing a PLI *read-only synchronization* callback to get to the end of the time slot region for safe evaluation. However, there is still a potential for false firings across multiple simulation time slots with immediate assertions, often due to a testbench driving stimulus into the DUV and delay modeling. For a detailed discussion of this problem in the context of PLI routines, see section 4.1.4 "False firing across multiple time slots" on page 116.

The syntax for the SystemVerilog immediate assertion is defined as follows:

Syntax 3-1 *SystemVerilog* immediate assertions

```
// See Appendix C for additional details

immediate_assert_statement ::=assert ( expression ) action_block
action_block        ::=  statement   [ else statement_or_null ]
                       | [statement_or_null] else statement_or_null
statement_or_null  ::=statement  |   `;'
```

Note that the SystemVerilog assert statement is similar to a Verilog if statement. For example, if the assertion expression evaluates to true, then an optional *pass statement* is executed. If the pass statement is omitted, then no action is taken when the assertion expression evaluates to true. Alternatively, if the assertion expression evaluates to 1'bx, 1'bz, or 0, then the assertion fails and the optional else *fail statement* is executed. If the optional fail statement is omitted, then a default error message is printed for whenever the assertion expression evaluates to false.

naming assertions

The optional assertion label (identifier and colon) associates a name with an assertion statement. And it can be displayed using the %m format code.

severity levels

SystemVerilog has created a new set of system tasks (also referred to as severity task) that are similar to the Verilog $display system task. These new tasks convey the severity level associated with an assertion's `action_block` while printing any user-defined message. The new severity tasks are:

- $fatal, which reports a Run-time Fatal severity level and terminates the simulation with an error code.
- $error, which reports a Run-time Error condition and does not terminate the simulation. Note that if the optional fail state is omitted, the `$error` is the default severity level.
- $warning, which reports a Run-time Warning severity level and can be suppressed in a tool-specific manner.

- $info, which reports any general assertion information, carries no specific severity, and can be used to capture functional coverage information during runtime.

The details and syntax for these system tasks are described in Appendix C.

Example 3-35 demonstrates a SystemVerilog *immediate* assertion for our previous FIFO example:

Example 3-35 *SystemVerilog* queue underflow check

```
always @ (push or pop or cnt or reset_n)
  if (reset_n)
    if ({push, pop}==2'b01)
      underflow_check: assert (cnt!=0) else
                          $error("underflow error at %m");
```

3.5.2 Concurrent assertions

SystemVerilog concurrent assertions describe behavior that spans time. The evaluation model is based on a clock such that a concurrent assertion is evaluated only at the occurrence of a clock tick. SystemVerilog 3.1a has proposed a new region within the simulation scheduler's time slot, called the *preponed region,* which evaluates at the beginning of a simulation time slot. Hence, the values of variables used in the concurrent assertion expression are sampled at the start of a simulation time slot and then the concurrent assertion is evaluated using the preponed sampled values in the time slot observe region. Further details on concurrent assertion sampling are described in Moorby et al. [2003].

3.5.2.1 Property declaration

reusing properties

As previously stated in this chapter, a *property* specifies a behavior of the design. Once defined, a property can be used in verification as an *assertion* (a property that is checked), a *functional coverage* specification (a property the must occur during verification), or a *constraint* (a property that limits the verification input space).

SystemVerilog allows you to define named property declarations with optional arguments, which facilitates property reuse. These parameterized properties can then be instantiated in multiple places in your design with unique argument values. A property can be referenced by its name. A hierarchical name can be used consistent with the System Verilog naming conventions.

For example, we could specify that `a` and `b` are mutually exclusive whenever `reset_n` is not active as follows:

Example 3-36 *SystemVerilog* property declaration example

```
property mutex (clk, reset_n, a, b);
    @(posedge clk) disable iff (reset_n) (!(a & b ));
endproperty
```

reset condition

The disable iff clause allows you to specify asynchronous resets. If the disable Boolean expression becomes true at any time during the evaluation of the assertion expression, then the property holds regardless of the assertion expression evaluation. SystemVerilog also supports the specification of properties which must never hold, using the not construct. Effectively, the not construct negates the property expression. For example, we re-code the previous example as follows:

Example 3-37 *SystemVerilog* property declaration example with not

```
property mutex_with_not (clk, reset_n, a, b);
    @ (posedge clk) disable iff (reset_n) not (a & b);
endproperty
```

See Appendix C for specific details on SystemVerilog property syntax.

3.5.2.2 Verifying concurrent properties

After declaring a property, a verification directive assert or cover can be used to state how the property is to be used. The SystemVerilog verification directives include:

- assert—which specifies that the property is to be used as an assertion (that is, a property whose failure is reported during verification).
- cover—which specifies that the property is to be used as a functional coverage specification (that is, a property whose occurrence is reported during verification).

In Example 3-36, we now create a concurrent assertion for a design property where write_en cannot occur at the same time as a read_en:

Example 3-38 ***SystemVerilog* assertion for mutex write_en & read_en**

```
property mutex (clk, reset_n, a, b);
    @(posedge clk) disable iff (reset_n) (!(a & b));
endproperty

assert_mutex: assert property (mutex(clk_a, master_rst_n,
                                      write_en, read_en));
```

Example 3-39 demonstrates an alternative form of directly specifying the same SystemVerilog concurrent assertion:

Example 3-39 ***SystemVerilog* simple concurrent assertions**

```
assert_mutex:  assert property @ (posedge clk_a)
    disable iff (master_reset_n) (!(write_n & read_en));
```

Note that a concurrent assertion may be used directly within procedural code, or alternatively stand alone as a declarative assertion within a module (that is, outside of procedural code).

See Appendix C for specific details on SystemVerilog assert and cover syntax.

3.5.2.3 SystemVerilog sequences

sequences of Boolean expressions

A *sequence* is a finite series of Boolean events, where each expression represents a linear progression of time. Thus, a sequence describes a specific behavior. A SystemVerilog *sequence expression,* like the PSL SERE previously discussed, describes sequences using *regular expressions.* This enables us to concisely specify a range of possibilities for when a Boolean expression must hold.

Example 3-40 shows how we use SystemVerilog to concisely describe the sequence "*a request is followed three cycles later by an acknowledge*".

Example 3-40 ***SystemVerilog* sequence expression with fixed delay**

```
req ##3 ack
```

specifying cycle delays

In SystemVerilog, the ## construct is referred to as a *cycle delay* operator. The number after the ## construct represents the cycle in which the right-hand side Boolean event must occur with respect to the left-hand Boolean event. For the case ##0, both the left- and right-hand Boolean events overlap in time (that is, they occur in parallel).

We can specify a time window with a cycle delay operation and a range. Example 3-41 uses SystemVerilog to describe the sequence "*a request is followed by an acknowledge within two to three cycles*".

Example 3-41 *System Verilog* sequence expression with a range of delays

```
req ##[2:3] ack
```

The previous examples are referred to as *binary delays* (that is, a delay between two Boolean expressions). SystemVerilog also permits us to specify *unary delays* (that is, a Boolean expression that begins with a delay). Examples of unary delays are as follows:

Example 3-42 *SystemVerilog* unary delays relationship to binary delays

```
(##0 start)     // that is, (start)
(##1 start)     // that is, (1'b1 ##1 start)
(##[1:2] start)  // that is, (1'b1 ##1 start) or (1'b1 ##2 start)
```

Note that unary delays are useful when associated with implication. For example, if we want to describe a sequence in which a `req` must be followed by a `ack` within two to three cycle, which is then followed by a `gnt`, we would write it as follows (using the SystemVerilog implication operator |->):

Example 3-43 *SystemVerilog* unary delays relationship to binary delays

```
req |-> ##[2:3] ack ##3 gnt
```

Sequence declaration

In SystemVerilog, sequences can be declared and then reused with optional parameters, as shown in Syntax 3-2.

Syntax 3-2 ***SystemVerilog* sequence**

```
// See Appendix C for additional details

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.
```

You can replace expression names within the sequence expression via parameters specified through the sequence_formal_list. This enables us to declare sequences and reuse them in multiple properties. For example, we could define a request-acknowledge sequence with parameters that allow redefining the `req` and `ack` variables as follows:

Example 3-44 ***SystemVerilog* sequence declaration**

```
sequence req_ack (req, del, ack);
  req ##[1:3] ack; // ack occurs within 1 to 3 cycles after req
endsequence;
```

Sequence operations

SystemVerilog defines a number of operations that can be performed on sequences, such as:

- specifying repetitions
- specifying the occurrence of two parallel sequences
- specifying optional sequence paths (for example, split transactions)
- specifying conditions within a sequence (such as the occurrence of a sequence within another sequence or that a Boolean expression must hold throughout a sequence)
- specifying a first match of possible multiple matches of a sequence
- detecting an endpoint for a sequence
- specifying a conditional sequence through implication
- manipulating data within a sequence

In this section, we focus on a set of common SystemVerilog sequence operators that we use in examples throughout the book. Details for all the SystemVerilog sequence operators are covered in Appendix C, “SystemVerilog Assertions” on page 353.

Repetition operators

SystemVerilog allows the user to specify repetitions when defining sequences of Boolean expressions. The repetition counts can be specified as either a range of constants or a single constant expressions.

Like PSL, SystemVerilog supports three different types of repetition operators, as described in the following section.

expression repeated consecutively

Consecutive repetition. The consecutive repetition operator [*n:m] describes a sequence (or Boolean expression) that is consecutively repeated with one cycle delay between the repetitions. Note that this is exactly like the PSL [*m:n] operator. For example,

```
expr[*2]
```

specifies that `expr` is to be repeated exactly `2` times. This is the same as specifying:

```
expr ##1 expr
```

In addition to specifying a single repeat count for a repetition, SystemVerilog permits specifying a range of possibilities for a repetition.

rules for repeat counts

SystemVerilog repeat count rules are summarized as follows:

- Each repeat count specifies a *minimum* and *maximum* number of occurrences. For example, [*n:m], where n is the minimum, `m` is the maximum and `n <= m`.
- `The` repeat `count` [*n] is the same as [*n:n].
- Sequences as a whole cannot be empty.
- If `n` is 0, then there must be either a prefix, or a post fix term within the sequence specification.
- The keyword **$** can be used as a maximum value within a repeat count to indicate the end of simulation. For formal verification tools, **$** is interpreted as infinity (for example, **[*1:$]** describes a repetition of one to infinity). Note that this is similar to the PSL 1.0 inf keyword.

expression repeated possibly non-consecutively

Nonconsecutive count repetitions. The nonconsecutive count repetition operator [=n:m] describes a sequence where one or more cycle delays are possible between the repetitions. The resulting sequence may precede beyond the last Boolean expression occurrence in the repetition. Note that this is exactly like the PSL [=m:n] operator. For example,

```
a ##1 b[=1] ##1 c
```

is equivalent to the sequence:

```
a ##1 !b [*0:$] ##1 b ##1 !b [*0:$] ##1 c
```

In other words, there can be any number of cycles between a and c as long as there is one b. In addition, there can be any number of cycles between a and the occurrence of b, and any number of cycles between b and the occurrence of c (that is, b is not required to proceed c by exactly one cycle).

Note, the same sequence in PSL 1.0 would be coded as:

```
{a; b[=1]; c}
```

Nonconsecutive exact repetitions. The nonconsecutive exact repetition operator [->n:m] (also known as the *goto repetition* operator) describes a sequence where a Boolean expression is repeated with one or more cycle delays between the repetitions and the resulting sequence terminates at the last Boolean expression occurrence in the repetition. Note that this is exactly like the PSL 1.0 [->m:n] goto operator. For example,

```
a ##1 b[->1] ##1 c
```

expression repeated with one or more cycle delays between repetition while evaluating true on the last cycle

is equivalent to the sequence:

```
a ##1 !b [*0:$] ##1 b ##1 c
```

In other words, there can be any number of cycles between a and c as long as there is one b. In addition, b is required to precede c by exactly one cycle.

Note, the same sequence in PSL 1.0 would be coded as:

```
{a; b[->1]; c}
```

First match operator

The SystemVerilog first_match operator matches only the first occurrence of possibly multiple occurrences of a sequence expression. This allows you to discard all subsequent matches from consideration.

The syntax for the SystemVerilog first match operator is described as follows:

Syntax 3-3 ***SystemVerilog* first match operator**

```
// See Appendix C for additional details

sequence_expr ::=
   first_match ( sequence_expr )
```

Consider an example with a variable delay specification as shown in Example 3-45.

Example 3-45 ***SystemVerilog* first match for req ack sequence**

```
sequence seq_1;
  req ##[2:4]ack;
endsequence

sequence seq_2;
  first_match(req ##[2:4]ack);
endsequence
```

Each attempt of sequence `seq_1` can result in matches for up to four following sequences:

```
req ##2 ack
req ##3 ack
req ##4 ack
```

However, sequence `seq_2` can result in a match for only one of the above four sequences. Whichever of the above three sequences matches first becomes the result of sequence `seq_2`. Notice that this is useful if the `ack` signal is held high for multiple cycles. The first_match prevents multiple unwanted matches from occurring.

Throughout operators

SystemVerilog provides a means for specifying that a specific Boolean condition (that is, an invariant) must hold throughout a sequence using the following construct:

Syntax 3-4 ***SystemVerilog* throughout operator**

```
// See Appendix C for additional details

sequence_expr::=
   expression_or_dist throughout sequence_expr
```

For example, to specify sequence such that an `interrupt` must not occur during an `req-ack-gnt` transaction, we would code the following:

Example 3-46 ***SystemVerilog* sequence with Boolean condition**

```
!interrupt throughout (req ##[2:4] ack #[1:2] gnt)
```

Dynamic variables within sequences

SystemVerilog *dynamic variables* are local variables with respect to a sampling point within a sequence. The advantage of dynamic variables (over global variables) is that each time the sequence is entered, a new local variable is dynamically created. This ensures the sampling of data in overlapping sequence is correctly related to the appropriate sequence evaluation.

In Example 3-47 we demonstrate the usefulness of dynamic variable when validating the correct input/output data relationship in a pipeline register of depth sixteen.

Example 3-47 ***SystemVerilog* dynamic variable to validate pipeline latency**

```
// pipeline regster of depth 16
sequence pipe_operation;
  int x;
  write_en, (x = data_ in)) |-> ##16 (data_out == x) ;
endsequence
```

Restriction on dynamic variable usage, as well as syntax details, are defined in Appendix C.

3.5.2.4 SystemVerilog implication operators

The SystemVerilog implication operator supports sequence implication using the following constructs:

Syntax 3-5 ***SystemVerilog* implication operators**

```
// See Appendix C for additional details

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

SystemVerilog provides two forms of implication: *overlapped* using operator |->, and *non-overlapped using* operator |=>. The *overlapped implication operator* |-> is similar to the PSL *suffix implication operator* |-> which can be read as: *If the left hand side prerequisite sequence holds, then the right hand side sequence must hold.* Likewise, the *non-overlapped implication operator* |=> is similar to the PSL *suffix next implication operator*

|=>, which takes us forward in time by a single clock. For example, the non-overlapped implication operator:

```
(a |=> b)
```

is the same as the overlapped implication operator with a unary delay of one:

```
(a |-> ##1 b).
```

The following points should be noted for sequential implication.

- If the *antecedent sequence* (left hand operand) does not succeed, implication succeeds vacuously by returning true.
- For each successful match of the antecedent sequence, the *consequence sequence* (right hand operand) is separately evaluated, beginning at the end point of the matched *antecedent sequence.*
- All matches of *antecedent sequence* require a match of the *consequence sequence.*

3.5.3 System functions

SystemVerilog provides a number of new system functions useful when defining assertions, which include:

- **$past** (*bit_vector_expr* [, *number_of_ticks*], *clock_enable, clock*) returns a previous value of the *bit_vector_expr.* The *number_of_ticks* argument specifies the number clock ticks used to retrieve the previous value of *bit_vector_expr.* If *number_of_ticks* is not specified, then it defaults to one. If the *clock_enable* is specified, the clock tick is counted when the expression is true. If the *clock* is specified it is the clock for the evaluation. If not specified, the clock from the context of the expression is used.
- **$isunknown** (<expression>) returns true if any bit of the expression is 'x' or 'z'. This is equivalent to

  ^<expression>===’bx.

- **$countones** (<expression>) returns a count that represents the number of bits in the expression set to one. The 'x' and 'z' value of a bit is not counted towards the number of ones.

See Appendix C for additional details related to SystemVerilog.

3.6 PCI property specification example

In this section, our goal is to demonstrate a process of translating a set of natural language requirements into a set of properties. We have chosen examples from the Peripheral Component Interconnect (PCI) specification [PCI-2.2 1998]. Please note that it is not our intention to fully specify all functional requirements of the PCI—we leave this as an exercise for the reader.

You will note that many of the properties we specify in this section are at a transaction-level. Protocol specification and verification at a transaction level is more efficient than at a signal interaction level. Transaction level specification not only permits more efficient test stimulus generation—it also enables debugging and measuring functional coverage at a higher level of abstraction. Nonetheless, specifying transaction level properties is generally not efficient for formal verification (see section 2.5.2 "Formal methodology" on page 48).

Transactions are conveniently constructed by partitioning the behavior definition into a set of sequence specifications, with each sequence representing a specific behavior segment of a transaction. These sequences are then combined to form a more complex bus transaction specification. We recommend that interface protocol or transaction specification occur prior to coding the RTL, at the *specify* or *design/architect* phases described in section 1.4 "Phases of the design process" on page 14.

3.6.1 PCI overview

The PCI local bus is an industry standard, high performance 32- or 64-bit local bus architecture with multiplexed address and data lines. The bus was defined with the primary goal of establishing an industry standard high performance, low cost interconnect mechanism between highly integrated peripheral controller components, peripheral add-in boards, and processor/memory systems.

We begin our discussion of creating a PCI formal specification by illustrating the bus interface required pin list as shown in Figure 3-4. This is followed by a brief description for each required PCI signal. Finally, we demonstrate how to convert a natural language specification of the PCI bus protocol into a set of assertions.

Figure 3-4 **PCI 2.2 Required Pin List**

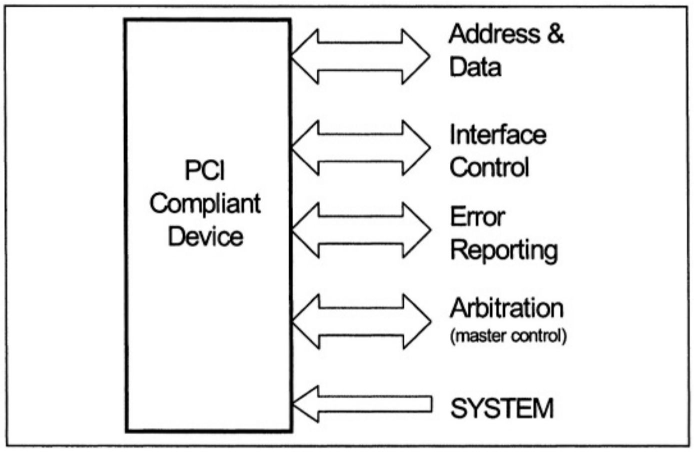


Table 3-1 **Address & Data**

Pin	Description	Direction
AD[31:0] *(ad)*	Address and Data are multiplexed on the same PCI pins.	*bi-directional*
C/BE[3:0 *(cbe_n)*	Bus Command and Byte Enables are multiplexed on the same PCI pins.	*bi-directional*
PAR *(par)*	Parity is even parity across AD[31:0] and C/BE[3:0]#.	*bi-directional*

Table 3-2 **Interface Control**

Pin	Description	Direction
FRAME# *(frame_n)*	FRAME# is activated by the current master to indicate the beginning and duration of a transaction. When FRAME# is deasserted, the transaction is in the final data phase or has completed.	*bi-directional*
IRDY# *(irdy_n)*	Initiator Ready indicates the bus master's ability to complete the current data phase of the transaction. IRDY# is used in conjunction with TRDY#. Note that a data phase is completed on any clock when both IRDY# and TRDY# are asserted.	*bi-directional*
TRDY# *(trdy_n)*	Target Ready indicates the target-selected device's ability to complete the current data phase of the transaction.	*bi-directional*
STOP# *(stop_n)*	STOP# indicates that the current target is requesting the master to stop the current transaction.	*bi-directional*

Pin	Description	Direction
IDSEL *(idsel)*	Initialization Device Select is used as a chip select during configuration read and write transactions.	*input*
DEVSEL# *(devsel_n)*	Device Select, when actively driven, indicates the driving device has decoded its address as the target of the current access.	*bi-directional*

Table 3-3 **Error Reporting**

Pin	Description	Direction
PERR# *(perr_n)*	Parity Error is only for reporting data parity errors during all PCI transactions (except a Special Cycle not discussed here).	*bi-directional*
SERR# *(serr_n)*	System Error is for reporting address parity errors, data parity errors on the Special Cycle command, or any other system error where the result will be catastrophic.	*bi-directional*

Table 3-4 **Arbitration**

Pin	Description	Direction
REQ# *(req_n)*	Request indicates to the arbiter that this agent desires use of the bus.	*output*
GNT# *(gnt_n)*	Grant indicates to the agent that access to the bus has been granted.	*input*

Table 3-5 **System**

Pin	Description	Direction
CLK *(clk)*	Clock provides timing for all transactions on PCI and is an input to every PCI device. All other PCI signals, except RST#, INTA#, INTB#, INTC#, and INTD#, are sampled on the rising edge of CLK and all other timing parameters are defined with respect to this edge.	*input*
RST# *(req_n)*	Reset is used to bring PCI-specific registers, sequencers, and signals to a consistent state.	*input*

A PCI bus transaction consists of an *address phase* followed by one or more *data phases.* During the address phase, the `C/BE[3:0]#` bus command indicates the type of transaction. During the data phase, `C/BE[3:0]#` are used as Byte Enables.

Note that the # symbol at the end of the signal name indicates an active low signal. For our examples, we convert the # symbol into "`_n`" as part of the name to indicate an active low signal.

3.6.1.1 PCI master reset requirement

In this section, we demonstrate how to translate a simple PCI reset requirement, stated in section 2.2.1 (page 9) of the PCI Local Bus Specification [PCI-2.2 1998]. The PCI reset requirement is stated as follows:

> *To prevent AD, C/BE#, and PAR signals from floating during reset, the central resource may drive the RST# line during reset (bus parking) but only to a logic low level; they may not be driven high.*

This is an example of a *conditional expression pattern,* described in section 6.4.1 on page 179. In Example 3-48, we have written a PSL assertion to check that the `AD`, `C/BE#`, and `PAR` signals are never driven high during reset.

Example 3-48 *PSL* master reset assertion

```
assert always ((rst_n==0) -> !(|{ad, cbe_n, par})) @ (posedge clk);
```

Note that we have used the Verilog *reduction or* operator to determine if any bit in this example is a logical one. The same assertion could be specified using a Verilog OVL implication monitor as shown in Example 3-49.

Example 3-49 *OVL* master reset assertion

```
assert_always master_reset (clk, !rst_n, !(|{ad, cbe_n, par});
```

3.6.1.2 PCI burst order encoding requirement

The memory address space for the PCI is defined by the bits `AD[31:2]`. the lower two bits (that is, `AD[1:0]`) are encoded to indicate the order in which the master is requesting the data transfer, as defined in section 3.2.2.2 (page 29) of the PCI Local Bus Specification. Table 3-6 specifies the legal burst order encoding for memory transactions. Hence, address bit `AD[0]` must never be set to an active high value for a memory transaction burst order request.

Table 3-6 **Burst Order Encoding**

AD[1]	AD[0]	Burst Order
0	0	Linear Increment
0	1	Reserved
1	0	Cache Wrap Mode
1	1	Reserved

Example 3-50 ***PSL*** **PCI legal memory- transaction burst order encoding**

```
'define mem_cmd ((cbe_n ==  'MEM_READ) || \
                 (cbe_n ==  'MEM_WRITE) || \
                 (cbe_n ==  'MEM_RD_MULTIP) || \
                 (cbd_n ==  'MEM_RD_LINE) || \
                 (cbd_n ==  'MEM_WR_AND_INV))

sequence SERE_MEM_ADDR_PHASE = (frame_n; !frame_n && mem_cmd);

property PCI_VALID_MEM_BURST_ENCODING =
  always ({SERE_MEM_ADDR_PHASE} |-> (!ad[0]}) @ (posedge clk)
    abort !rst_ n ;

assert PCI_VALID_MEM_BURST_ENCODING;
```

In Example 3-50, we code a PSL assertion to validate a correct memory burst order request. Note that this assertion uses a sequence to define a memory address phase sequence (that is, a falling edge of `FRAME#`, along with the decoding of a memory transaction from the bus command `C/BE#`). Whenever this prefix sequence occurs, then bit `AD[0]` must always be active low for a valid burst order encoding.

Note that Example 3-50 is an example of a *sequence implication pattern,* as described in section 6.4.2 on page 181. The PCI memory address phase is described by defining the sequence SERE_MEM_ADDR_PHASE, which matches sequences containing a falling edge of `FRAME#` combined with a decoding of a memory command. This forms a prefix sequence, which implies that the reserved `AD[0]` is not active high. For additional details on sequence and the suffix implication operator |->, see Appendix B.

3.6.1.3 PCI basic read transaction

In this section, we demonstrate (via an simplified example) another transaction-level property, which we construct by partitioning the transaction into a set of partial behaviors specified as sequences. A PCI basic read operation consists of the following phases:

- an *address phase,* which for a basic read consists of a single address transfer in one clock
- a *data phase,* which includes one transfer state plus zero or more wait states

The address phase occurs on the first clock cycle in which `FRAME#` is asserted. For a basic read transaction, there must be at least one turn around cycle between the address phase and the data phase. A data phase completes when an active `IRDY#` and either an active `TRDY#` or `STOP#` is clocked. The read transaction completes when FRAME# becomes inactive. In reality, there are numerous transaction terminating conditions defined in section 3.3.3 of the PCI specification that can be initiated by either the master or target (for example, *timeout, abort, retry, disconnect*). For our PCI basic read operation, our goal is to demonstrate how to build a transaction through a set of sequence specifications. Hence, we have chosen to simplify our example and ignore these special terminating cases. We leave it to the reader to modify our example by specifying all terminating conditions.

byte enable requirement

Section 3.3.1 (page 47) of the PCI Local Bus Specification states the following requirement associated with a read transaction:

> *The C/BE# output buffers must remain enabled (for both read and writes) for the first clock of the data phase through the end of the transaction.*

Example 3-51 demonstrates how to specify a PCI basic read transaction as specified with the C/BE# output buffer requirement. The property PCI_READ_TRANSACTION begins with an address phase (that is, SERE_RD_ADDR_PHASE). We then specify a sequence that describe the initial required turn around cycle (that is, SERE_TURN_AROUND), which occurs the first clock after the address phase. Then, the C/BE# signals remain unchanged throughout the remaining data phase (cbe_n==prev(cbe_n) throughout SERE_DATA_PHASE.

When specifying protocol requirements, you have the choice of creating a complex property that captures all requirements required for the transaction—or partitioning the different requirements of the transaction into a set of simpler properties. For example, for simplicity we decided not to specify the read transaction latency requirements in Example 3-51 for either the bus target or master (as defined in section 3.5 of the PCI specification). Hence, you could either modify our assertion example by directly writing in the additional bus latency requirements, or you could create a separate simpler property for the latency requirements.

Example 3-51 *PSL* PCI basic read transaction

```
`define data_complete ((!trdy_n || !stop_n) && !irdy_n && !devsel_n)
`define end_of_transaction (data_complete && frame_n)
`define adr_turn_around (trdy_n & !irdy_n)
`define data_tranfer (!trdy_n && !irdy_n && !devsel_n && !frame_n)
`define wait_state ((trdy_n || irdy_n) && !devsel_n)
`define cbe_stable (cbe_n==prev(cbe_n))
`define read_cmd ((cbe_n == `IO_READ) || \
                  (cbe_n == `MEM_READ) || \
                  (cbe_n == `CONFIG_RD) || \
                  (cbe_n == `MEM_RD_MULTIP) || \
                  (cbe_n == `MEM_RD_LINE))

sequence SERE_RD_ADDR_PHASE = {frame_n; !frame_n && read_cmd};
sequence SERE_TURN_AROUND = {adr_turn_around};
sequence SERE_DATA_TRANSFER = {{wait_state [*] ;data_transfer} [1 :inf]}
sequence SERE_END_OF_TRANSFER = (data_complete && frame_n};
sequence SERE_DATA_PHASE =
  {
    {{SERE_DATA_TRANSFER};{SERE_END_OF_TRANSFER}} && {cbe_stable}
  };

property PCI_READ_TRANSACTION =
  always ({SERE_RD_ADDR_PHASE) |=>
           {SERE_TURN_AROUND; SERE_DATA_PHASE}) @ (posedge clk)
             abort !rst_n ;

assert PCI_ READ_TRANSACTION;
```

Note for this example we are using the PSL *sequence length-matching AND operator* (&&). Hence, this enables us to check throughout the data phase that `C/BE#` is stable throughout the *data transfer* and *end of transfer* sequence.

3.7 Summary

In this chapter, we introduced general concepts related to property specification. We then applied these concepts as we introduced emerging specification standards, which included the Accellera PSL property specification language proposal [Accellera PSL-1.1 2004], the Open Verification Library [Accellera OVL 2003], and SystemVerilog 3.1a assertion constructs [Accellera SystemVerilog-3.1a 2004]. Each of the assertion standards we discuss has its own merits. Our objective is to help the engineer understand the advantages (and limitations) of the various assertion forms and their usage model. This will prepare readers to select appropriate specification forms that suit their needs (or preferences). Finally, we demonstrated a process of translating a set of natural language requirements for the Peripheral Component Interconnect (PCI) specification into a set of properties.