CHAPTER 8

SPECIFYING CORRECT BEHAVIOR

In this chapter, we present a set of tips that we have found effective when attempting to specify various aspects of a design at multiple levels of abstraction. We first present a set of common ambiguities that arise when interpreting a natural language specification. We then discuss the limitations of temporal property languages and the need for additional modeling to overcome these limitations when attempting to specify higher-level requirements. We conclude by presenting a construction guide we have found useful in our own work with assertion and functional coverage specification.

8.1 Natural language interpretation

Historically, the process of specification has consisted of creating a natural language description of a set of design requirements. This form of specification is both ambiguous and, in many cases, unverifiable because of the lack of a standard machine-executable representation. Furthermore, ensuring that all functional aspects of the specification have been adequately verified (that is, covered) is problematic. A formal specification overcomes these problems. Yet, we must take care when interpreting the natural language specification to create a formal specification.

In this section, we examine various natural language phases and ambiguities associated with these common phases. While a natural language specification allows multiple interpretations, a formal specification is precise. One comment often heard by engineers who begin writing formal specification is:

- *I didn't mean that type of behavior when I specified the property,* or
- *You mean I have to explicitly tell it not to do that?*

Thus, we must carefully consider intent when creating a formal specification to ensure that it explicitly captures the behavior that we want to specify while explicitly forbidding behavior that is not permitted.

8.1.1 Temporal ambiguity

Ambiguity of next. In this section, we examine the ambiguity of the word *next* in a natural language interpretation. There are two possible specification interpretations of the word next: the next cycle (immediate) and some future cycle (eventually).

immediate next

Interpreting an immediate next. Consider the English specification shown in Example 8-1.

Example 8-1 English: Specification for immediate next event

If a snoop hits a modified line in the L1 cache, then the next *transaction must be a snoop writeback.*

The word *next* could be interpreted in multiple ways. For instance, in Example 8-2 the creator of the PSL formal specification interpreted the English specification to mean that the next clock after a snoop would be a writeback.

Example 8-2 PSL: Specification for immediate next event

```
default clock = (posedge clk);
property CacheSnoopWriteback =
     always ({snoop && hit_modified} |=> {writeback});
assert (CacheSnoopWriteback ;
```

Example 8-3 demonstrates how to code the same natural language specification described in Example 8-1 using SystemVerilog.

Example 8-3 SystemVerilog; Specification for immediate next event

```
property CacheSnoopWriteback;
     always @ (posedge clk) (snoop && hit_modified|=> writeback);
endproperty
assert property (CacheSnoopWriteback);
```

Note that if the *next* transaction does not coincide with the *next* clock (that is, the next valid transaction might occur multiple clocks after a snoop), then the formal specification must be modified to reflect this case.

eventual next

Interpreting an eventual next. Example 8-4 illustrates another specification associated with the word *next,* which is less ambiguous than the previous example. For this example it is probably safe to assume that there can be inactive cycles where packets are not sent. The English specification is not clear on this point.

Example 8-4 English: Specification for eventual next event

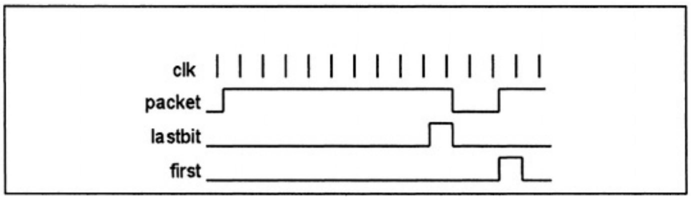
If a data "packet" of any size starts and eventually gets a "lastbit" , then the next *data "packet" must have the "first" bit asserted.*

Example 8-5 demonstrates how to code this property using PSL, assuming the possibility of inactive packet cycles. Notice that using `rose(packet)` prevents launching (that is, starting) multiple assertions at every clock edge where the packet is asserted high.

Example 8-5 PSL: specification for eventual next event

```
default clock = (posedge clk);

property PacketLastFirst =
    always ({rose(packet);lastbit [->1]} |=>
                                {rose(packet) [->1] : first});
assert PacketLastFirst;
```

Figure 8-1 Waveforms for logic implementing Example 8-4



Example 8-6 demonstrates how to write a SystemVerilog property for this specification.

Example 8-6 SystemVerilog: Specification for eventual next event

```
property PacketLastFirst;
    @(posedge clk) $rose(packet);lastbit [->1] |=>
        $rose (packet)[->1] ##0 first;
endproperty
assert property (PacketLastFirst);
```

Ambiguity of after. In this section, we examine the ambiguity of the word *after* in a natural language interpretation. The ambiguity of the word *after* has two possible interpretations: next cycle (immediate) and some future cycle (eventually).

immediate next

Interpreting an immediate after. Consider the English specification shown in Example 8-7.

Example 8-7 English: Specification for after an event

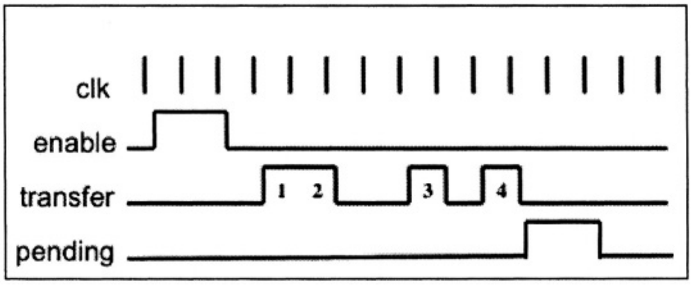
```
If signal "enable" rises, then after the fourth transfer signal
"pending" must rise.
```

In Example 8-8, the creator of the PSL formal specification interpreted the English speciation to mean that the `pending` signal would be active the clock immediately after the fourth `transfer`. Note that it is coded with the PSL goto repetition operator. This 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, as shown in Figure 8-2.

Example 8-8 PSL: Specification for immediately after an event

```
default clock = (posedge clk);

property PendingImmediatelyAfter =
     always ( {rose (enable) } |-> {transfer [->4] ; rose(pending)});
assert  PendingImmediateAfter;
```

Figure 8-2 Pending occurs immediately after fourth transfer



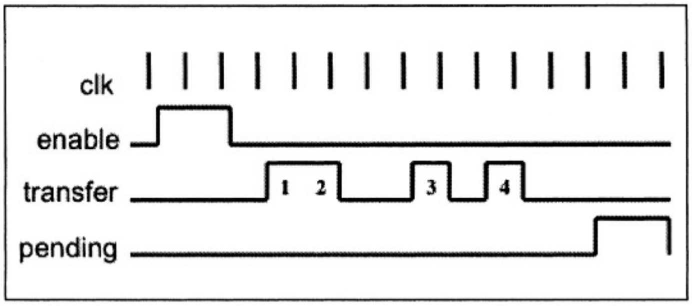
eventual next

Interpreting an eventual after. An alternative way that the word *after* could be interpreted is demonstrated in Example 8-9. The creator of the PSL formal specification interpreted the English specification to mean that the `pending` signal would eventually be active on some future clock after the fourth `transfer`. The nonconsecutive count repetition operator [=4] describes a sequence where one or more inactive cycle delays are possible between the four repetitions of transfer. Note that the resulting sequence may continue beyond the occurrence of the last Boolean expression in the repetition with additional inactive cycle delays, as shown in Figure 8-3.

Example 8-9 PSL: Specification for eventually after an event

```
default clock = (posedge clk);

property PendingEventuallyAfter =
     always ({rose (enable)} |-> {transfer [=4] ; rose(pending)}) ;
assert PendingEventuallyAfter;
```

Figure 8-3 Pending occurs eventually after the fourth transfer



Example 8-10 demonstrates how to code the same two previous examples using SystemVerilog assertions.

Example 8-10 SystemVerilog: immediately and eventually after

```
property PendingImmediatelyAfter;
  @(posedge clk)
    $rose (enable)  |-> transfer [->4]  ##1 $rose (pending);
endproperty
assert property (PendingImmediateAfter);

property PendingEventuallyAfter;
  @ (posedge clk)
     ($rose (enable)  | -> transfer[=4]  ##1 $rose (pending));
endproperty
assert property (PendingEventuallyAfter);
```

8.1.2 Active ambiguity

In this section, we examine the ambiguity of the word *active* in a natural language interpretation. Consider the English specification shown in Example 8-11.

Example 8-11 English: Specification for active events

If signal "hit" is active *and signal "pending" is not* active, *then the next time "pending" is* active, *signal "sel5" is* active.

The question that arises is: What does the word *active* mean? In other words, are we referring to edge-sensitive or level-sensitive events? For instance, in Example 8-12 we are specifying an active event for the case when `hit` rises and `pending` is low.

Example 8-12 PSL: Specification for edge-sensitive events

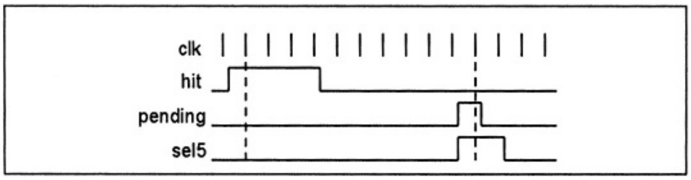
```
default clock = (posedge clk);

property HitPendingSel5 =
    always ({rose(hit) && !pending; pending[->1]} |-> {sel5});
assert HitPendingSel5;
```

Note that if we did not specify `rose(hit)`, then multiple properties would begin evaluation at every clock edge in which `hit` is high and `pending` is low. This could result in performance issues during verification or a false failure.

Also note the use of the goto repetition operator (`[->1]`), which allows us to describe a sequence where one or more inactive cycle delays are possible before an active `pending`. However, the sequence described by this repetition must end on the first occurrence of pending (that is, the sequence does not extend beyond an active pending).

Figure 8-4 illustrates the case when the active event is the rising occurrence of `hit` and `pending` is low. Even though `hit` remains high for four clock cycles, only a single evaluation of the assertion occurs (as opposed to starting a new evaluation at every clock cycle in which `hit` is high). This form of specification is more efficient for verification.

Figure 8-4 Edge-sensitive sequence implication



Note, however, there are situations where multiple independent evaluations of the assertion are required. Example 7-12 on page 221 illustrates this case. For these cases, a level active form of specification is required. Example 8-13 demonstrates a SystemVerilog property for this specification.

Example 8-13 SystemVerilog: Specification for edge-sensitive events

```
property  HitPendingSel5;
   @(posedge clk)
   $rose(hit) &&  !pending  ##1  pending[->1]  |->  sel5;
endproperty
assert property (HitPendingSel5);
```

8.1.3 Boundary ambiguity

In this section, we examine the ambiguity of the word *between* in a natural language interpretation. Consider the English specification shown in Example 8-14.

Example 8-14 English: Specification for between events

Between *a request and its acknowledge the busy signal must remain asserted.*

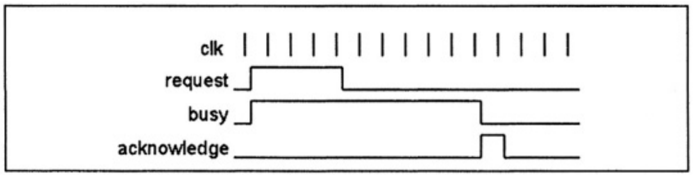
The ambiguity of this natural language form of specification arises from the boundary conditions. Are we explicitly including or excluding the overlap between the bounding events? For example, it is unclear whether the `busy` signal is supposed to be asserted at the same time as the `request` or `acknowledge`. Furthermore, the specification does not seem to require that the `request` be held active until the occurrence of an `acknowledge`.

Example 8-15 demonstrates a PSL specification that does not require `busy` to occur when `acknowledge` is asserted, and Figure 8-5 illustrates the waveform for this behavior.

Example 8-15 PSL: Specification for nonoverlapping between event

```
default clock = (posedge clk);

property ReqAckBusy =
      always ({rose(request)} |-> {busy[1:*]; acknowledge});
assert ReqAckBusy;
```

Figure 8-5 Busy between request and acknowledge, not required to overlap

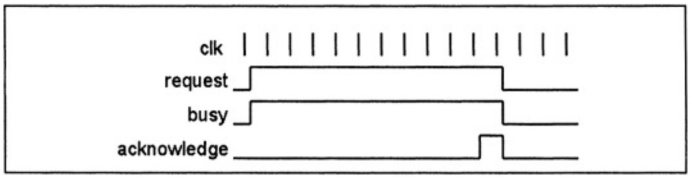


Example 8-16 demonstrates a PSL specification that requires both the request and the `busy` to occur when `acknowledge` is asserted. The waveform for this behavior is illustrated in Figure 8-6.

Example 8-16 PSL: Specification for overlapping between event

```
default clock = (posedge clk);

property ReqAckBusy =
     always ({rose(request)} |->
                        {(request & busy & !acknowledge) [1: *];
                           request & busy & acknowledge});
assert ReqAckBusy;
```

Figure 8-6 Request and busy asserted up to and including acknowledge



Example 8-17 demonstrates how to code the same example in SystemVerilog.

Example 8-17 SystemVerilog: Specification for between event

```
property ReqAckBusy;
  @(posedge clk)
  $rose(request)} |->
       (request & busy)[1:$] intersect acknowledge [->1];
endproperty
assert property(ReqAckBusy);
```

8.1.4 Too strong interpretation

In this section, we examine the problem of interpreting natural language specification too strongly. Consider the English specification shown in Example 8-18.

Example 8-18 English: Specification for even and odd write addresses

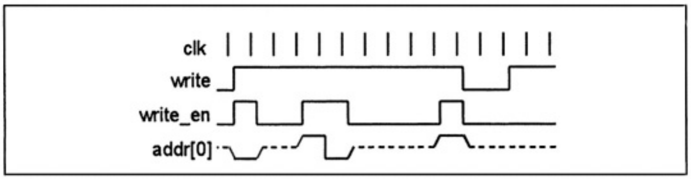
```
For every write, data transfers must alternate between even and odd
entries. In other words, if there is a write, then as long as we are
transferring data belonging to this write, consecutive data transfers
must alternate between even and odd addresses.
```

Example 8-19 illustrates one interpretation for the specification described in Example 8-18.

Example 8-19 PSL: Specification for even and odd write addresses

```
default clock = (posedge clk);

property EvenOddAddr =
   always ({rose(write)} |->
             {{transfer[->1] : !addr[0];
               transfer[->1] : addr[0] }[*];
              fell(write)
             });
assert EvenOddAddr;
```

Figure 8-7 Pair-wise even and odd memory writes



Note that this specification might have been interpreted too strongly. For example, the interpreter implicitly assumes that all writes will start on an even address. In addition, it only allows pair-wise writes. If this is not the case, then we must relax the formal specification.

Example 8-20 demonstrates how to write the same assertion using SystemVerilog.

Example 8-20 SystemVerilog: Specification for even and odd write addresses

```
property EvenOddAddr =
   @ (posedge clk)
    $rose(write) |->
        (transfer[->1] ##0 !addr[0]
     ##1 transfer[->1] ##0  addr[0])[*]
     ##1 $fell(write);
endproperty
assert property (EvenOddAddr);
```

8.1.5 Implicit assumption

In this section, we examine the problem of making assumptions when interpreting a natural language specification. Consider the English specification shown in Example 8-21.

Example 8-21 English: Specification for two consecutive writes

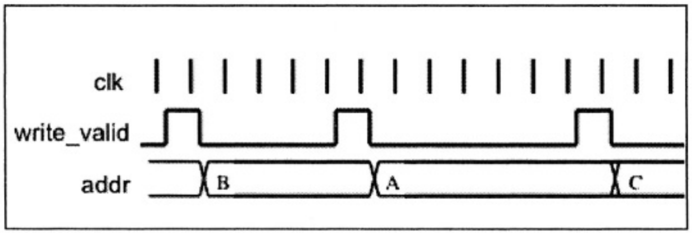
```
Two consecutive writes cannot be to the same address. Address appears
one cycle after write_valid.
```

Example 8-22 demonstrates a case in which the engineer assumed that the address would be held constant between active `write_valid` signals, as demonstrated in Figure 8-8.

Example 8-22 PSL: Specification for two consectutive writes

```
property DifferntAddr =
  always (write_valid -> next (addr != prev(addr)));
assert DifferentAddr;
```

Figure 8-8 Address is held valid between active write valids

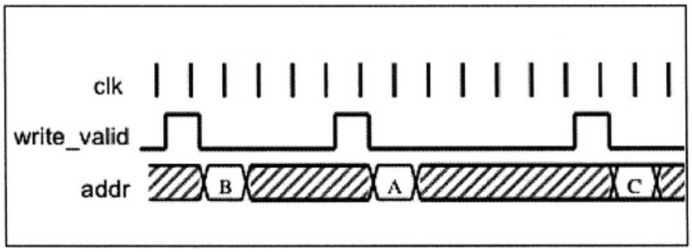


In Example 8-23, as an alternative, the engineer only assumed that the address would be valid exactly one clock after an active high `address_valid` signal, as demonstrated in Figure 8-9. No assumption was made about address values at any other point in time.

Example 8-23 SystemVeriliog: Specification for two consecutive writes

```
property DifferntAddr;
  reg [15:0] last_addr;
  @(posedge clk) write_valid, last_addr=addr
    |=> write_valid[->1] ##0 addr != last_addr);
endproperty
assert property (DifferentAddr);
```

Figure 8-9 Address only valid for one clock cycle after a write valid



Which interpretation is correct? Actually, either would work. Or due to future changes in implementation, one may fail. The point is that clarification of the English specification is often required to truly obtain the design intent.

8.1.6 Partial specification

In this section, we examine the issues with partially interpreting a natural language specification. Consider the English specification shown in Example 8-24.

Example 8-24 English: Specification for last read related to write.

The data that returns for a read is the last data that was written to the register before the read was issued.

In Example 8-25, the engineer created a SystemVerilog specification using a local variable `last_data` to capture the last data value written when an active `write` command occurs. During the next read, the captured data is compared against the value on the data_out port.

Example 8-25 SystemVeriliog: Specification for last read related to write.

```
property LastReadRelatedToWrite;
  reg [15:0] last_data;
  @(posedge clk)
        $rose(write), last_data=data_in
     ##1 (!write throughout read [->1]} |->
        data_out==last_data);
endproperty
assert property (LastReadRelatedToWrite);
```

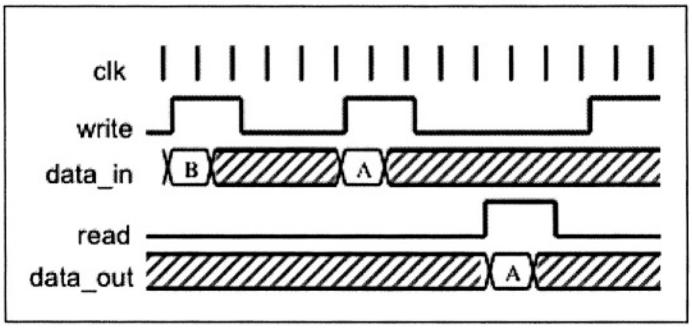
Notice that we are not specifying that we can read the value of *any* written data—just the very last data written. Hence, the following sequence limits the specified behavior to only sequences of a last write before a read:

- An active rising write, followed by

- No more writes, followed by
- An active rising read

All other occurrences of an active rising write followed by another active rising write will be ignored (that is, not checked). We use this sequence as an antecedent, which is combined with the SystemVerilog overlapping implication operator (|->), to only check the read data value for the last data written as shown in Figure 8-10.

Figure 8-10 Read data value same as last data written



As we previously stated in Chapter 2, often it is easier to partition complex behavior into a set of discrete behaviors when formally specifying properties. For example, it would have been too complicated to fully describe the bus write and read behavior on the previous example. Even though the last specification is only a partial specification of the design, it is certainly easier to capture than trying to globally describe all possible write and read combinations in a single property. Nonetheless, there is also a danger of contradictions when writing large amounts of partial specification. Hence, you must carefully consider the impact of each property specification on the others in a set.

8.2 Property specification guidelines

The process of misinterpreting a natural language specification during RTL design can also occur during formal property specification. Furthermore, engineers often misinterpret the defined syntax and correct use of formal property languages and often introduce errors in the specification. The following is a list of guidelines we have found useful when coding formal properties. And when followed consistently, it helps the engineer avoid errors during the specification process:

- Cover *sequence*—do not cover properties
- Assert *non-negated* implication—do not assert a negated implication
- Assert *negated* forbidden sequences—do not assert a non-negated sequence
- Assume Boolean properties or implication properties—do not assume a sequence

The process of covering a sequence and asserting a negated forbidden sequence is the act of attempting to match the specified sequence. If the sequence fails to match, no harm is done. However, if the sequence matches, then the cover command reports the occurrence of the sequence match while the assert command identifies a forbidden sequence (that is, a violation).

Non-negated implication assertions should define expected behavior based on a preceding event or sequence. Only when the event or sequence occurs should the resulting expected behavior appear. A common mistake made by engineers when specifying a forbidden sequence is to assert a negated implication. The problem with this approach is when the antecedent fails to match, the overall assertion will fail due to the negation, as demonstrated in Example 8-26.

Example 8-26 PSL: Incorrect use of a negated Implication

```
// Will produce a false error at every clock when req does not occur
assert never (req -> next halt);
```

Note that if `req` is false, then the implication (->) is true. However, since we are asserting that the property would never be true, the assertion fails at every clock that `req` is not true, which is not what we intended. See section 3.3.3 "RTL cycle related assertions" on page 73 for addition details on Boolean implication.

Alternatively, asserting a forbidden sequence enables us to specify the intended behavior as demonstrated in Example 8-27.

Example 8-27 PSL: Correct use of a forbidden sequence

```
// Will procduce correct error whenever a halt occurs after a req
assert never ({req; halt)};
```

Finally, a common mistake made when specifying assumptions is to assume a sequence. The problem with this approach is that at every clock cycle, a new sequence would start prior to the completion of any previous sequence. This is probably not the behavior intended, and can lead to many failures. Hence, we recommend that you only assume Boolean expressions or an implication sequence.

prevents false failures due to repeated implications.

8.2.1 Sequence ambiguity

Specifying a sequence as the antecedent to an implication can result in multiple matches on the antecedent—each of which requires the consequent to match from that point forward. For example, multiple matches may occur when using repetition ranges or delay ranges when specifying a sequence. The problem with multiple matches is that a property may encounter a false failure if you neglect to account for overlapping sequences. We recommend that you use the SystemVerilog **first_match** when only one match of the sequence is required.

Example 8-28 Use of first_match on antecedent sequences

```
property complex_req_start_retry;
  @(posedge clk)
  first_match(req_start ##[1:5] restart ##1 retry)
    |-> abort ##1 !abort throughout req_start[->1]
endproperty
```

If the previous Example 8-28 had been written without the **first_match** operator, the antecedent sequence could match more than once if `restart` occurred more than once in the five cycle window after `req_start`. Any subsequent matches would cause the consequent sequence for the implication to matched again, which might not be possible and could create a false failure.

8.2.2 Syntax ambiguity

The syntax of both PSL and SystemVerilog provide rules that define the precedence of their operators. This allows for common expressions to be written in a straightforward manner. There are other common expressions that require explicit identification (for example, using parenthesis) of the evaluation order. Here are some guidelines we recommend you follow when specifying common expressions in both PSL and SystemVerilog to avoid unexpected results.

PSL always. The implication operations (for example, ->, |->, |=>), and bounding operators (until and before), are lower precedence than the occurrence operators (for example, always, never, next, and eventually). We recommend that you use parenthesis around all expression associated with any occurrence operator to ensure property grouping.

PSL sequences. As a guideline, you should always use braces around each sequence when combining multiple sequences using the various PSL operators.

PSL *fusion* and SystemVerilog *##0*. A common mistake made when specifying that a Boolean expression must evaluate true on the very last cycle of a sequence is to and the Boolean expression with the sequence, as shown in Example 8-29.

Example 8-29 PSL: Incorrectly specifying overlapping expressions

```
// The following error was an attempt to specify that done will occur
// whenever transfer occurs
default clock = (posedge clk);

property EvenOddAddr =
  always ({rose (write)} |-> {{transfer [->1]} & {done}});
assert EvenOddAddr;
```

Example 8-29 demonstrates the correct way to specify the overlapping behavior using the PSL fusion operator.

Example 8-30 PSL: Correctly specifying overlapping expression

```
// The following correct example uses the fusion operator (:) to
// specify that done will overlap on the last occurence of transfer
default clock = (posedge clk) ;

property EvenOddAddr =
  always ({rose (write )} |-> ( (transfer [->1]} : {done}});
assert EvenOddAddr;
```

A similar common coding problem occurs when specifying overlapping expressions using SystemVerilog. For SystemVerilog, you should use the ##0 delay operator instead of an & operator.

8.3 Clarity in higher-level specification

In this section, we discuss the process of property specification at various levels of design abstraction including:

- RTL implementation assertions
- Block-level specification (both protocol and end-to-end)
- System-level global properties

While many lower-level RTL implementation properties and some simple protocols can be specified entirely by using the Boolean and temporal layer of a property language, many higher-level properties often require additional modeling outside of the property language, particularly for block-level end-to-end data integrity properties that involve checking transaction ordering. For example, designs containing any type of queues.

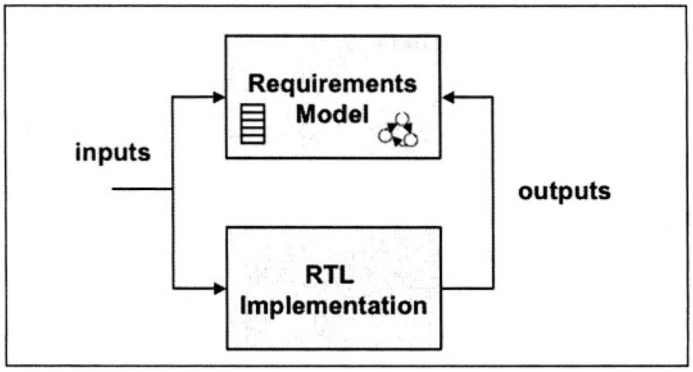
Even if a block interface[1] could be entirely described with a large set of temporal properties (that is, our specification does not involve queuing semantics), this approach suffers from the following problems:

- It is often difficult to partition higher-level behavior into a set of discreet temporal properties (even though this might be desirable)
- It is difficult to establish the completeness of a large set of properties (that is, it is difficult to think of all possible behaviors)
- It is often difficult to understand the original design intent (that is, the high-level behavior) by examining a large set of complex temporal properties

An alternate approach to specifying higher-level behavior is to model the complex behavior as an architectural abstract state-machine (which monitors the appropriate input and output ports of a block implementation) and then apply a smaller set of simpler temporal properties (assertions and constraints) to the abstract FSM.

We refer to this combination of modeling with a small set of temporal properties as a *requirements model* (as illustrated in Figure 8-11).

Figure 8-11 Specifying end-to-end behavior with a requirements model



One key benefit of a requirements model is that design intent is easier to understand from this abstract model than trying to extract the intended behavior from a large set of temporal properties.

1. Note that we do not mean block-level end-to-end properties, we are talking about properties that specify a single interface to a block.

property structure review

Before we discuss various levels of specification, we first review a few fundamental concepts related to specifying design intent that were introduced in Chapter 3, "Specifying RTL Properties". As we previously described, informally a property is a specification of design intent. When discussing properties, it is generally easier to view their composition as distinct layers. That is:

- The *Boolean layer,* which is comprised of Boolean expressions (for example, any valid Verilog Boolean expression)
- The *temporal layer,* which describes the relationship of Boolean expressions over time
- The *modeling layer,* which provides a means to model complex high-level end-to-end requirements—as well as auxiliary logic not easily described by the Boolean and temporal layers (for example, an architectural FSM or data-storage)
- The *verification layer,* which describes how to use a property during verification (for example, a directive that states that the property is to be used as a target or assertion that must be proved by the verification engine—or the property should be used as a constraint or assumption to the verification engine)

8.3.1 Implementation assertions

As this book has demonstrated, standards and techniques for specifying implementation and structural level properties directly within the RTL are emerging. For example (as we have previously demonstrated), while coding an RTL FIFO, it is a good idea to specify that the FIFO must never underflow or overflow. One way to accomplished this objective is by instantiating a Verilog OVL assertion monitor directly into the RTL model as follows:

Example 8-31 OVL: Specification for FIFO overflow and underflow

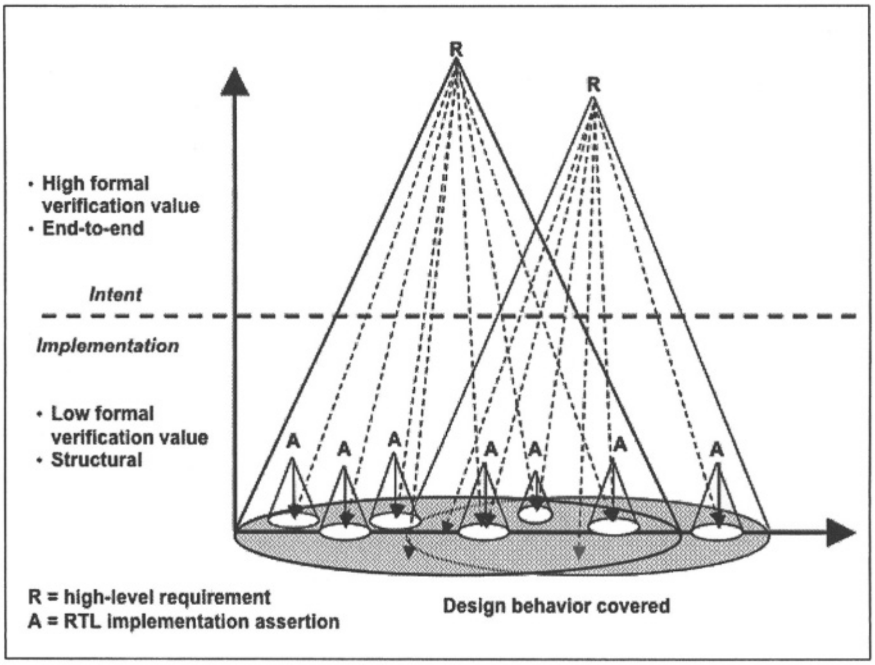
```
// OVL assert that the FIFO cannot overflow
assert_never no_overflow (clk, reset_n,
                ({push,pop}==2'b10 && cnt==`DEPTH-1));

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

If the Boolean expression that represents an underflow or overflow condition ever evaluates to true during the simulation process, the assertion will fire and help isolate the problem. Hence, assertions added to an RTL model can reduce the simulation debug by up to 50%.

implementation assertions rarely need additional modeling

Coding lower-level RTL implementation assertions is generally straight forward and rarely requires additional modeling. Example 6-43 "PSL and Verilog pipelined req/ack handshake protocol" on page 200 provides one example where additional simple modeling was required to help keep track of multiple overlapping req/ack pairs, although some might argue that this example is actually not a lower-level RTL implementation assertion, but more of a somewhat higher-level interface assertion. In general, whenever any additional modeling code is required for implementation assertions, it is usually only a few lines of additional HDL.

Figure 8-12 High-level requirements versus implementation assertions


R
R
• High formal verification value
• End-to-end
Intent
Implementation
• Low formal verification value
• Structural
A
A
A
A
A
A
A
R = high-level requirement
A = RTL implementation assertion
Design behavior covered

proving implementation assertions

One key point worth noting, that concerns specifying lower-level implementation and structural assertions, is that the aggregate of all these lower-level assertions would certainly not be comprehensive and does not represent the design intent from an end-to-end block-level perspective. In fact, we could run into a situation where the RTL does not violate any of the lower-level implementation or structural assertions, yet the block functions incorrectly. This is not to say that adding implementation assertions is bad. On the contrary, implementation assertions tend to force the designer to think about the details related to a particular structure—and in this way, they prevent bugs during the design process. And keep in mind that implementation assertions will significantly reduce debug time during the simulation process. However, it is important to note that while this form of specification is useful for bug-hunting—it is not a useful form of specification for formal verification (that is, verification assurance) in terms of return on investment. Our experience has been that the effort to formally prove hundreds of lower-level properties can often be as much as the effort to prove tens of higher-level properties, yet the amount of behavior they cover is significantly less, as demonstrated in Figure .

8.3.2 Higher-level requirements

The process of specification can be viewed as a spectrum of requirements—ranging from lower-level implementation assertions (previously discussed) up to very high-level architectural properties. This section discusses specification of design intent at a higher level of abstraction, which is referred to as *high-level requirements* [Foster et al. 2004]. Unlike implementation or structural assertions, high-level requirements take a black-box view when specifying a block's intended behavior. Hence, high-level requirements are more comprehensive than implementation assertions in that they specify end-to-end behavior. The following English text demonstrates the end-to-end nature of high-level requirement for a PCI Express data link layer block illustrated in Figure 8-13 [Loh et al. 2004].

Example 8-32 English: Specification for PCI Express data link layer

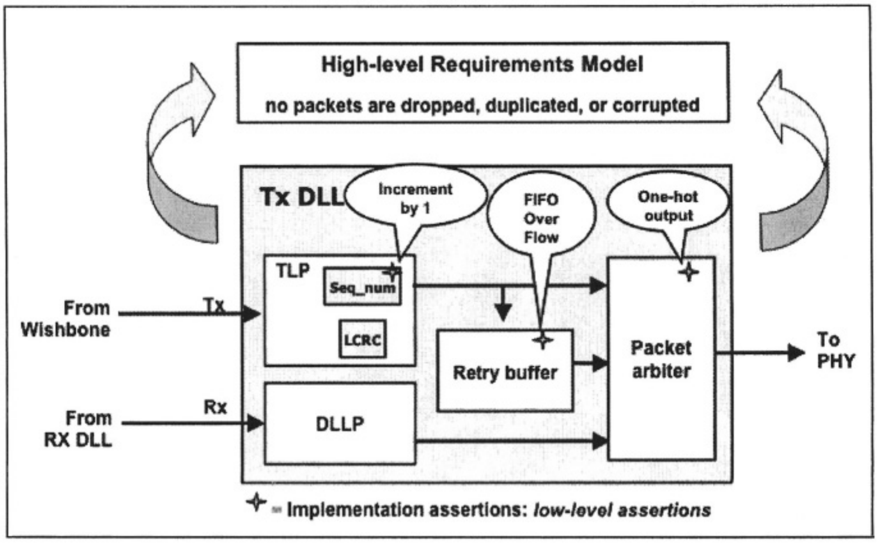
No packet sent to the PCI Express data link layer block (from the wishbone transaction layer interface) may be dropped, duplicated, or corrupted as it exits the block to the physical layer.

This high-level requirement necessitates additional modeling to specify the behavior of the retry buffer, which cannot be adequately specified through a set of temporal properties. Note: To reduce complexity during formal verification, as well as the complexity of specifying the high-level requirements, we recommend that separate high-level requirements be written for the following:

- Packets entering the Wishbone Tx interface
- Packets entering the Rx DLL interface
- Replay request

A characteristic of high-level requirements is that they are written independent of a particular implementation. Hence, the specification would not require change as the implementation undergoes modifications. In fact, the design's implementation might change significantly (for example, re-coding of the RTL to achieve timing closure) as long as the modified design does not violate its spec-level requirement. Implementation or structural assertions, on the other hand, generally require re-coding as the RTL implementation changes.

Figure 8-13 PCI Express high-level requirement



High-level Requirements Model
no packets are dropped, duplicated, or corrupted
Tx DLL
Increment by 1
FIFO Over Flow
One-hot output
TLP
Seq_num
LCRC
Retry buffer
Packet arbiter
DLLP
From Wishbone
Tx
From RX DLL
Rx
To PHY
= Implementation assertions: low-level assertions

For example, the PCI Express data link layer block implementation of our previous high-level requirement might contain multiple FIFOs, state-machines, and other complex structures. As a FIFO depth changes sizes or a state-machine changes encoding, our implementation assertions must be updated accordingly. However, our higher, spec-level requirement, which is specifying the block's expected end-to-end behavior remains unchanged (for example, any implementation is permitted provided the requirements of the data integrity and proper sequences are not violated).

Finally, it is important to note that a high-level requirements model provides less value when used in simulation compared to implementation assertions since they only isolate a problem to the boundary of a block (that is, they do not help isolate the problem to a particular region or line of RTL code). However, since high-level requirements are more comprehensive (that is, they cover more design behavior than lower-level implementation assertions) there is a greater return on investment when they are formally proven.

8.3.3 Modeling high-level requirements

In this section, we use a simple example to illustrate how to create a high-level requirements model. This example does not require specifying data-integrity type properties (for example, properties involving queuing semantics). Hence, we do not present the extra modeling typically required to specify queuing semantics. However, the concepts are easily extended to include this class of properties.

Our simple example in this section could be specified easily through a set of simple to moderately complex temporal properties using either PSL or SystemVerilog. However, recall that often it is more difficult to extract the design intent from a set of declarative properties. Alternatively, creating an architectural abstract FSM with a small set of simpler properties, the specified design intent is generally obvious.

For example, in practice, interface transactions into a block follow a sequence of operations. The expected or correct sequencing of these operations can be modeled architecturally as a state-machine, as demonstrated in the following Verilog fragment in Example 8-33.

Example 8-33 SystemVerilog: Code fragment to model transaction sequence

```
// extra modeling code (not part of design) used with property
// specification

always @ (posedge clk) begin
  if (~rstN)
    prevState <= `IDLE;
  else
    prevState <= currentState;
end // always

always @ (prevState or ...) begin
  case (prevState)
    'IDLE: if (. . .) currentState = 'ADDRESS
       . . . .
       . . . .

    default: currentState = 'ERROR;
  endcase
end // always

// SystemVerilog temporal property specification
property ValidTransaction;
  @ (posedge clk) (currentState != 'ERROR);
endproperty
assert property (ValidTransaction) ;
```

In our architectural (that is, conceptual) state-machine, you can see that from the 'IDLE state of the interface, one of our next possible states we can reach is the `ADDRESS state. Using this form of abstract modeling, it is possible to specify the permissible sequence of events that can occur on an interface (from a black-box perspective). We now can reuse this interface high-level requirements model as an assertion on one block and as a constraint on another.

For example, as a constraint on one block during formal verification, we could assume that the architectural state-machine (representing input sequences) will never enter into an `ERROR state. It has been our experience that a set of properties for complex protocols, such as the PCI Express, can be specified simply by using Verilog as our property's modeling layer language. In fact, these high-level requirements typically are only a few hundred lines of Verilog code (that is, typically less than 1/10th the size of the RTL implementation). The advantage with this approach is that the specified intended behavior becomes clear through the extra modeling. Furthermore, this approach can significantly reduce the complexity and number of temporal properties required to specify an interface (in some interfaces we have seen a 5 to 1 reduction in the number of required temporal properties when additional modeling was used).

8.4 Summary

In this chapter, we presented a set of common ambiguities that arise when interpreting natural language forms of specification. Recognizing these potential problem spots, and then clarifying the design intent with the original author, will greatly improve your efforts to create precise formal properties. We then introduced a set of guidelines that should be followed when creating assertions, assumptions, and functional coverage. Next, we discussed the need for additional modeling for certain classes of properties. For example, we have seen that higher-level forms of specification that combined modeling with a smaller set of temporal properties can be an alternative to a larger set of complex properties—particularly giving clarity to the overall intent of the design, more stable specifications, and higher formal verification value.

Finally, we summarize a set of simple principles defined throughout the book that should be followed when specifying properties for your designs.

- Do not assume your design is correct—assert it! What you do not check is probably broken.
- Do not waste time debugging another block's problem from your block! Validate your inputs with assertions.

- Do not duplicate the original RTL code in the form of an assertion! Describe *what* the design is suppose to do, not *how* the design is to be implemented.
- Do not write assertions to validate simple components! For example, avoid writing assertions for simple multiplexers, registers, and so forth.
- Do not ignore your design assumptions—document them as assertions for others to review and verification to validate.
- Keep your specifications simple!