Chapter 7

CHECKING THE CHECKER

Isolating assertion errors early

Assertion based verification provides excellent potential for finding design bugs early in the verification cycle. The SVA language is defined to address ABV with powerful built-in constructs. Assertion failures are indicated to the user by default as required by the SystemVerilog 3.1a standard. It is not required to display the success of an assertion by default. The user can use the action block of an assertion to display successes. Since the number of successes can be numerous (since most assertions are evaluated on every clock edge), displaying every success by default can create huge log files depending on the number of assertions that are active during simulation, slowing down the simulation.

A typical test configuration is shown in Figure 7-1. This is the same as Figure 0-2 shown in Chapter 0. Let's assume that a user executes this configuration and the simulation completes with a few assertion errors. The user should be absolutely confident that the error issued is a real design error. In other words, a user should be confident that his assertion code is correct and that the assertion failure is not a false condition. Debugging the entire design based on an assertion error is a tough task. If the error issued was due to bad assertion code, a user could waste a lot of time in the verification process. On the other hand, if there are no assertion failures during simulation, the verification engineer should be absolutely confident that the design works. If the assertion is not written accurately, it might not capture the intent of the design and hence, can miss a real error.

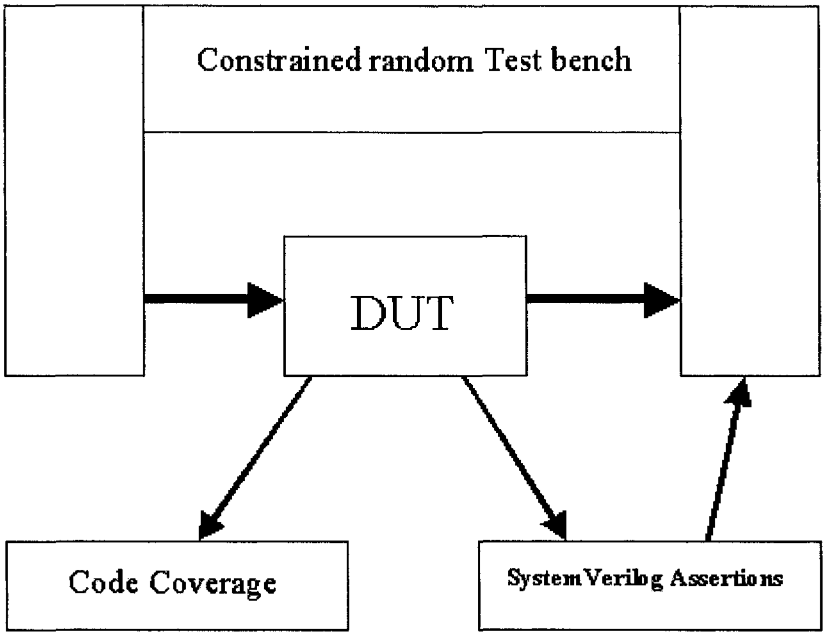

Figure 7-1. Typical simulation configuration

The declarative nature of the language makes SVA checks look very concise. If the checks are not coded well, the real intent of the property may not get represented accurately. It is critical to verify the functionality of the assertion code before binding it to the design. This involves investing some time upfront in the verification process, but it will prevent a user from navigating along a wrong debugging path. This chapter provides a few tips on how to check the checker. A sample configurable testbench for checking assertions that can be written between two signals is discussed in detail. The theory behind the configurable testbench will be used to verify a more complex protocol checker. Assertion validation is a vast topic and we just try exploring a few basic techniques in this chapter.

7.1 Assertion Verification

A simple testbench can be created to verify the functionality of an assertion. In most cases, the number of input conditions for an assertion is a finite number. This assumes that unbounded timing is not used in the definition of the property. An exhaustive testbench could be written to test all possible input conditions. If an unbounded timing is involved, it becomes impossible to create all possible input design conditions. Checkers involving unbounded timing can detect incorrect behavior, but do not fail at the end of simulation if an expected event does not occur. This is because the checker can not assume that a missing event would not happen if the simulation was run a bit longer. Unbounded timing checkers are thus considered incomplete. It is important to realize that even with a bounded time, the number of possible input design conditions can be numerous. This really depends on the complexity of the checker. An assertion is always based on two important concepts, as shown in Figure 7-2:

1. Logical relationship
2. Temporal relationship

If an assertion is written by logically combining (and, or, xor, etc.) an n-bit expression, then the possible number of input conditions is ($2^n - 1$). Consider the logical expression shown below:

```
signal1 && signal2 && signal3
```

This expression will have 8 possible input conditions that need to be tested to guarantee the correct evaluation of the expression.

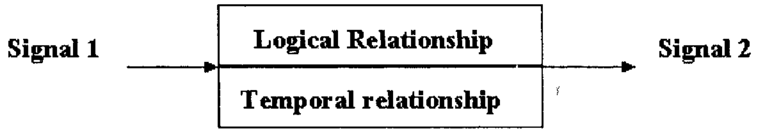

Figure 7-2. Assertion relationship

An assertion that involves a timing relationship between two signals can be tested thoroughly by using the bounds of the minimum and maximum timing limits. For example, consider the following case:

```
signal1 ##[min:max] signal2
```

This expression can fail on two conditions:

1. If the timing between the two signals is less than "min." (This implies that signal 2 arrived before "min" time and did not stay true between "min" and "max" time)
2. If the timing between the two signals is more than "max."

The assertion must succeed for all timings within the window (min and max). This means that varying the timing between signal1 and signal2 from (min-1) to (max+1) will cover all possible successes and at the least one error condition. If the timing between signal1 and signal2 is fixed, then the value of "min" and "max" are same. For a fixed timing relation, all possible successes and at least one error condition can be observed by varying the time from (min-1) to (min+1).

7.2 Assertion Test Bench (ATB) for SVA with two signals

In this section, we show how to create a configurable ATB. The objective is to create a testbench that can generate stimulus for SVA code that is written for two signals. The most basic requirement of an ATB is to get a correct response from the assertion for all possible successes and at least one error. An assertion involving two signals always has a leading signal (LS) and a trailing signal (TS). Consider the examples shown below:

```
signal1 && signal2
signal1 |-> signal2
signal1 ##[1:3] signal2
signal1 |-> ##2 signal2
```

In all these examples, irrespective of whether we are checking for a logical relationship or a timing relationship, we will address signal1 as the leading signal and signal2 as the trailing signal.

7.2.1 Logical relationship between two signals

There are several possible logical relationships between two signals involved in an SVA. Logical relationships are evaluated on a per clock basis. In other words, these are combinational checks.

Figure 7-3 shows a logical relationship tree for two signals. Based on the figure, there are 16 possible logical relationships between two signals. The logical relationship tree involves the following possibilities:

1. Logical relationship between two level sensitive signals.
2. Logical relationship between two edge sensitive signals.
3. Logical relationship between two level sensitive signals with overlapping implication.
4. Logical relationship between two edge sensitive signals with overlapping implication.

Any assertion that involves two level sensitive signals has the following possibilities for the leading signal (LS) and the trailing signal (TS):

a. HH – Both LS and TS are high
b. HL – LS is high and TS is low
c. LH – LS is low and TS is high
d. LL – Both LS and TS are low

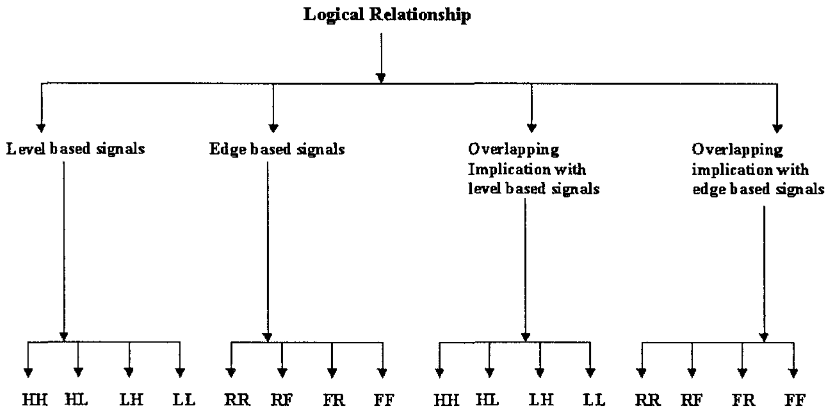

Figure 7-3. Logical relationship tree for SVA with two signals

Any assertion that involves two edge sensitive signals has the following possibilities for the leading signal (LS) and the trailing signal (TS):

a. RR – Both LS and TS have a rising edge
b. RF – LS has a rising edge and TS has a falling edge
c. FR – LS has a falling edge and TS has a rising edge
d. FF – Both LS and TS have falling edges

Note that the overlapping implication is listed as part of the logical relationship tree. If there is no timing involved between the leading signal and the trailing signal, the checker with an overlapping implication is a simple "if" statement. Hence, it can be grouped with the logical relationship tree.

It is possible to have a mix of a level sensitive signal and an edge sensitive signal in the same assertion. These combinations are not listed as part of the tree to simplify the discussion. In a checker, if the leading signal is level sensitive and the trailing signal is edge sensitive, it can be grouped with the level sensitive checks. Similarly, if the leading signal is edge sensitive and the trailing signal is level sensitive, the checker can be grouped with the edge sensitive checks. In the next section, we show how to generate stimulus that can verify all 16 possible relationships, as shown in Figure 7-3 thoroughly.

7.2.2 Stimulus generation for logical relationship – Level sensitive

A list of possible properties for logical relationship between two level sensitive signals is shown below. Note that a logical "and" operator is used in this example. This can be replaced with any other logical operator and the stimulus generation will remain exactly the same.

```
// On a given clock edge, both leading signal and
// trailing signal are high

property p_1_hh;
 @(posedge clk) a && b;
endproperty

// On a given clock edge, the leading signal is
// high and the trailing signal is low

property p_1_hl;
 @(posedge clk) a && !b;
endproperty

// On a given clock edge, the leading signal is
// low and the trailing signal is high

property p_1_lh;
 @(posedge clk) !a && b;
endproperty

// On a given clock edge, both leading signal and
// trailing signal are low

property p_1_ll;
 @(posedge clk) !a && !b;
endproperty

a_1_hh : assert property(p_1_hh);
a_1_hl : assert property(p_1_hl);
```

```
a_1_lh : assert property(p_1_lh);
a_1_ll : assert property(p_1_ll);
```

Overlapping implication is very similar to a simple logical operator with the only difference of the pre-condition. There is a hidden "if" statement that evaluates the trailing signal conditionally. If the leading signal is not true, then the property succeeds by default. A list of possible properties for logical relationship between two level sensitive signals with overlapping implication is shown below.

```
// on a given clock edge, if the leading signal
// is high, check that the trailing signal is
// also high

property p4_oli_hh;
 @(posedge clk) a |-> b;
endproperty

// on a given clock edge, if the leading signal
// is high, check that the trailing signal is
// low

property p4_oli_hl;
 @(posedge clk) a |-> !b;
endproperty

// on a given clock edge, if the leading signal
// is low, check that the trailing signal is
// high

property p4_oli_lh;
 @(posedge clk) !a |-> b;
endproperty

// on a given clock edge, if the leading signal
// is low, check that the trailing signal is
// low

property p4_oli_ll;
 @(posedge clk) !a |-> !b;
endproperty

a4_oli_hh: assert property(p4_oli_hh);
```

```
a4_oli_hl: assert property(p4_oli_hl);
a4_oli_lh: assert property(p4_oli_lh);
a4_oli_ll: assert property(p4_oli_ll);
```

To check a logical relation between two level sensitive signals, there are four possible input conditions (00, 01, 10, 11). By producing stimulus that covers all these four possible conditions, one can verify any logical operation between two level sensitive signals. The same stimulus will also satisfy the 4 conditions (HH, HL, LH, LL) shown in Figure 7-3.

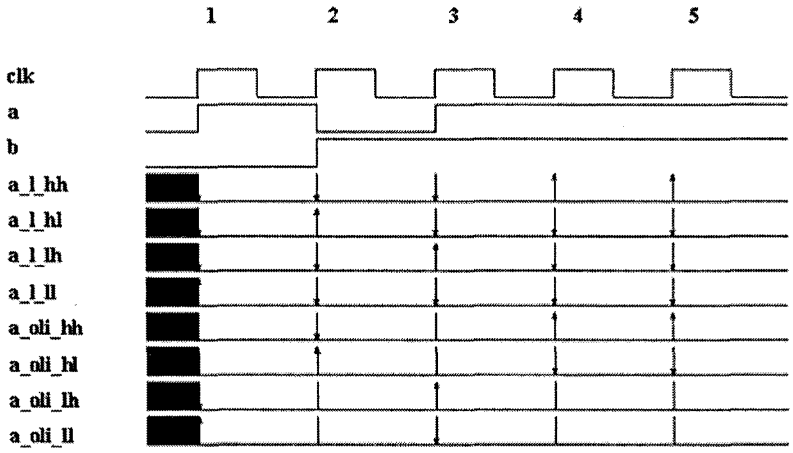

Figure 7-4. Waveform for logical relation between two level sensitive signals

The sample Verilog test code shown below is a simple 2-bit counter. The LSB of the counter drives the leading signal "a" and the MSB of the counter drives the trailing signal "b." The results produced by testing the above properties with the stimulus generated by the sample Verilog test code are shown in Figure 7-4. As shown in the figure, for the stimulus used, every assertion responded correctly for all real success and at least one real error.

```
// sample test code for logical relationship
// between level sensitive signals

logic [1:0] logical_op_reg;
logical_op_reg = 2'b00;

for(i=0; i<4; i++)
```

```
begin
 a <= logical_op_reg[0];
 b <= logical_op_reg[1];
 repeat(1) @(posedge clk);
 logical_op_reg++;
end
```

7.2.3 Stimulus generation for logical relationship – Edge sensitive

A list of possible properties for logical relationship between two edge sensitive signals is shown below.

```
// on a given clock edge the leading signal has a
// falling edge and the trailing signal has a
// falling edge

property p2_ff;
 @(posedge clk) $fell(a) && $fell(b);
endproperty

// on a given clock edge the leading signal has a
// falling edge and the trailing signal has a
// rising edge

property p2_fr;
 @(posedge clk) $fell(a) && $rose(b);
endproperty

// on a given clock edge the leading signal has a
// rising edge and the trailing signal has a
// falling edge

property p2_rf;
 @(posedge clk) $rose(a) && $fell(b);
endproperty

// on a given clock edge the leading signal has a
// rising edge and the trailing signal has a
// rising edge

property p2_rr;
 @(posedge clk) $rose(a) && $rose(b);
endproperty
```

```
a2_ff: assert property(p2_ff);
a2_fr: assert property(p2_fr);
a2_rf: assert property(p2_rf);
a2_rr: assert property(p2_rr);
```

A list of possible properties for logical relationship between two edge sensitive signals with overlapping implication is shown below.

```
// on a given clock edge, if the leading signal
// has a falling edge, then the trailing signal
// must have a falling edge

property p4_oei_ff;
 @(posedge clk) $fell(a) |-> $fell(b);
endproperty

// on a given clock edge, if the leading signal
// has a falling edge, then the trailing signal
// must have a rising edge

property p4_oei_fr;
 @(posedge clk) $fell(a) |-> $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then the trailing signal
// must have a falling edge

property p4_oei_rf;
 @(posedge clk) $rose(a) |-> $fell(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then the trailing signal
// must have a rising edge

property p4_oei_rr;
 @(posedge clk) $rose(a) |-> $rose(b);
endproperty

a4_oei_ff: assert property(p4_oei_ff);
a4_oei_fr: assert property(p4_oei_fr);
a4_oei_rf: assert property(p4_oei_rf);
```

```
a4_oei_rr: assert property(p4_oei_rr);
```

Though we are checking only for logical relationship, the use of **$rose** or **$fell** constructs makes the stimulus generation slightly more complex. The stimulus generated should be capable of satisfying all the edge transitions. The sample Verilog test code shown below is a 2-bit counter and it accommodates all possible successes for both Fall-Fall and Fall-Rise conditions, as shown in the Figure 7-5.

```
// sample test code for logical relationship
// between edge sensitive signals

for(i=0; i<8; i++)
begin
  a <= logical_op_reg[0];
  b <= logical_op_reg[1];
  repeat(1) @(posedge clk);
  logical_op_reg++;
end
```

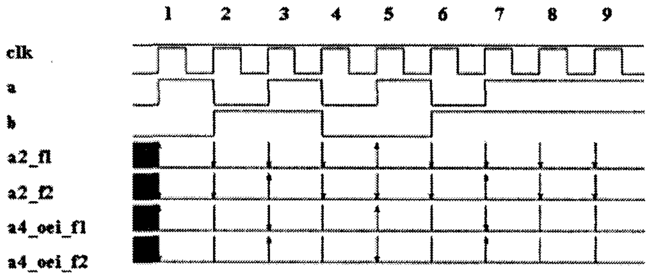

Figure 7-5. Logical condition on edge based signals - FF, FR

The sample Verilog test code shown below is also a 2-bit counter, but uses the negated values of the counter bits. It accommodates all possible successes for both Rise-Fall and Rise-Rise conditions, as shown in the Figure 7-6.

```
// sample test code for logical relationship
// between edge sensitive signals
for(i=0; i<8; i++)
begin
```

```
    a <= !logical_op_reg[0];
    b <= !logical_op_reg[1];
    repeat(1) @(posedge clk);
    logical_op_reg++;
  end
```

As seen in the last two sections, satisfying logical relationships is simple. The possible input conditions increase as the number of signals involved in the logical expression increase.

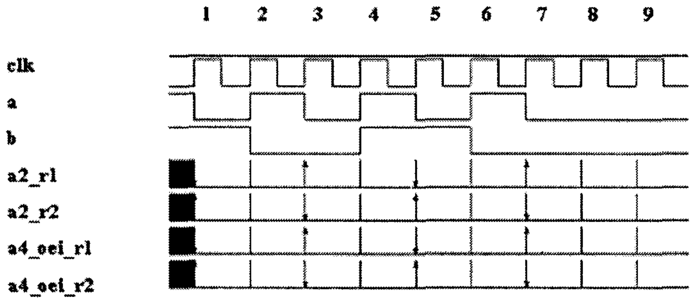

Figure 7-6. Logical condition on edge based signals - RR, RF

7.2.4 Timing relationship between two signals

The timing between the leading signal and the trailing signal can be a fixed delay or a variable delay. A timing relationship tree is very similar to the logic relationship tree except that it has twice the number of possibilities (fixed timing and variable timing) as shown in Figure 7-7. Also note that a timing relationship has a non-overlapping condition between the leading signal and the trailing signal.

The timing relationship tree involves the following possibilities:

1. Fixed timing relationship between two level sensitive signals.
2. Variable timing relationship between two level sensitive signals.
3. Fixed timing relationship between two edge sensitive signals.
4. Variable timing relationship between two edge sensitive signals.
5. Fixed timing relationship between two level sensitive signals with non-overlapping implication.

6. Variable timing relationship between two level sensitive signals with non-overlapping implication.
7. Fixed timing relationship between two edge sensitive signals with non-overlapping implication.
8. Variable timing relationship between two edge sensitive signals with non-overlapping implication.

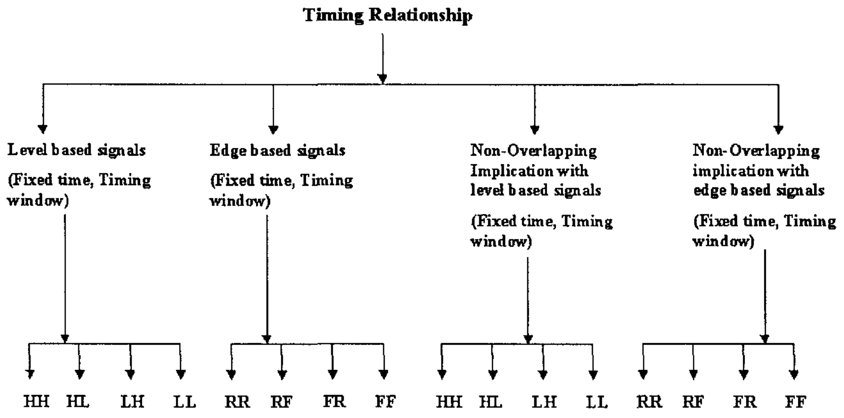
Timing Relationship
Level based signals (Fixed time, Timing window)
Edge based signals (Fixed time, Timing window)
Non-Overlapping Implication with level based signals (Fixed time, Timing window)
Non-Overlapping implication with edge based signals (Fixed time, Timing window)
HH HL LH LL RR RF FR FF HH HL LH LL RR RF FR FF

Figure 7-7. Timing relationship tree

A timing relationship between a level sensitive signal and an edge sensitive signal is possible. To simplify the timing relationship tree, these possibilities are not listed. As mentioned in Section 7.2.1, if the leading signal is level sensitive and the trailing signal is edge sensitive, it can be grouped with the level sensitive checks. Similarly, if the leading signal is edge sensitive and the trailing signal is level sensitive, the checker can be grouped with the edge sensitive checks.

7.2.5 Stimulus generation for timing relationship

A list of possible properties for fixed timing relationship between two level sensitive signals is shown below.

```
// On a given clock edge, the leading signal is
// high and after "min_time" clock cycles the
// trailing signal is high

property p3_hh;
  @(posedge clk) a ##min_time b;
```

```
endproperty

// On a given clock edge, the leading signal is
// high and after "min_time" clock cycles the
// trailing signal is low

property p3_hl;
 @(posedge clk) a ##min_time !b;
endproperty

// On a given clock edge, the leading signal is
// low and after "min_time" clock cycles the
// trailing signal is high

property p3_lh;
 @(posedge clk) !a ##min_time b;
endproperty

// On a given clock edge, the leading signal is
// low and after "min_time" clock cycles the
// trailing signal is low

property p3_ll;
 @(posedge clk) !a ##min_time !b;
endproperty

a3_f1: assert property(p3_hh);
a3_f2: assert property(p3_hl);
a3_f3: assert property(p3_lh);
a3_f4: assert property(p3_ll);
```

A list of possible properties for variable timing relationship between two level sensitive signals is shown below.

```
// On a given clock edge, the leading signal is
// high and between "min_time" and "max_time"
// clock cycles the trailing signal is high

property p3_w1_hh;
 @(posedge clk) a ## [min_time : max_time] b;
endproperty
// On a given clock edge, the leading signal is
// high and between "min_time" and "max_time"
// clock cycles the trailing signal is low

property p3_w2_hl;
 @(posedge clk) a ## [min_time : max_time] !b;
endproperty

// On a given clock edge, the leading signal is
// low and between "min_time" and "max_time"
// clock cycles the trailing signal is high

property p3_w3_lh;
 @(posedge clk) !a ## [min_time : max_time] b;
endproperty

// On a given clock edge, the leading signal is
// low and between "min_time" and "max_time"
// clock cycles the trailing signal is low

property p3_w4_ll;
 @(posedge clk) !a ## [min_time : max_time] !b;
endproperty

a3_w1: assert property(p3_w1_hh);
a3_w2: assert property(p3_w2_hl);
a3_w3: assert property(p3_w3_lh);
a3_w4: assert property(p3_w4_ll);
```

A list of possible properties for fixed timing relationship between two level sensitive signals with non-overlapping implication is shown below.

```
// On a given clock edge, if the leading signal
// is high, then after "min_time" clock cycles
// the trailing signal must be high

property p5_f_hh;
 @(posedge clk) a |-> ##min_time b;
endproperty

// On a given clock edge, if the leading signal
// is high, then after "min_time" clock cycles
// the trailing signal must be low
property p5_f_hl;
 @(posedge clk) a |-> ##min_time !b;
```

```
endproperty

// On a given clock edge, if the leading signal
// is low, then after "min_time" clock cycles
// the trailing signal must be high

property p5_f_lh;
 @(posedge clk) !a |-> ##min_time b;
endproperty

// On a given clock edge, if the leading signal
// is low, then after "min_time" clock cycles
// the trailing signal must be low

property p5_f_ll;
 @(posedge clk) !a |-> ##min_time !b;
endproperty

a5_f_hh: assert property(p5_f_hh);
a5_f_hl: assert property(p5_f_hl);
a5_f_lh: assert property(p5_f_lh);
a5_f_ll: assert property(p5_f_ll);
```

A list of possible properties for variable timing relationship between two level sensitive signals with non-overlapping implication is shown below.

```
// On a given clock edge, if the leading signal
// is high, then between "min_time" and
// "max_time" clock cycles the trailing signal
// must be high

property p5_w_hh;
 @(posedge clk)
 a |-> ##[min_time : max_time] b;
endproperty

// On a given clock edge, if the leading signal
// is high, then between "min_time" and
// "max_time" clock cycles the trailing signal
// must be low

property p5_w_hl;
 @(posedge clk)
```

```
  a |-> ##[min_time : max_time] !b;
endproperty

// On a given clock edge, if the leading signal
// is low, then between "min_time" and
// "max_time" clock cycles the trailing signal
// must be high

property p5_w_lh;
  @(posedge clk)
  !a |-> ##[min_time : max_time] b;
endproperty

// On a given clock edge, if the leading signal
// is low, then between "min_time" and
// "max_time" clock cycles the trailing signal
// must be low

property p5_w_ll;
  @(posedge clk)
  !a |-> ##[min_time : max_time] !b;
endproperty

a5_w_hh: assert property(p5_w_hh);
a5_w_hl: assert property(p5_w_hl);
a5_w_lh: assert property(p5_w_lh);
a5_w_ll: assert property(p5_w_ll);
```

A list of possible properties for fixed timing relationship between two edge sensitive signals is shown below.

```
// on a given clock edge, the leading signal has
// a falling edge and after "min_time" cycle the
// trailing signal has a falling edge

property p4_f_ff;
  @(posedge clk) $fell(a) ##min_time $fell(b);
endproperty

// on a given clock edge, the leading signal has
// a rising edge and after "min_time" cycle the
// trailing signal has a rising edge
```

```
property p4_f_rr;
 @(posedge clk) $rose(a) ##min_time $rose(b);
endproperty

// on a given clock edge, the leading signal has
// a falling edge and after "min_time" cycle the
// trailing signal has a rising edge

property p4_f_fr;
 @(posedge clk) $fell(a) ##min_time $rose(b);
endproperty

// on a given clock edge, the leading signal has
// a rising edge and after "min_time" cycles the
// trailing signal has a falling edge

property p4_f_rf;
 @(posedge clk) $rose(a) ##min_time $fell(b);
endproperty

a4_f_rr: assert property(p4_f_rr);
a4_f_ff: assert property(p4_f_ff);
a4_f_rf: assert property(p4_f_rf);
a4_f_fr: assert property(p4_f_fr);
```

A list of possible properties for variable timing relationship between two edge sensitive signals is shown below.

```
// on a given clock edge, the leading signal has
// a falling edge and within "min_time" to
// "max_time" cycles the trailing signal has a
// falling edge

property p4_w_ff;
 @(posedge clk)
 $fell(a) ##[min_time : max_time] $fell(b);
endproperty

// on a given clock edge, the leading signal has
// a rising edge and within "min_time" to
// "max_time" cycles the trailing signal has a
// rising edge
```

```
property p4_w_rr;
 @(posedge clk)
 $rose(a) ##[min_time : max_time] $rose(b);
endproperty

// on a given clock edge, the leading signal has
// a falling edge and within "min_time" to
// "max_time" cycles the trailing signal has a
// rising edge

property p4_w_fr;
 @(posedge clk)
 $fell(a) ##[min_time : max_time] $rose(b);
endproperty

// on a given clock edge, the leading signal has
// a rising edge and within "min_time" to
// "max_time" cycles the trailing signal has a
// falling edge

property p4_w_rf;
 @(posedge clk)
 $rose(a) ##[min_time : max_time] $fell(b);
endproperty

a4_w_rr: assert property(p4_w_rr);
a4_w_ff: assert property(p4_w_ff);
a4_w_rf: assert property(p4_w_rf);
a4_w_fr: assert property(p4_w_fr);
```

A list of possible properties for fixed timing relationship between two edge sensitive signals with non-overlapping implication is shown below.

```
// on a given clock edge, if the leading signal
// has a falling edge, then after "min_time"
// cycles the trailing signal must have a
// falling edge

property p6_f_ff;
 @(posedge clk)
 $fell(a) |-> ##min_time $fell(b);
endproperty
```

```
// on a given clock edge, if the leading signal
// has a rising edge, then after "min_time"
// cycles the trailing signal must have a
// rising edge

property p6_f_rr;
 @(posedge clk)
 $rose(a) |-> ##min_time $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a falling edge, then after "min_time"
// cycles the trailing signal must have a
// rising edge

property p6_f_fr;
 @(posedge clk)
 $fell(a) |-> ##min_time $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then after "min_time"
// cycles the trailing signal must have a
// falling edge

property p6_f_rf;
 @(posedge clk)
 $rose(a) |-> ##min_time $fell(b);
endproperty
a6_f_rr: assert property(p6_f_rr);
a6_f_ff: assert property(p6_f_ff);
a6_f_rf: assert property(p6_f_rf);
a6_f_fr: assert property(p6_f_fr);
```

A list of possible properties for variable timing relationship between two edge sensitive signals with non-overlapping implication is shown below.

```
// on a given clock edge, if the leading signal
// has a falling edge, then within "min_time" to
// "max_time" cycles the trailing signal must
// have a falling edge
property p6_w_ff;
 @(posedge clk)
```

```
  $fell(a) |-> ##[min_time : max_time] $fell(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then within "min_time" to
// "max_time" cycles the trailing signal must
// have a rising edge

property p6_w_rr;
  @(posedge clk)
  $rose(a) |-> ##[min_time : max_time] $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a falling edge, then within "min_time" to
// "max_time" cycles the trailing signal must
// have a rising edge

property p6_w_fr;
  @(posedge clk)
  $fell(a) |-> ##[min_time : max_time] $rose(b);
endproperty

// on a given clock edge, if the leading signal
// has a rising edge, then within "min_time" to
// "max_time" cycles the trailing signal must
// have a falling edge

property p6_w_rf;
  @(posedge clk)
  $rose(a) |-> ##[min_time : max_time] $fell(b);
endproperty

a6_w_rr: assert property(p6_w_rr);
a6_w_ff: assert property(p6_w_ff);
a6_w_rf: assert property(p6_w_rf);
a6_w_fr: assert property(p6_w_fr);
```

The following sample Verilog test code can generate stimulus that will satisfy all timing relationships defined between two signals except eventuality. Note that the signals "a" and "b" are initialized based on what the "timing level" is set to. For example, if the "timing level" is set to a "10," then the test code will generate stimulus for an "active high" level on the leading signal and an "active low" level on the trailing signal, if it is a level sensitive checker. For an edge sensitve checker, if the "timing level" is set to a "10," then the test code will generate stimulus for a "rising edge" on the leading signal and a "falling edge" on the trailing signal. A simple "for" loop is used to generate timing windows starting from (min_time-1) to (max_time+3). This will make sure that all possible successes are created and at least one error is created. The same results can be achieved with an upper window of (max_time+1). By using (max_time+3), we produce more error conditions. If the timing is fixed, then the values of "min_time" and "max_time" are the same.

Figure 7-8 shows a sample waveform of a fixed timing relationship property between two level sensitive signals (leading signal is active low and trailing signal is active high). Figure 7-9 shows a sample waveform of a variable timing relationship property between two edge sensitive signals (leading signal and the trailing signal look for a rising edge).

```
// sample Verilog test code for timing
// relationship between two signals

if(timing_level == 2'b11) begin
a = 1'b0; b=1'b0; end
if(timing_level== 2'b00) begin
a = 1'b1; b=1'b1; end
if(timing_level == 2'b01) begin
a = 1'b1; b=1'b0; end
if(timing_level == 2'b10) begin
a = 1'b0; b=1'b1; end

for(i=(min_time-1); i<(max_time+3); i++)
begin
  repeat(1) @(posedge clk);
  a <= ~a;
  if(i == 0)
  begin
    b <= ~b;
    repeat(1) @(posedge clk);
    a <= ~a; b <= ~b;
  end
  else
  begin
    repeat(1) @(posedge clk);
    a<= ~a;
    repeat((i-1)) @(posedge clk);
    b<= ~b;
    repeat(1) @(posedge clk);
    b<= ~b;
  end
end
```

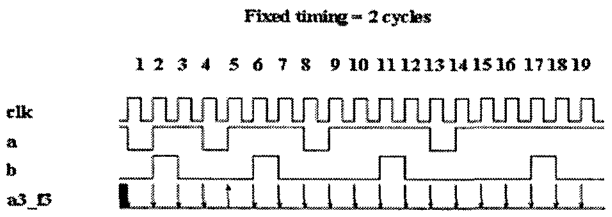

Figure 7-8. Timing (fixed) between two level sensitive signals

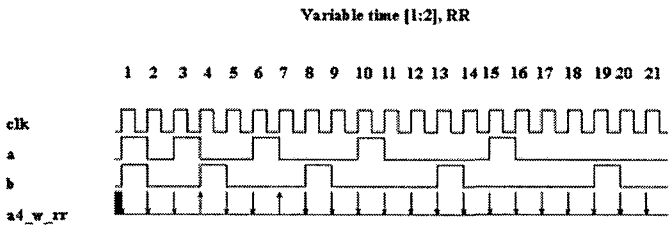

Figure 7-9. Timing (variable) between two edge sensitive signals

Timing relationships can be tested by looping from (min-1) to (max+1) values. This will guarantee that all possible successes are tested and at least one failure is tested. Repetition of signals is an extension of timing relationships. Repetitions also involve timing relationships, but it requires that the leading signal or the trailing signal repeat its value for a defined number of cycles. Repetition relationships are discussed in detail in the next section.

7.2.6 Repetition relationship between two signals

There are two main categories of repetition between two signals:

1. **Repeat after** – After an expected edge on the leading signal, with or without a time delay, the trailing signal is expected to repeat "n" times.

2. **Repeat until** – After an expected edge on the leading signal, the leading signal repeats until the expected value arrives on the trailing signal.

The repeat operators often involve the combination of an edge sensitive signal and a level sensitive signal. The leading signal is often an edge sensitive signal and the trailing signal is often a level sensitive signal. The "repeat until" condition can have an edge sensitive signal as a trailing signal. The naming convention for the repetition properties is as follows:

RH – LS has a rising edge and TS is high
RL – LS has a rising edge and TS is low
FH – LS has a falling edge and TS is high
FL – LS has a falling edge and TS is low

A list of possible properties for "repeat after" relationship between two signals is shown below.

```
// on a given clock edge the leading signal has a
// rising edge and after "start_wait" cycles, the
// trailing signal is high "repetition" times

property p7_c_rpt_rh;
  @(posedge clk)
  $rose(a) ##start_wait b[*repetition];
endproperty

// on a given clock edge the leading signal has a
// rising edge and after "start_wait" cycles, the
// trailing signal is low "repetition" times

property p7_c_rpt_rl;
  @(posedge clk)
  $rose(a) ##start_wait !b[*repetition];
endproperty

// on a given clock edge the leading signal has a
// falling edge and after "start_wait" cycles,
// the trailing signal is high "repetition" times

property p7_c_rpt_fh;
  @(posedge clk)
  $fell(a) ##start_wait b[*repetition];
endproperty
```

```
// on a given clock edge the leading signal has a
// falling edge and after "start_wait" cycles,
// the trailing signal is low "repetition" times

property p7_c_rpt_fl;
 @(posedge clk)
 $fell(a) ##start_wait !b[*repetition];
endproperty
```

A list of possible properties for "repeat until" relationship between two signals is shown below.

```
// on a give clock edge, the leading signal has a
// rising edge and stays high until the trailing
// signal is low

property p7_cu_rpt_rl;
 @(posedge clk) $rose(b) ##0 b[*1:$] ##1 !a;
endproperty

// on a given clock edge, the leading signal has
// a falling edge and stays low until the
// trailing signal is low

property p7_cu_rpt_fl;
 @(posedge clk) $fell(b) ##0 !b[*1:$] ##1 !a;
endproperty

// on a given clock edge, the leading signal has
// a rising edge and stays high until the
// trailing signal is high

property p7_cu_rpt_rh;
 @(posedge clk) $rose(b) ##0 b[*1:$] ##1 a;
endproperty

// on a given clock edge, the leading signal has
// a falling edge and stays high until the
// trailing signal is high

property p7_cu_rpt_fh;
 @(posedge clk) $fell(b) ##0 !b[*1:$] ##1 a;
endproperty
```

A sample code that can generate stimulus to verify both the "repeat after" and "repeat until" conditions for all possible properties is shown below. The stimulus uses the same concept as that of the timing. It loops between (repetition–1) and (repetition+3) to cover all possible successes and at the least one error condition. The variable "start_wait" defines the number of cycles to wait before looking for the repetition on the trailing signal (relevant to the "repeat_after" condition). The variable "stop_wait" defines the number of cycle to wait before re-setting the leading signal (relevant to the "repeat_until" condition).

```
// sample Verilog test code for repetition

logic [1:0] stop_wait;

if(rpt_edge == 2'b11) begin
a = 1'b0; b=1'b0; end

if(rpt_edge == 2'b00) begin
a = 1'b1; b=1'b1; end

if(rpt_edge == 2'b01) begin
a = 1'b1; b=1'b0; end

if(rpt_edge == 2'b10) begin
a = 1'b0; b=1'b1; end

for(i=(repetition-1); i<(repetition+3); i++)
begin
  repeat(1) @(posedge clk);
  a <= ~a;
  repeat(start_wait) @(posedge clk);
  b <= ~b;

  // consecutive repeat condition

    repeat((i)) @(posedge clk);
    b <= ~b;
    stop_wait <= $random() % 4;
    repeat(stop_wait[0]) @(posedge clk);
    a <= ~a;
  end
```

Figure 7-10 shows a sample waveform for a "repeat after" condition. Figure 7-11 shows a sample waveform for a "repeat until" condition.

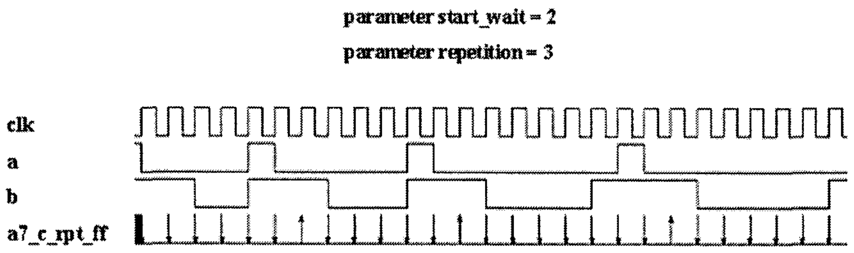

Figure 7-10. Waveform for "repeat after" condition

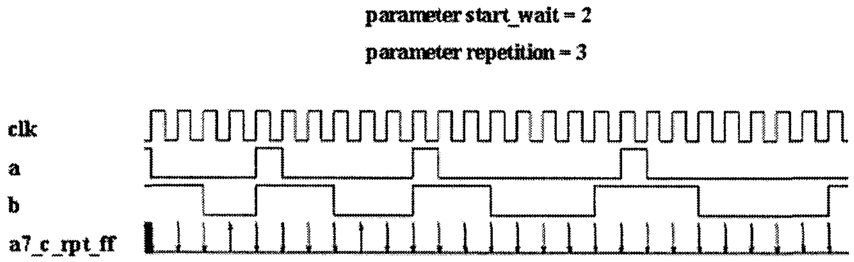

Figure 7-11. Waveform for "repeat until" condition

A few possible relationships between 2 signals were discussed so for. The SVA constructs are abundant and there is a big list of possible relationships between 2 signals. We are not trying to produce a solution for all of those cases. What we have is a small part of the solution. The solution becomes more difficult as the number of signals involved increase.

7.2.7 Environment for ATB involving two signals

In the last few sections, we saw how an exhaustive set of stimulus can be generated to test different relationships between two signals. Several sample Verilog test codes that can satisfy different relationships were shown. In this section, we put the pieces together to create a single configurable testing structure, as shown in Figure 7-12.

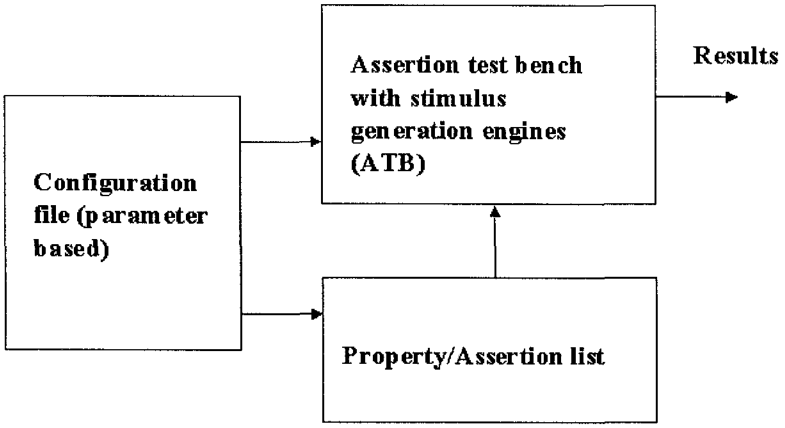
Assertion test bench with stimulus generation engines (ATB)
Results
Configuration file (parameter based)
Property/Assertion list

Figure 7-12. ATB Environment

There are three parts in the testing structure:

1. A parameter based configuration file wherein the user can specify the relationship he wishes to test.
2. A SVA listing file containing both the property definitions and the code that will selectively assert properties, based on the parameter configuration.
3. The top-level Verilog test code containing the various stimulus generation blocks discussed in the previous sections. Based on the parameter configuration, the relevant stimulus generation block will be executed to thoroughly verify the current property under test. The parameter definitions are shown in Table 7-1.

Table 7-1. Parameter definitions

Parameter	Functionality
parameter sig_edge = 0;	Defines if signals involved are edge sensitive, 0 indicates no, 1 indicates yes
parameter sig1_edge = 1;	Defines the edge of the leading signal, 1 means rising edge and 0 means falling edge (used only for logical relationship)
parameter logic_op = 0;	Defines if the assertions involve logical relationship, 0 indicates no, 1indicates yes
parameter timing = 1;	Defines if the assertions involve temporal relationship, 0 indicates no, 1 indicates yes
parameter min_time = 2;	Defines timing information, maximum time should be

Parameter	Functionality
parameter max_time = 2;	greater than the minimum time, minimum time cannot be zero, maximum time should be bounded
parameter timing_level = 2'b10;	Defines levels/edges of the signals, "10" means HL for level and RF for edge
parameter o_l_implication = 0;	Parameter to indicate overlapping implication with level sensitive signals, 0 means no, 1 is yes
parameter o_e_implication = 0;	Parameter to indicate overlapping implication with edge sensitive signals
parameter non_o_implication = 1;	Parameter to indicate non-overlapping implication
parameter rpt_me = 0;	Parameter to indicate repetitions are involved
parameter rpt_edge = 2'b00;	Parameter defining the levels/edges of signals
parameter start_wait = 2;	Paramter to define the wait period before repetition in "repeat after"
parameter repetition = 3;	Parameter to define number of repetitions. Repetition value has to be greater than 1
parameter c_rpt = 0;	Parameter indicating the repeat after test
parameter c_rpt_until = 0;	Parameter indicating the repeat until test

By setting the right parameters, a user can generate stimulus for a specific relationship.

- If the parameter "logic_op" is set to 1 and the other parameters are set to 0, then the ATB will generate stimulus for "logical relationship" between two level sensitive signals.

- If the parameter "timing" is set to 1 and all other parameters are set to 0, then the ATB will generate stimulus for "timing relationship" between two signals. A user can specify whether he wants a fixed time or a variable time relation by setting the values of the parameters "min_time" and "max_time." If the user sets the "sig_edge" parameter to 1, then the signals will be treated as edge sensitive.

A sample SVA listing file used for the verification of SVA involving two signals is shown below.

```
module sig_sva (a, b, clk);

// include the parameter definitions
```

```
`include "config.v"

input logic a, b, clk;

// List definitions of all properties under test

// code to selectively include assertions

always@(posedge clk)
begin

// logical relationship between two level
// sensitive signals

if(logic_op == 1 && timing == 0 && sig_edge == 0)
begin
a_1_hh : assert property(p_1_hh);
a_1_hl : assert property(p_1_hl);
a_1_lh : assert property(p_1_lh);
a_1_ll : assert property(p_1_ll);
end

// logical relationship between two edge
// sensitive signals FF,FR

if(logic_op == 1 && timing == 0 && sig_edge == 1
&& sig1_edge == 0)
begin
a2_ff: assert property(p2_ff);
a2_fr: assert property(p2_fr);
end

// logical relationship between 2 edge sensitive
// signals RF,RR

if(logic_op == 1 && timing == 0 && sig_edge == 1
&& sig1_edge == 1)
begin
a2_rf: assert property(p2_rf);
a2_rf: assert property(p2_rr);
end

// timing relationship between 2 level sensitive
// signals

if(logic_op == 0 && timing == 1 && sig_edge == 0
&& non_o_implication == 0)
begin
if(min_time == max_time)
begin
if(timing_level == 2'b11)
 a3_hh: assert property(p3_hh);
if(timing_level == 2'b10)
 a3_hl: assert property(p3_hl);
if(timing_level == 2'b01)
 a3_lh: assert property(p3_lh);
if(timing_level == 2'b00)
 a3_ll: assert property(p3_ll);
end
if(min_time != max_time)
begin
if(timing_level == 2'b11)
 a3_w1_hh: assert property(p3_w1_hh);
if(timing_level == 2'b10)
 a3_w2_hl: assert property(p3_w2_hl);
if(timing_level == 2'b01)
 a3_w3_lh: assert property(p3_w3_lh);
if(timing_level == 2'b00)
 a3_w4_ll: assert property(p3_w4_ll);
end
end

// logical relationship between 2 level sensitive
// signals with overlapping implication

if((logic_op == 1 || o_l_implication == 1) &&
timing == 0 && sig_edge == 0)
begin
a4_oli_hh: assert property(p4_oli_hh);
a4_oli_hl: assert property(p4_oli_hl);
a4_oli_lh: assert property(p4_oli_lh);
a4_oli_ll: assert property(p4_oli_ll);
end

// logical relationship between 2 edge sensitive
// signals with overlapping implication FF, FR
```

```
if((logic_op == 1 || o_e_implication == 1) &&
timing == 0 && sig_edge == 1 && sig1_edge == 0)
begin
a4_oei_ll: assert property(p4_oei_ll);
a4_oei_lh: assert property(p4_oei_lh);
end

// logical relationship between 2 edge sensitive
// signals with overlapping implication RF, RR

if((logic_op == 1 || o_e_implication == 1) &&
timing == 0 && sig_edge == 1 && sig1_edge == 1)
begin
a4_oei_hl: assert property(p4_oei_hl);
a4_oei_hh: assert property(p4_oei_hh);
end

if(logic_op == 0 && timing == 1 && sig_edge == 1
&& non_o_implication == 0)
begin
if(min_time == max_time)
begin
a4_f_rr: assert property(p4_f_rr);
a4_f_ff: assert property(p4_f_ff);
a4_f_rf: assert property(p4_f_rf);
a4_f_fr: assert property(p4_f_fr);
end
if(min_time != max_time)
begin
a4_w_rr: assert property(p4_w_rr);
a4_w_ff: assert property(p4_w_ff);
a4_w_rf: assert property(p4_w_rf);
a4_w_fr: assert property(p4_w_fr);
end
end

// timing relation between 2 level sensitive
// signals with non-overlapping implication

if(logic_op == 0 && timing == 1 && sig_edge == 0
&& non_o_implication == 1)
begin
if(min_time == max_time)
```

```
begin
if(timing_level == 2'b11)
 a5_f_hh: assert property(p5_f_hh);
if(timing_level == 2'b10)
 a5_f_hl: assert property(p5_f_hl);
if(timing_level == 2'b01)
 a5_f_lh: assert property(p5_f_lh);
if(timing_level == 2'b00)
 a5_f_ll: assert property(p5_f_ll);
end
if(min_time != max_time)
begin
if(timing_level == 2'b11)
 a5_w_hh: assert property(p5_w_hh);
if(timing_level == 2'b10)
 a5_w_hl: assert property(p5_w_hl);
if(timing_level == 2'b01)
 a5_w_lh: assert property(p5_w_lh);
if(timing_level == 2'b00)
 a5_w_ll: assert property(p5_w_ll);
end
end

// timing relation between 2 edge sensitive
// signals with non-overlapping implication

if(logic_op == 0 && timing == 1 && sig_edge == 1
&& non_o_implication == 1)
begin

if(min_time == max_time)
begin
a6_f_rr: assert property(p6_f_rr);
a6_f_ff: assert property(p6_f_ff);
a6_f_rf: assert property(p6_f_rf);
a6_f_fr: assert property(p6_f_fr);
end
if(min_time != max_time)
begin
a6_w_rr: assert property(p6_w_rr);
a6_w_ff: assert property(p6_w_ff);
a6_w_rf: assert property(p6_w_rf);
a6_w_fr: assert property(p6_w_fr);
```

```
  end
  end

  // repetition relationship

  if(rpt_me == 1 && c_rpt == 1 && rpt_edge ==
  2'b11)
  begin
  a7_c_rpt_rh: assert property(p7_c_rpt_rh);
  a7_cu_rpt_rh: assert property(p7_cu_rpt_rh);
  end

  if(rpt_me == 1 && c_rpt == 1 && rpt_edge ==
  2'b10)
  begin
  a7_c_rpt_rl: assert property(p7_c_rpt_rl);
  a7_cu_rpt_rl: assert property(p7_cu_rpt_rl);
  end

  if(rpt_me == 1 && c_rpt == 1 && rpt_edge ==
  2'b01)
  begin
  a7_c_rpt_fh: assert property(p7_c_rpt_fh);
  a7_cu_rpt_fh: assert property(p7_cu_rpt_fh);
  end

  if(rpt_me == 1 && c_rpt == 1 && rpt_edge ==
  2'b00)
  begin
  a7_c_rpt_fl: assert property(p7_c_rpt_fl);
  a7_cu_rpt_fl: assert property(p7_cu_rpt_fl);
  end
  end

// Config Parameters illegal values. If
// logical_op is asserted then timing cannot be
// asserted

 property config_check1;
   @(posedge clk)
         (logic_op == 1) |->
                        (timing == 0);
 endproperty
```

```
// only one of the implication operators can be
// asserted at any time

property config_check2;
  @(posedge clk)
      $onehot0({o_l_implication,
      o_e_implication, non_o_implication});
endproperty

// min_time should be atleast 1

property config_check3;
  @(posedge clk)
     (timing == 1) |-> (min_time >= 1);
endproperty

// repetition should be greater that one

property config_check4;
  @(posedge clk)
      ((c_rpt == 1) && (rpt_me == 1)) |->
                 (repetition > 1);
endproperty

a_check1: assert property(config_check1);
a_check2: assert property(config_check2);
a_check3: assert property(config_check3);
a_check4: assert property(config_check4);

 endmodule
```

The ATB gets executed based on the parameter configuration. A sample ATB used for the verification of SVA involving two signals is shown below.

```
module sig_sva_tb;

logic a,b;
logic clk;
logic [1:0] rpt_wait;
logic [1:0] stop_wait;

`include  "config.v"

integer i,j;
```

```
logic [1:0] logical_op_reg;

initial
begin
clk = 1'b0; a=1'b0; b=1'b0;
logical_op_reg = 2'b00;

//************************************************
// case 1
// logical operation, overlapping implication
// level sensitive signals
//************************************************

if((logic_op == 1 || o_l_implication == 1) &&
timing == 0 && sig_edge == 0)
begin
for(i=0; i<4; i++)
begin
 a <= logical_op_reg[0];
 b <= logical_op_reg[1];
 repeat(1) @(posedge clk);
 logical_op_reg++;
end
end

//************************************************
// case 2
// logical operation, overlapping implication
// edge sensitive signals
//************************************************

if((logic_op == 1 || o_e_implication == 1) &&
timing == 0 && sig_edge == 1)
begin

if(sig1_edge == 0) // ff, fr
begin
for(i=0; i<8; i++)
begin
 a <= logical_op_reg[0];
 b <= logical_op_reg[1];
 repeat(1) @(posedge clk);
 logical_op_reg++;
```

```
end
end

if(sig1_edge == 1) // rr, rf
begin
for(i=0; i<8; i++)
begin
 a <= !logical_op_reg[0];
 b <= !logical_op_reg[1];
 repeat(1) @(posedge clk);
 logical_op_reg++;
end
end

end

//*********************************************
// case 3
// timing relation between 2 signals
//*********************************************

if(logic_op == 0 && timing == 1)
begin

if(timing_level == 2'b11) begin
a = 1'b0; b=1'b0;
end

if(timing_level== 2'b00) begin
a = 1'b1; b=1'b1;
end

if(timing_level == 2'b01) begin
a = 1'b1; b=1'b0;
end

if(timing_level == 2'b10) begin
a = 1'b0; b=1'b1;
end
for(i=(min_time-1); i<(max_time+3); i++)
begin
 repeat(1) @(posedge clk);
 a <= ~a;
```

```
   if(i == 0)
   begin
     b <= ~b;
     repeat(1) @(posedge clk);
     a <= ~a; b <= ~b;
   end
   else
   begin
     repeat(1) @(posedge clk);
     a<= ~a;
     repeat((i-1)) @(posedge clk);
     b<= ~b;
     repeat(1) @(posedge clk);
     b<= ~b;
   end
 end
 end

 //************************************************
 // case 4
 // repetitions
 //************************************************

 if(rpt_me == 1)
 begin

 if(rpt_edge == 2'b11) begin
 a = 1'b0; b=1'b0;
 end

 if(rpt_edge == 2'b00) begin
 a = 1'b1; b=1'b1;
 end

 if(rpt_edge == 2'b01) begin
 a = 1'b1; b=1'b0;
 end

 if(rpt_edge == 2'b10) begin
 a = 1'b0; b=1'b1;
 end

 if(c_rpt == 1)
```

```
begin
for(i=(repetition-1); i<(repetition+3); i++)
begin
 repeat(1) @(posedge clk);
 a <= ~a;
 repeat(start_wait) @(posedge clk);
 b <= ~b;

 // consecutive repeat condition

 repeat((i)) @(posedge clk);
 b <= ~b;
 stop_wait <= $random() % 4;
 repeat(stop_wait[0]) @(posedge clk);
 a <= ~a;
end
end
end

repeat(2) @(posedge clk);
$finish();
end

initial
forever clk = #25 ~clk;

endmodule

bind sig_sva_tb sig_sva i1 (a, b, clk);
```

Note that the configuration file is included into the ATB and the SVA listing file is bound to the ATB module.

7.3 ATB example for a PCI Checker

In this section, we take a complex SVA checker and show how to verify its functionality based on the concepts discussed in Section 7.2. Following is a checker discussed in Chapter 6.

"Target latency for the completion of the first data phase is 16 cycles from the assertion of the signal `framen`"

```
sequence s_tchk9a;
@(posedge clk)
(!irdyn && !trdyn);
endsequence

sequence s_tchk9b;
@(posedge clk)
(!irdyn && !stopn);
endsequence

sequence s_tchk9_fast;
@(posedge clk)
$fell(framen) ##1 $fell(devseln);
endsequence
property p_tchk9_fast;
@(posedge clk)
s_tchk9_fast |->
(!framen && !devseln) throughout
(##[1:15] (s_tchk9a.ended || s_tchk9b.ended));
endproperty

a_tchk9_fast: assert property(p_tchk9_fast);
c_tchk9_fast: cover property(p_tchk9_fast);
```

The property p_tchk9_fast involves complex logical relationship and temporal relationship. The property becomes active when a valid start condition happens (s_tchk9_fast). Once the antecedent matches, the property can succeed in two different ways. Either s_tchk9a or s_tchk9b should match within 1 to 15 cycles, assuming the two signals that helped the antecedent match remain asserted throughout the evaluation.

To verify this check **exhaustively**, the following should be done:

1. All possible logical relationships between the signals "irdyn," "trdyn," "devseln" and "stopn" should be tested.
2. All possible temporal relationships between the antecedent and the consequent should be tested.

Based on the theory from Section 7.2, the possible logical combination for the four signals is 16, as shown in Table 7-2. Upon a successful match of the antecedent, any one of the success condition shown in Table 7-2 should occur within 1 to 15 clock cycles. By looping the checker from "(min-1)" to "(max+1)," which is 0 to 16, one can simulate all possible successes for all possible delay conditions and at least one error condition. While looping in a specific delay value, all 16 possible logical relationships should be executed. Hence, there are 16 possible time slots and 16 possible logical conditions, leading to 256 possible scenarios. The check will pass on three conditions in each of the delay values starting from 1 to 15 and hence, there should be 45 real successes for this check. In other words, this check can succeed in 45 different conditions. This metric can be cross checked by writing cover statements for the property under validation.

Table 7-2. Logical conditions for PCI check

irdyn	trdyn	devseln	stopn	Status
0	0	0	0	Success (s_tchk9a && s_tchk9b)
0	0	0	1	Success (s_tchk9a)
0	0	1	0	Failure
0	0	1	1	Failure
0	1	0	0	Success (s_tchk9b)
0	1	0	1	Failure
0	1	1	0	Failure
0	1	1	1	Failure
1	0	0	0	Failure
1	0	0	1	Failure
1	0	1	0	Failure
1	0	1	1	Failure
1	1	0	0	Failure
1	1	0	1	Failure
1	1	1	0	Failure
1	1	1	1	Failure

A sample Verilog code that generates stimulus to satisfy these 256 different scenarios is shown below.

```
module ctc_complex;

logic irdyn, trdyn, devseln, stopn, framen;
integer i, j;
logic clk;
logic [3:0] test_expr;

assign irdyn = test_expr[3];
assign trdyn = test_expr[2];
assign devseln = test_expr[1];
assign stopn = test_expr[0];
```

```
initial begin
clk = 1'b0; test_expr = 4'd15;
repeat(2) @(posedge clk);

for(i=1; i<17; i++) // timing loop
begin
for(j=0; j<16; j++) // logical loop
begin
  framen = 1'b0;
  repeat(1) @(posedge clk);
  test_expr = 4'b1101;
  repeat(i) @(posedge clk);
  test_expr = j;
  repeat(1) @(posedge clk);
  framen = 1'b1;
  repeat(1) @(posedge clk);
  test_expr = 4'b1111;
  repeat(1) @(posedge clk);
end
end

repeat(2) @(posedge clk);
$finish;
end

initial forever clk = #25 ~clk;

endmodule
```

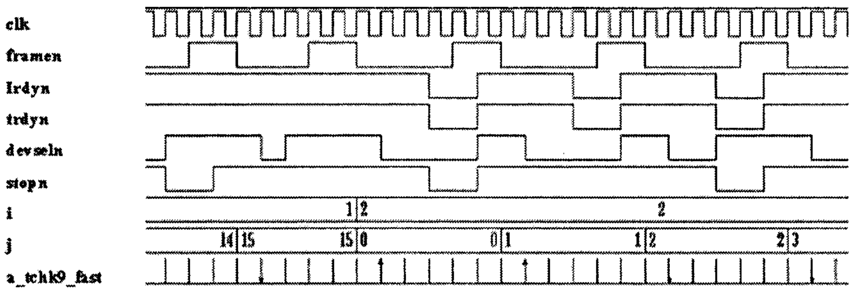

Figure 7-13. PCI Checker verification

Note that the timing constraint is used as the outer loop and the logical constraint as the inner loop. The simulation of this test code on the checker should produce 211 failures and 45 successes. This guarantees that the checker responds correctly for all possible input conditions. A sample waveform from this test is shown in Figure 7-13.

7.4 Summary for checking the checker

- The possible number of relationships between two signals can grow exponentially within the SVA domain.
- By exploring just three of these relationships (logic, timing and repetition) between 2 signals, 56 possible assertion statements were written. As the number of signals involved in an SVA definition increases, the possible assertion statements will increase exponentially.
- It is critical to have an automated way to validate these assertions. With basic Verilog language we were able to create some very effective stimulus generation schemes that tested the assertions thoroughly.
- The same stimulus generation methodology was applied on a real life PCI checker to verify its correctness.
- As the assertions get more complex, the advanced features of constrained random testbenches can be used effectively to check these assertions. Self checking mechanisms can be used to analyze the results of the assertion validation.
- While there is no automated way of checking the checker yet, a user can still verify them like any other design module using testbenches.
- Without a "checking the checker" methodology, a user will not know if the design is working or failing, or if the checker was written incorrectly.
- This process demands some time investment in the beginning of verification process, but can be a huge time saver in the latter part of verification.