The property is asserted on line 8 using the syntax specified above. In contrast to an immediate assertion, a concurrent assertion includes the keyword "property" and the property's name. Here, only a fail-Statement is provided, specifying to print "oops". Historically, other four letter words have been used.

The property definition and assertion are defined inside a module but outside of procedural constructs such as always and initial blocks.

The property specifies the following sequence of values, shown in Figure 9.1 as waveforms. Here we see that at clock tick A that q is TRUE, r is TRUE one tick later at tick B, and s is TRUE 3 ticks after that at clock tick E. The assert statement on line 7 would follow this sequence and find that the property is successfully matched. If a pass_Statement was provided, it would be executed. Since none is specified, the assertion succeeds quietly.

The above discussion follows what is labeled "Assertion Thread 1" in the figure. The way to think about how this works is that when the assertion sees the first variable (q) TRUE, it sets off an *assertion thread* that follows the rest of the assertion. That thread executes in the observed region of the simulation kernel and eventually either succeeds or fails.

Assertion Thread 2 is also shown in the figure. When the assertion sees the start of the second sequence (at tick C), it sets off another assertion thread to follow it. These two threads overlap in time between ticks C and E but they are separate; each thread will either succeed or fail on its own.

Assertion Thread 3 starts up at clock tick F but fails at tick G where it expects to see r TRUE.

Note what the assertion is checking for in each of these cases: only that q is TRUE now, r at the next tick, and s three ticks later. It is not checking that r is FALSE at the second tick after q, or that s is FALSE two ticks after r. As illustrated in Figure 9.1, r being TRUE at clock ticks E and H have no bearing on any of the assertion threads; these values are completely ignored.

A new semantic was introduced here, the ##n clock *cycle delay operator*. Instead of delaying by a specific amount of time as would happen with a #n delay, ##n specifies the delay in terms of the number of clock ticks to delay. In a description such as Example 9.2, the clock and its active edge are inferred by the @(posedge ck) statement on line 5. Thus it would delay for n positive clock edges on ck.

The ##n cycle delay operator can also be used in procedural blocks as shown in Example 9.3. You might consider using it in program blocks instead of "@(posedge ck);" as shown there. When using the cycle delay operator in an assertion, the clock is easily inferred. The same is not typically the case for procedural blocks, so you need to specify the clock to be used. This is done with the default clocking statement on lines 4-6. This names what is called the clocking block (cName) and, in this case, specifies the clock variable and active edge on line 5. On lines 17-20, the clock ck is initialized to 0; positive edges will occur on the fives (5, 15, 25, 35,

```
1   program cycleDelay;
2     bit ck, q;
3
4     default clocking cName
5       @(posedge ck);
6     endclocking
7
8     initial begin
9       q <= 0;
10      ##2 q <= 1;
11      ##3;
12      ##1 q <= 0;
13      ##1;
14      $finish;
15    end
16
17    initial begin
18      ck = 0;
19      forever #5 ck = ~ck;
20    end
21  endprogram: cycleDelay
```

Example 9.3 — Using Cycle Delay Operators in Procedural Code