The constants n and m can be any positive integer (including 0). m can also be the symbol "$" meaning *unbounded but finite;* sometimes read as *eventually.* $ means that the range should be satisfied by the end of simulation, no matter how far in the future that might be. If it isn't, then the assertion fails as the simulation ends.

But the design situation here is that there are really two operations that can occur and, in this case, have different possible delay times. To capture this situation, an *or* connective can be used in the sequence to specify either of the read or write conditions as shown in the following property:

```
1  property ckReadOrWrite;
2    @(negedge ck) start |=> ((read ##[1:9] dataValid) or
3                             (~read ##[1:6] dataValid));
4  endproperty
```

In this case, two sequences are specified and separated by the or operator. The first sequence to succeed causes the whole property to succeed. If neither succeeds, then the property fails.

Another aspect of the protocol that could be checked regards the start signal. The start signal is important in that it tells all the interfaces on the bus that a bus transaction (read or write) is starting. In this protocol, there can only be one transaction at a time. To check for this we need to make sure that once a transaction starts (as signaled by start), start should be FALSE until the clock tick after dataValid is TRUE. This requires checking the start signal at every clock tick. But this can be specified more efficiently with the *throughout* operator.

```
1  property ckOnlyOneStart;
2    @(negedge ck) start |=> ~start throughout
3                  ((read ##[1:9] dataValid) or
4                  (~read ##[1:6] dataValid));
5  endproperty
```

The *throughout* operator has an expression on the left that must remain TRUE throughout the sequence specified to its right. This sequence states that when start is TRUE, then at the next clock tick start will be FALSE (~start is TRUE) and it will remain FALSE until dataValid is seen in either of the read or write cycle conditions. In this way, it checks for start being TRUE for only one clock tick, and not again until the tick after dataValid is TRUE; the ending condition here is dataValid AND ~start. It could have been written more simply as:

```
1  property ckOnlyOneStartAlt;
2    @(negedge ck) start |=> ~start throughout ##[1:9] (dataValid & ~start);
3  endproperty
```

This approach only looks for dataValid regardless of whether it's a read or write. Thus, it won't detect if a write transaction takes longer than 7 clock ticks; it's just checking the start signal's value over the longest bus transaction.

Besides the sequence operators throughout and or, there are several others. Here is a full list: