```
set   Cons  = {cons[1..Ncons]} // consumer threads

const Nthread  = Nprod + Ncons
set   Threads  = {Prod,Cons}
```

The producer and consumer processes are composed with the processes modeling the buffer monitor by:

```
||ProdCons = (Prod:PRODUCER || Cons:CONSUMER
          || BUFFERMON).
```

13.4.3 Analysis

To verify our implementation model of the bounded buffer, we need to show that it satisfies the same safety and progress properties as the design model. However, the bounded buffer design model was specified in Chapter 5, which preceded the discussion of how to specify properties. Consequently, we simply inspected the *LTS* graph for the model to see that it had the required synchronization behavior. The *LTS* of the implementation model is much too large to verify by inspection. How then do we proceed? The answer with respect to safety is to use the design model itself as a safety property and check that the implementation satisfies this property. In other words, we check that the implementation cannot produce any executions that are not specified by the design. Clearly, this is with respect to actions that are common to the implementation and design models – the put and get actions. The property below is the same BUFFER process shown in Figure 5.11, with the addition of a relabeling part that takes account of multiple producer and consumer processes.

```
property
    BUFFER = COUNT[0],
    COUNT[i:0..Size]
        = (when (i<Size) put->COUNT[i+1]
          |when (i>0)    get->COUNT[i-1]
          )/{Prod.put/put,Cons.get/get}.
```

The *LTS* for this property with two producer processes, two consumer processes and a buffer with two slots (Size = 2) is shown in Figure 13.7.

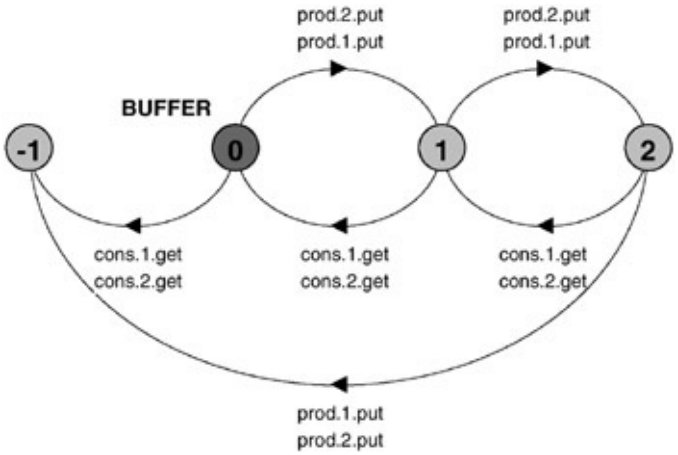

Figure 13.7: LTS for property BUFFER.

We are now in a position to perform a safety analysis of the bounded buffer implementation model using the composition:

||ProdConsSafety = (ProdCons || BUFFER).

With two producer processes (Nprod=2), two consumer processes (Ncons=2) and a buffer with two slots (Size=2), safety analysis by the *LTSA* reveals no property violations or deadlocks. In this situation, the implementation satisfies the design. However, safety analysis with two producer processes (Nprod=2), two consumer processes (Ncons=2) and a buffer with only one slot (Size=1) reveals the following deadlock:

Trace to DEADLOCK:

```
 cons.1.acquire
 cons.1.count.read.0
 cons.1.wait              // consumer 1 blocked
 cons.1.release
 cons.2.acquire
 cons.2.count.read.0
 cons.2.wait              // consumer 2 blocked
 cons.2.release
 prod.1.acquire
 prod.1.count.read.0
 prod.1.put              // producer 1 inserts item
 prod.1.count.inc
 prod.1.notify            // producer 1 notifies item available
 prod.1.release
```

```
prod.1.acquire
prod.1.count.read.1
cons.1.unblock          // consumer 1 unblocked by notify
prod.1.wait             // producer 1 blocks trying to insert 2nd item
prod.1.release
prod.2.acquire
prod.2.count.read.1
prod.2.wait             // producer 2 blocks trying to insert item
prod.2.release
cons.1.endwait
cons.1.acquire
cons.1.count.read.1
cons.1.get              // consumer 1 gets item
cons.1.count.dec
cons.1.notify           // consumer 1 notifies space available
cons.1.release
cons.1.acquire
cons.1.count.read.0
cons.2.unblock          // consumer 2 unblocked by notify
cons.1.wait
cons.1.release
cons.2.endwait
cons.2.acquire
cons.2.count.read.0
cons.2.wait             // consumer 2 blocks since buffer is empty
cons.2.release
```

The deadlock occurs because at the point that the consumer process calls notify to indicate that a space is available in the buffer, the wait set includes the second consumer process as well as both the producer processes. The consumer is unblocked and finds that the buffer is empty and goes back to waiting. At this point no further progress can be made and the system deadlocks since neither of the producer processes can run. This deadlock occurs if either the number of producer processes or the number of consumer processes is greater than the number of slots in the buffer. Clearly in this situation, the implementation given in the first printing of Chapter 5 was incorrect!

To correct the bounded buffer program of Chapter 5, to handle the situation of a greater number of producer or consumer threads than buffer slots, we need to replace the calls to notify() with calls to notifyAll(). This unblocks both consumer and the producer threads, allowing an insertion or removal to occur. Replacing the corresponding actions in the implementation model removes the deadlock and verifies that the Java program is now correct.

The lesson here is that it is always safer to use notifyAll() unless it can be rigorously shown that notify() works correctly. We should have followed our own advice in Chapter 5! The general rule is that notify() should only be used if at most one thread can benefit from the change of state being signaled and it can be guaranteed that the notification will go to a thread that is waiting for that particular state change. An implementation model is a good way of doing this.

The corrected model satisfies the following progress properties, which assert lack of starvation for put and get actions:

progress PUT[i:1..Nprod] = {prod[i].put}
progress GET[i:1..Ncons] = {cons[i].get}