$ Program Verification

In the book, we take a modeling approach to the design of concurrent programs. Models are constructed, so that we can focus on actions, interaction and concurrency before proceeding to implementation and adding the details concerned with data representation, resource usage and user interface. We use the model as a basis for program construction by identifying a mapping from model processes to Java threads and monitor objects. However, we do not demonstrate other than by testing and observation that the behavior of the implementation corresponds to the behavior predicted by the model. Essentially, we rely on a systematic translation of the model into Java to ensure that the program satisfies the same safety and progress properties as the model.

In this supplement, we address the problem of verifying implementations, using *FSP* and its supporting *LTSA* tool. In doing verification, we translate Java programs into a set of *FSP* processes and show that the resulting *FSP* model satisfies the same safety and progress properties that the design model satisfied. In this way, we can show that the program is a satisfactory implementation of its design model.

Chapter 4 of the book took exactly this approach of translating a Java program into an FSP model to investigate the problem of interference. To perform verification, we need to model the Java program at the level of variables, monitor locks and condition synchronization. In Chapter 4, we showed how to model variables and monitor locks. This supplement develops a model for condition synchronization so that we can verify Java programs that use `wait()`, `notify()` and `notifyAll()`. This model is used in verifying the Bounded Buffer and Readers - Writers Java programs from Chapter 5 and 6.

$.1 Modeling condition synchronization

In chapter 5, we outlined how a guarded action in a design model can be translated into a synchronized method as shown below:

```
FSP:    when cond act -> NEWSTAT
```

```
Java:  public synchronized void act()
               throws InterruptedException
       {
         while (!cond) wait();
         // modify monitor data
         notifyAll()
       }
```

We noted that if an action modifies the data of the monitor, it can call `notifyAll()` to awaken all other threads that may be waiting for a particular condition to hold with respect to this data. We also noted that if it is not certain that only a single thread needs to be awakened, it is safer to call `notifyAll()` than `notify()` to make sure that threads are not kept waiting unnecessarily.

$.1.1 wait, notify & notifyAll

To verify that a program using this translation satisfies the required safety and progress properties, we must model the behavior of `wait()`, `notify()` and `notifyAll()`. We can then rigorously check the use of `notify()` versus `notifyAll()`. A model for the interaction of these methods is developed in the following. Firstly, we define a process `ELEMENT` to manage the blocked state of each thread that accesses a monitor.

```
ELEMENT
  = (wait -> BLOCKED | unblockAll -> ELEMENT),
BLOCKED
  = ({unblock,unblockAll} -> UNBLOCK),
UNBLOCK
  = (endwait -> ELEMENT).
```

The `wait` action, representing a call to `wait()`, puts the process into the `BLOCKED` state. Then either an `unblock` action caused by a `notify()` or an `unblockAll` action caused by a `notifyAll()` causes the process to move to `UNBLOCK` and signal the return of the `wait()` method by the `endwait` action. We will deal with the way that the monitor lock is released and acquired by `wait()` later.

The `CONTROL` process manages how `notify` and `notifyAll` actions, representing the eponymous Java methods, cause `unblock` and `unblockAll` actions:

```
CONTROL = EMPTY,
EMPTY   = (wait -> WAIT[1]
          |{notifyAll,notify} -> EMPTY
          ),
WAIT[i:1..Nthread]
        = (when (i<Nthread)  wait -> WAIT[i+1]
          |notifyAll -> unblockAll -> EMPTY
          |notify -> unblock ->
           if (i==1) then EMPTY else WAIT[i-1]
          ).
```

Since we can only check systems with a finite set of states, we must define a static set of identifiers `Threads` to represent the set of threads that can potentially access a monitor object. The cardinality of this set is defined by the constant `Nthread`. The `CONTROL` process maintains a count of the number of processes in the blocked state. If there are no blocked processes then `notify` and `notifyAll` have no effect. If there are many blocked processes then a `notify` action unblocks any one of them. The set of threads waiting on a monitor and the effect of the `wait()`, `notify()` and `notifyAll()` methods is modeled by the composition:

```
const Nthread  = 3        //cardinality of Threads
set   Threads  = {a,b,c}  //set of thread indentifiers
set   SyncOps  = {notify,notifyAll,wait}

||WAITSET
      = (Threads:ELEMENT || Threads::CONTROL)
        /{unblockAll/Threads.unblockAll}.
```

The composition defines an `ELEMENT` process for each thread identifier in the set `Threads`. The `CONTROL` process is shared by all the threads in this set. The relabeling ensures that when any thread calls `notifyAll()` then all waiting threads are unblocked. The behavior of `WAITSET` is best illustrated using the animation facilities of LTSA.

Figure $.1 shows a trace in which thread **b** calls `wait`, then thread **a** calls `wait`. A call by thread **c** to `notify` unblocks thread **a**. Note that while thread **b** was blocked before **a**, it is **a** that is unblocked first. In other words, the model does not assume that blocked threads are held in a FIFO queue, although many Java Virtual Machines implement thread blocking this way. The Java Language Specification specifies only that blocked threads are held in a set and consequently may be unblocked in any order by a sequence of

notifications. An implementation that assumes FIFO blocking may not work correctly on a LIFO implementation. The `WAITSET` model permits all possible unblocking orders and consequently, when we use it in verification, it ensures that if an implementation model is correct, it is correct for all blocking/unblocking orders.

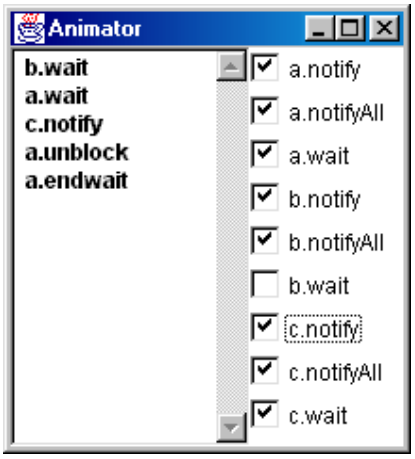

Figure $.1 – `WAITSET` trace for `notify`

Figure $.2 illustrates the behavior for `notifyAll` when threads **`a`** and **`b`** are blocked.

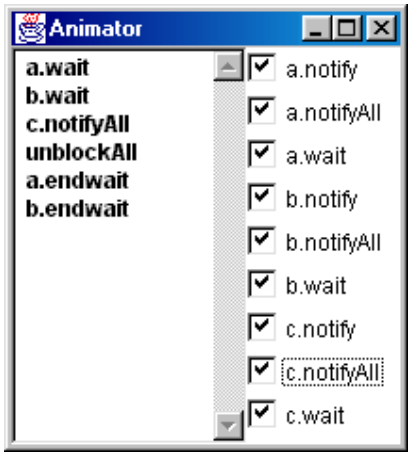

Figure $.2 – `WAITSET` trace for `notifyAll`

$.2 Modeling Variables & Synchronized methods

$.2.1 Variables

Variables are modeled in exactly the same way as presented in Chapter 4. However, for convenience, we add actions to model incrementing and decrementing integer variables. An integer variable is modeled by:

```
const Imax =    // maximum value that variable can take
range Int  = 0..Imax
set   VarAlpha = {read[Int],write[Int],inc,dec}

VAR         = VAR[0],
VAR[v:Int] = (read[v]      ->VAR[v]      //  v
             |inc          ->VAR[v+1]    //  ++v
             |dec          ->VAR[v-1]    //  --v
             |write[c:Int]->VAR[c]       //  v = c
             ).
```

A boolean variable is modeled by:

```
const False = 0
const True  = 1
range Bool  = False..True
set   BoolAlpha = {read[Bool],write[Bool]}

BOOLVAR          = BOOLVAR[False],
BOOLVAR[b:Bool] = (read[b]       ->BOOLVAR[b] // b
                  |write[c:Bool]->BOOLVAR[c] // b = c
                  ).
```

$.2.2 Monitor exit & entry

In Chapter 4, we noted that synchronized methods acquire the monitor lock before accessing the variables of a monitor object and release the lock on exit. We will use the same simple model of a lock used in chapter 4, ignoring the detail that locks in Java support recursive locking:

```
set LockOps = {acquire,release}
LOCK = (acquire -> release ->LOCK).
```

We can now model the state of a monitor by the set of processes that

represent its waitset, lock and variables. For example, the state for a monitor *M* that encapsulates a single boolean variable *cond* is modeled by:

```
||Mstate = (Threads::LOCK  || WAITSET
           || Threads::(cond:BOOLVAR)
           ).
```

In Java, the notification and waiting operations are only valid when the thread calling these operations holds the lock for the monitor object on which the operations are invoked. The following safety property checks that this is the case in the implementation models we construct:

```
property SAFEMON
   = ([a:Threads].acquire -> HELDBY[a]),
HELDBY[a:Threads]
   = ([a].{notify,notifyAll,wait} -> HELDBY[a]
     |[a].release -> SAFEMON
     ).
```

$.2.3 Synchronized Methods

A synchronized method of the form:

```
synchronized void act()throws InterruptedException {
     while (!cond) wait();
     // modify monitor data
     notifyAll()
}
```

can now be modeled by the following FSP fragment:

```
ACT   // act()
   = (acquire -> WHILE),     // monitor entry – acquire lock
WHILE
 = (cond.read[b:Bool] ->   // while (!cond) wait();
    if !b then
     (wait ->release ->endwait ->acquire -> WHILE)
    else
     CONTINUE
    ),
CONTINUE
   = (                      // modify monitor data
      notifyAll             // notifyAll()
       -> release           // monitor exit – release lock
      -> RETURN
      ),
```

Note that `wait()` is modeled by the sequence of actions: `wait ->release ->endwait ->acquire.` This reflects the way the monitor lock is release before a thread is blocked and acquired before it re-enters the monitor. Since FSP does not have a direct way of modeling method calls, we simply embed the fragment describing a synchronized method in the process modeling a thread that calls the synchronized method. This is analogous to way an optimizing compiler can substitute or *inline* the code for a method in place of the call to that method. This embedding should become clear when we look at models of threads in the next section. A process that models a synchronized method in this way must have its alphabet extended with the alphabet of monitor actions. In the example above, this would be the set `{SyncOps, LockOps, cond.BoolAlpha}`.

Lastly, we should note that with the above constructions, while we can now model monitors in some detail, we are still ignoring the effect of `InterruptException` occurrence and handling. In the book, we have only used this mechanism to terminate all the threads that constitute the concurrent Java program. We will discuss at the end of this supplement some of the problems that can occur if only a subset of threads are terminated in this way.

$.3 Bounded Buffer example

Program $.1 below reproduces the bounded buffer implementation from Chapter 5, Program 5.6. In this section, we develop a detailed model of the synchronization of this program and investigate its properties. As usual, we abstract from the details of what items are stored in the buffer and how these items are stored. Consequently, the only variable that we need to consider modeling is the variable count which stores the number of items currently stored in the buffer. The state of the buffer monitor implementation is thus modeled by:

```
||BUFFERMON  = ( Threads::LOCK || WAITSET || SAFEMON
                ||Threads::(count:VAR)
               ).
```

The set `Threads` is defined by:

```
const Nprod = 2                  // #producers
const Ncons = 2                  // #consumers
set   Prod  = {prod[1..Nprod]}   // producer threads
set   Cons  = {cons[1..Ncons]}   // consumer threads

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

The alphabet that must be added to each thread process is defined by:

```
set BufferAlpha = {count.VarAlpha, LockOps, SyncOps}
```

```
public interface Buffer {
  public void put(Object o)
     throws InterruptedException; //put object into buffer
  public Object get()
     throws InterruptedException; //get object from buffer
}

class BufferImpl implements Buffer {
  protected Object[] buf;
  protected int in = 0;
  protected int out= 0;
  protected int count= 0;
  protected int size;

  BufferImpl(int size) {
    this.size = size; buf = new Object[size];
  }

  public synchronized void put(Object o)
            throws InterruptedException {
    while (count==size) wait();
    buf[in] = o;
    ++count;
    in=(in+1)%size;
    notify();
  }

  public synchronized Object get()
            throws InterruptedException {
    while (count==0) wait();
    Object o =buf[out];
    buf[out]=null;
    --count;
    out=(out+1)%size;
    notify();
    return (o);
    }
}
```

Program $.1 - `Buffer` interface and `BufferImpl` class

$.3.1 Producer & Consumer Threads

To investigate the properties of the bounded buffer implementation, we model systems consisting of one or more producer threads and one or more consumer threads. The producer threads call `put()` and the consumer threads call `get()`. The process that models the implementation of the producer thread is listed below. The program statements concerned with putting an item into the array and incrementing the incrementing the input index are represented by the single action `put`. The constant Size defines the maximum number of items that may be stored in the buffer.

```
PRODUCER                              /* producer thread */
  = (put.call -> PUT),
PUT                                   /* inlined put method */
  = (acquire -> WHILE),
WHILE
  = (count.read[v:Int] ->     // while (count == size) wait();
     if v==Size then
       (wait ->release ->endwait ->acquire ->WHILE)
     else
       CONTINUE
    ),
CONTINUE
  = (put                        // buf[in] = o;  in=(in+1)%size;
     -> count.inc               // ++count;
     -> notify                  // notify()
     -> release
     -> RETURN
    ),
RETURN = PRODUCER + BufferAlpha.
```

The consumer thread is modeled below. Again, the statements concerned with the buffer array manipulation are represented by the single action `get`.

```
CONSUMER                          /* consumer thread */
  = (get.call -> GET),
GET                               /* inlined get method */
  = (acquire -> WHILE),
WHILE
  = (count.read[v:Int] ->         // while (count == 0 ) wait()
     if v==0 then
      (wait ->release ->endwait ->acquire ->WHILE)
     else
       CONTINUE
     ),
CONTINUE
  = (get                  // Object[o] = buf[out]; buf[out] = null;
     -> count.dec         // --count;
     -> notify            // notify()
     -> release
     -> RETURN
     ),
RETURN = CONSUMER + BufferAlpha.
```

The entire system of producer and consumer threads, together with the buffer monitor is now modeled by:

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

$.3.2 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 $.1.

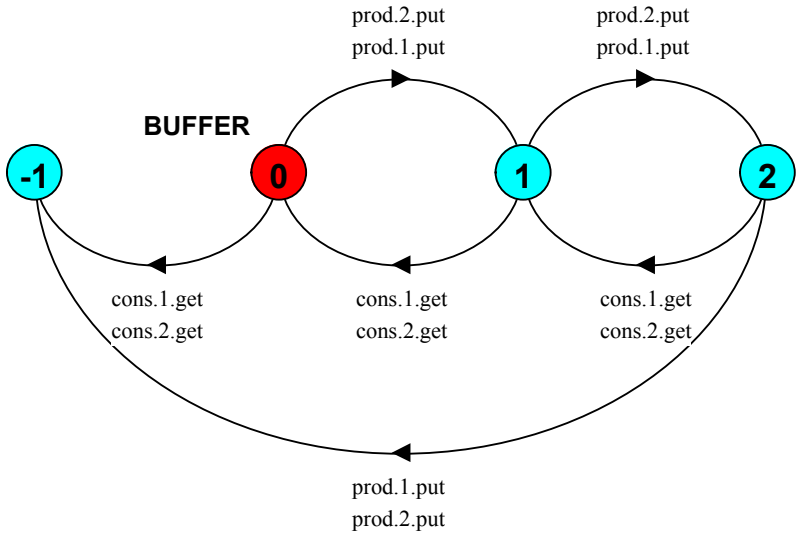

Figure $.1 – LTS for `property BUFFER`

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:
      prod.1.put.call
      prod.2.put.call
      cons.1.get.call
      cons.1.acquire
      cons.1.count.read.0
      cons.1.wait            // consumer 1 blocked
      cons.1.release
      cons.2.get.call
      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.put.call
      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.get.call
      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 Chapter 5 is incorrect!

To correct the bounded buffer program of chapter 5, to handle the situation of 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}
```

$.4 Readers-Writers example

Program $.2 reproduces the Readers-Writers program from Chapter 7, program 7.8. This is the version that gives Writers priority.

```
class ReadWritePriority implements ReadWrite{
  private int readers =0;
  private boolean writing = false;
  private int waitingW = 0; // no of waiting Writers.

  public synchronized void acquireRead()
              throws InterruptedException {
    while (writing || waitingW>0) wait();
     ++readers;
  }

  public synchronized void releaseRead() {
    --readers;
    if (readers==0) notify();
  }

  public synchronized void acquireWrite()
              throws InterruptedException {
    ++waitingW;
    while (readers>0 || writing) wait();
    --waitingW;
    writing = true;
  }

  public synchronized void releaseWrite() {
    writing = false;
    notifyAll();
  }
}
```

Program 7.17 – class ReadWritePriority

The `ReadWritePriority` monitor class has three variables – `readers`, `writing` and `waitingW` – which all play a part in synchronization. Consequently, in this example, we model the state of the monitor by:

```
||RWPRIORMON = ( Threads::LOCK || WAITSET || SAFEMON
                ||Threads::( readers:VAR
                             ||writing:BOOLVAR
                             ||waitingW:VAR
                            )
                ).
```

The set `Threads` is defined by:

```
const Nread  = 2                    // #readers
const Nwrite = 2                    // #writers
set   Read   = {reader[1..Nread]}   // reader threads
set   Write  = {writer[1..Nwrite]} // writer threads

const Nthread =  Nread + Nwrite
set   Threads = {Read,Write}
```

The alphabet that must be added to each reader and writer process in this example is defined by:

```
set ReadWriteAlpha =
  {{readers,waitingW}.VarAlpha, writing.BoolAlpha,
   LockOps, SyncOps,
   acquireRead,acquireWrite,releaseRead,releaseWrite
  }
```

$.4.1 Reader & Writer Threads

As before, we develop a model of each type of thread. Each reader thread repetitively calls `acquireRead()` followed by `releaseRead()`. Each writer thread repetitively calls `acquireWrite()` followed by `releaseWrite()`. These methods calls are modeled as described in section $.2.3 by embedding the FSP fragment that describes their behavior in the processes modeling threads. The process that models the implementation of the `Reader` thread is listed below:

```
READER                                        /* reader thread */
  = (acquireRead.call -> ACQUIREREAD),
ACQUIREREAD                          // inlined acquireRead method
  = (acquire -> WHILE),
WHILE                          // while (writing || waitingW>0) wait();
  = (writing.read[v:Bool] -> waitingW.read[u:Int] ->
     if (v || u>0) then
        (wait ->release ->endwait ->acquire ->WHILE)
     else
        CONTINUE
     ),
CONTINUE
  = (acquireRead
     -> readers.inc                  // ++readers
     -> release -> RELEASEREAD
     ),
RELEASEREAD                        // inlined releaseRead method
  = (releaseRead.call -> acquire -> releaseRead
    -> readers.dec                   //   --readers;
    -> readers.read[v:Int] ->        //   if (readers==0) notify();
    if (v==0) then
       (notify -> RETURN)
    else
       RETURN
    ),
 RETURN
   = (release -> READER) + ReadWriteAlpha.
```

The `Writer` thread implementation is modeled by:

```
WRITER                                  /* writer thread */
  = (acquireWrite.call -> ACQUIREWRITE),
ACQUIREWRITE              // inlined acquireWrite method
  = (acquire
    -> waitingW.inc -> WHILE      // ++waitingW;
    ),
WHILE                 // while (readers>0 || writing) wait();
  = (writing.read[b:Bool] -> readers.read[v:Int]->
     if (v>0 || b) then
        (wait ->release ->endwait ->acquire ->WHILE)
     else
        CONTINUE
     ),
CONTINUE
  = (acquireWrite
     -> waitingW.dec           // --waitingW;
     -> writing.write[True]    // writing = true;
     -> release -> RELEASEWRITE
    ),
RELEASEWRITE                   // inlined releaseWrite method
   = (releaseWrite.call -> acquire -> releaseWrite
     -> writing.write[False]   // writing = false;
     -> notifyAll              // notifyAll();
     -> release-> WRITER
     ) + ReadWriteAlpha.
```

$.4.2 Analysis

To verify that the implementation model satisfies the desired safety properties, we use the safety property `RW_SAFE` originally specified in section 7.5.1 to check the correct behavior of the design model.

```
property SAFE_RW
  = (acquireRead -> READING[1]
    |acquireWrite->WRITING
    ),
READING[i:1..Nread]
  = (acquireRead -> READING[i+1]
    |when(i>1)  releaseRead -> READING[i-1]
    |when(i==1) releaseRead -> SAFE_RW
    ),
WRITING  = (releaseWrite -> SAFE_RW).
```

The system we perform safety analysis on consists of the reader and writer threads, the monitor state and the safety property as shown below:

```
||RWSYS = ( Read:READER || Write:WRITER
          ||RWPRIORMON
          ||Threads::SAFE_RW
          ).
```

Safety analysis detects the following deadlock:

```
Trace to DEADLOCK:
      reader.1.acquireRead.call
      reader.1.acquire
      reader.1.writing.read.0
      reader.1.waitingW.read.0
      reader.1.acquireRead
      reader.1.readers.inc
      reader.1.release          // reader 1 acquires RW lock
      reader.1.releaseRead.call
      reader.2.acquireRead.call
      writer.1.acquireWrite.call
      writer.1.acquire
      writer.1.waitingW.inc
      writer.1.writing.read.0
      writer.1.readers.read.1
```

```
writer.1.wait   // writer 1 blocked as reader has RW lock
writer.1.release
reader.2.acquire
reader.2.writing.read.0
reader.2.waitingW.read.1
reader.2.wait   // reader 2 blocked as writer 1 waiting
reader.2.release
writer.2.acquireWrite.call
writer.2.acquire
writer.2.waitingW.inc
writer.2.writing.read.0
writer.2.readers.read.1
writer.2.wait // writer 2 blocked as reader has RW lock
writer.2.release
reader.1.acquire
reader.1.releaseRead
reader.1.readers.dec
reader.1.readers.read.0
reader.1.notify // reader 1 releases RW lock & notifies
writer.2.release
reader.1.release
reader.1.acquireRead.call
reader.1.acquire
reader.1.writing.read.0
reader.1.waitingW.read.2
reader.2.unblock // reader 2 unblocked by notify
reader.1.wait
reader.1.release
reader.2.endwait
reader.2.acquire
reader.2.writing.read.0
reader.2.waitingW.read.2
reader.2.wait   // reader 2 blocks as writers waiting
reader.2.release
```

The deadlock happens because the `notify` operation performed by Reader 1 when it releases the read-write lock unblocks another Reader rather than a Writer. This unblocked Reader subsequently blocks again since there are Writers waiting. The solution is again to use a `notifyAll` to awake all waiting threads and thus permit a Writer to run. Changing the `notify` action to `notifyAll` in the `RELEASEREAD` part of the model and rerunning the safety analysis conforms that the deadlock does not occur and that the

implementation model satisfies the safety property.

Why did we not observe this deadlock in the actual implementation of `ReadWritePriority` when running the demonstration applet? The reason is quite subtle. In most Java Virtual Machines, and in particular in the JVM distributed on the CDROM, the set of threads waiting on notification is implemented as a first-in-first-out (FIFO) queue. With this queuing discipline, the deadlock cannot occur as for the second Reader to block, a Writer must have previously blocked. This Writer will be unblocked by the notification when the first Reader releases the read-write lock and consequently, the deadlock does not occur. However, although the implementation works for some JVMs, it is not guaranteed to work on all JVMs since as noted earlier, the Java Language Specification specifies only that blocked threads are held in a set. Our implementation would exhibit the deadlock on a JVM that used a stack for the wait set. Consequently, the implementation is clearly erroneous and the `notify()` in the `releaseRead()` method should be replaced with `notifyAll()`. Again the lesson is that `notify()` should only be used with extreme care! However, it should be noted that the use of `notify()` in the `ReadWriteSafe` version of the read-write lock is correct since it is not possible in that implementation to have both readers and writer waiting simultaneously.

Progess Analysis

Having demonstrated that the implementation model satisfies the required safety properties, it now remains to show that it exhibits the same progress properties as the design model. These properties assert lack of starvation for `acquireRead` and `acquireWrite` actions.

```
progress WRITE[i:1..Nwrite] = writer[i].acquireWrite
progress READ [i:1..Nwrite] = reader[i].acquireRead
```

The adverse scheduling conditions needed to check progress in the presence of competition for the read-write lock are arranged by making the actions, representing calls to release read an write access to the lock, low priority:

```
||RWPROGTEST
  = RWSYS >> {Read.releaseRead.call,
              Write.releaseWrite.call}.
```

Progress analysis reveals that the `RWPROGTEST` system satisfies the `WRITE` progress properties but violates the `READ` progress properties. In other words,

the Writers priority implementation of the read-write lock satisfies its design goal of avoiding Writer starvation, but as with the design model, it permits Reader starvation.

Summary

This supplement has presented a way of verifying that Java implementations satisfy the same safety and progress properties as the design models from which they were developed. The approach is to translate the Java program into a detailed FSP model that captures all aspects of the Java synchronization mechanisms – in particular, monitor locks and notification. This *implementation* model is then analyzed with respect to the same safety and progress properties used in analyzing the design model. We also showed that in the bounded buffer example that the design model itself can be used as a safety property when verifying the implementation model. The current version of the LTSA tools only permits deterministic primitive processes to be used as safety properties. However, the next release of the tool will permit any composite process to be used as a safety property.

Implementation models are considerably more detailed than design models and as such generate much larger state spaces during analysis. It is in general only possible to analyze small parts of an implementation. This is why in the book we have advocated a model-based design approach in which properties are investigated with respect to a design model and then this model is used as a basis for program implementation. Clearly, as we have demonstrated in the examples contained in this supplement, errors can be introduced in going from design model to implementation. Interestingly, the two bugs discovered both arise from the use of `notify()` in place of `notifyAll()`. Perhaps the most important lesson from this supplement is that strict attention must be paid to the rule 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 in the monitor class itself or in any subclasses.

Notes

We are indebted to David Holmes of Microsoft Research Institute, Macquarie University, Sydney, Australia for initially pointing out the problems with the bounded buffer and read-write lock that we have exposed in this chapter. He also motivated this supplement by suggesting that we should address the problem of verifying implementations.

We pointed out in the supplement that our model of notification ignores the effect of an interrupt exception. It is possible, in the current versions of JDK, for a waiting thread to be notified, but to be interrupted before actually returning from the `wait()` call. As a result it returns via an `InterruptedException` not a normal return and essentially, the notification is lost even though other uninterrupted threads may be waiting. This means that programs that use `notify()` and allow `InterruptedException` to be thrown directly are not guaranteed to work correctly. This is a bug in JDK that hopefully will be resolved in future versions. Although we use this technique in the book, it does not result in inconsistencies since in all cases, the interrupt exception is used to terminate all active threads. However, it is another reason for using `notifyAll()` rather than `notify()`. However, this may sometimes result in a large number of unnecessary thread activations and consequently be inefficient. For example, in the semaphore program of section 5.2.2, a better way to deal with the lost notification problem is to catch the `InterruptedException` and perform an additional `notify()` before rethrowing the exception.. Thanks again to David for pointing this out.