6SENG002W Concurrent Programming

Lecture 11

Translating FSP Models into Concurrent Java Programs

Translating FSP Models into Concurrent Java Programs

The aim of this lecture is to illustrate how to translate FSP into Java.

We shall do this by considering:

- The following FSP program examples:
  - the *Ornamental Garden Problem* – involving processes interacting & sharing an object using *mutual exclusion*.
  - a *Semaphore* – involving *conditional synchronisation*.
- The issues related to Threads *interacting* & *sharing objects using Mutual Exclusion*.
- How to translate various aspects of FSP processes & programs into Java threads, monitors & programs:
  - an *FSP process* into either a Java *monitor* or *thread*,
  - an *FSP composite process* (program) into a Java *main program*.

Note: these lecture notes are based on material in Chapters 4, 5, & 7 of the recommended book: *Concurrency: State Models & Java Programs, (2nd Edition)*, J. Magee & J. Kramer, Wiley, 2006. (ISBN 978-0-470-09355-9)

PART I

The Ornamental Garden Problem an Example of Process Interaction & Mutual Exclusion

The Ornamental Garden Problem

To illustrate the issues of *thread interaction & mutual exclusion*, we use the *Ornamental Garden problem*. (See the recommended book Chapter 4.)

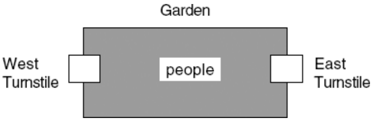

Figure : 11.1 Ornamental Garden

- People visit an ornamental garden, by entering through either of *two turnstiles* – East & West.
- To simplify the problem people are allowed to *enter* but **never leave**.
- Management needs to determine how many people are in the garden at any one time, so a single *people counter* is required.
- So when a person passes through either turnstile, the turnstile must increment the people counter, therefore the counter must be **shared**.
- The FSP program used to model this system must represent the:
  - *shared people counter*, that requires mutually exclusive access,
  - *East & West turnstiles* that have *shared ME access* to the people counter.

FSP Model of the Ornamental Garden

In general *each object* or *set of objects* will be **modelled** as an *FSP process*.

The Ornamental Garden FSP program models the *increment* action by incrementing a variable counter process `SAFE_COUNTER`, that describes the read & write accesses to a variable.

There is **no explicit mention** of an *increment* action, instead, increment is modelled using `read` & `write` actions by `TURNSTILE`'s local process `INCREMENT`.

The **`east`** & **`west`** `TURNSTILE` processes, each has its own copy of the `read` & `write` actions that make up the increment operation.

In other words, the increment operation is **not an atomic action**.

Therefore, *interference*, i.e. destructive updates, can occur by means of the arbitrary interleaving of `read` & `write` actions.

The *solution to interference* is to give operations that access a shared object *mutually exclusive* access to that object.

This ensures that an update, i.e. an increment (`read` then `write`), is **not interrupted** by concurrent updates.

Ornamental Garden: Preliminary Definitions

We use the following preliminary definitions of: a *constant*, a *number range*, an *action set* & 2 *process label sets*.

These will be used in the definition of the FSP processes that model the Ornamental Garden problem.

Ornamental Garden: Definitions

```
const  N = 4           /* Max value for counter */

range CR = 0..N        /* Range for counter */

/* Alphabet of SAFE_COUNTER process */
set CounterAlpha = { counter.{ read[CR], write[CR],
                               acquire,  release   } }

/* Process labels for the active processes */
set Turnstiles   = { east, west }

set CounterUsers = { east, west, management }
```

CounterAlpha is the counter process's alphabet & is used to extend the alphabets of the two turnstiles **east** & **west**.

Ornamental Garden: an "Unsafe" Shareable Counter

We begin by defining an FSP process `UNSAFE_COUNTER` that models a *counter variable* & the *basic operations* on the *counter*.

This process **DOES NOT** attempt to enforce *mutual exclusion*, this will be added in the next stage.

Ornamental Garden: UNSAFE_COUNTER

```
UNSAFE_COUNTER      = US_COUNT[ 0 ] ,

US_COUNT[ v : CR ] = (  read[ v ]          -> US_COUNT[ v ]
                      | write[ nv : CR ] -> US_COUNT[ nv ] ) .
```

Where:

- the counter is *initialised* to 0 by `US_COUNT[ 0 ]`,
- the `read[ v ]` action is used to *read/get* the current value of the counter `v`,
- the `write[ nv : CR ]` action is used to *write/set* the new value of the counter to `nv`.

Ornamental Garden: Expanded UNSAFE_COUNTER

To understand how this process UNSAFE_COUNTER models a counter here is the expanded version:

Expanded: UNSAFE_COUNTER

```
UNSAFE_COUNTER      = US_COUNT[ 0 ] ,

US_COUNT[ 0 ] = (   read[ 0 ]           -> US_COUNT[ 0 ]
                  | write[ nv : CR ] -> US_COUNT[ nv ] ) ,

US_COUNT[ 1 ] = (   read[ 1 ]           -> US_COUNT[ 1 ]
                  | write[ nv : CR ] -> US_COUNT[ nv ] ) ,

US_COUNT[ 2 ] = (   read[ 2 ]           -> US_COUNT[ 2 ]
                  | write[ nv : CR ] -> US_COUNT[ nv ] ) ,

US_COUNT[ 3 ] = (   read[ 3 ]           -> US_COUNT[ 3 ]
                  | write[ nv : CR ] -> US_COUNT[ nv ] ) ,

US_COUNT[ 4 ] = (   read[ 4 ]           -> US_COUNT[ 4 ]
                  | write[ nv : CR ] -> US_COUNT[ nv ] ) .
```

So the local process US_COUNT[0] represents a *counter* with the value 0, similarly for the others.

Ornamental Garden: Expanded UNSAFE_COUNTER's US_COUNT[0]

We can further expand UNSAFE_COUNTER by expanding its local process US_COUNT[0], the expanded version:

Expanded: US_COUNT[0]

```
US_COUNT[ 0 ] = (  read[ 0 ]  -> US_COUNT[ 0 ]
                 | write[ 0 ] -> US_COUNT[ 0 ]
                 | write[ 1 ] -> US_COUNT[ 1 ]
                 | write[ 2 ] -> US_COUNT[ 2 ]
                 | write[ 3 ] -> US_COUNT[ 3 ]
                 | write[ 4 ] -> US_COUNT[ 4 ] )
```

US_COUNT[0] represents a *counter* with the value 0, its value can be read & it can be set to any value 0 – 4 by write.

The other local processes: US_COUNT[1] – US_COUNT[4], can be expanded similarly.

Ornamental Garden: a “Safe” Shareable Counter

To ensure that the UNSAFE_COUNTER counter can be *“safely shared”* access to it by the 2 turnstiles must be *mutually exclusive*.

This is achieved by combining it with a *locking* process.

A *lock* can be modelled by the LOCK process & then combined with the UNSAFE_COUNTER process by the composition SAFE_COUNTER:

Ornamental Garden: VISITORS_COUNTER

```
LOCK = ( acquire -> release -> LOCK ) .

|| SAFE_COUNTER = ( LOCK || UNSAFE_COUNTER ) .

|| VISITORS_COUNTER = ( counter : SAFE_COUNTER ) .
```

We use the *“safe”* VISITORS_COUNTER version in the complete system, rather than the *“unprotected”* & *“unsafe”* UNSAFE_COUNTER version.

This is just SAFE_COUNTER labelled with “**counter**”, which in effect collects the actions together & defines a *“counter interface”*.

Ornamental Garden: Defining a `TURNSTILE` Process

Finally, the definition of the `TURNSTILE` process involves three steps.

Step 1: Communication between a `TURNSTILE` & the `VISITORS_COUNTER`

The 2 processes communicate using *inter-process communication* (IPC), by *synchronising* the two actions `read[v]` & `write[nv]`:

1. The `TURNSTILE` process *"reads"* the counter's value from the `VISITORS_COUNTER` by them performing a *synchronised* `read[v]` action:
   - `VISITORS_COUNTER` *outputs* `v`, the *current value of the counter*
   - `TURNSTILE` *inputs* the value of `v`.
2. The `TURNSTILE` process *"writes"* the counter's new value to the `VISITORS_COUNTER` by them performing a *synchronised* `write[nv]` action:
   - `TURNSTILE` *outputs* `nv`, the *new value of the counter*,
   - `VISITORS_COUNTER` *inputs* `nv`.

Defining a TURNSTILE Process: Step 2

Step 2: `TURNSTILE` **uses Mutual Exclusion Protocol to Access the** `VISITORS_COUNTER`**'s value**

Each `TURNSTILE` process **must be required to adhere** to the *mutual exclusion protocol* (MEP):

```
MEP = lock -> ``use resource'' -> unlock
```

We implement this in FSP by requiring the `TURNSTILE` process to:

1. **`acquire`** the lock before accessing the **`counter`** variable,
2. update the the **`counter`** variable, by *synchronising* the two actions `read[v]` & `write[nv]`,
3. **`release`** the lock after it has finished updating the **`counter`** variable.

Defining a TURNSTILE Process: Step 3

Step 3: Block Resource Performing Asynchronous Actions

Ensuring Step 2 implements ME correctly, we need to make sure that the VISITORS_COUNTER **cannot do any unauthorised asynchronous actions**.

```
alphabet( VISITORS_COUNTER )  = CounterAlpha

  = { counter.{ read[CR], write[CR], acquire, release } }

  = { counter.{ read[0],  read[1],  read[2],  read[3],  read[4],
                write[0], write[1], write[2], write[3], write[4],
                acquire, release } }
```

So the actions that would cause a problem by being performed *asynchronous* by VISITORS_COUNTER are: write[0] – write[4].

This is because a turnstile will **only every attempt to write one of the values 0 – 4 at a time**, but VISITORS_COUNTER is **always willing to do all of them**. (See expanded US_COUNT[0] above.)

So VISITORS_COUNTER **must be stopped** from doing a write *asynchronously* that a turnstile **does not** want to perform.

This is prevented by requiring all actions of VISITORS_COUNTER to be *synchronised* with the 2 turnstile processes.

Ornamental Garden: TURNSTILE Processes

First define the single TURNSTILE process.

```
                         Ornamental Garden: TURNSTILE
TURNSTILE = ( open -> OPEN_TURNSTILE ) ,

OPEN_TURNSTILE = (  arrive -> INCREMENT
                  | close  -> END ) ,

INCREMENT
 = ( counter.acquire          ->  /* Lock counter */
       counter.read[ x : CR ] ->  /* Get counter value */
       counter.write[ x + 1 ] ->  /* Write new counter value */
     counter.release          ->  /* Unlock counter */
     OPEN_TURNSTILE
   ) +CounterAlpha .
```

Requiring **all actions** of VISITORS_COUNTER to be *synchronised* with a turnstile process is achieved by *extending the alphabet* of TURNSTILE's local process INCREMENT with the alphabet of VISITORS_COUNTER using **CounterAlpha**.

The result is that VISITORS_COUNTER can only do an action if the TURNSTILE is willing to do it as well, i.e. *synchronise* on it.

Ornamental Garden: East & West TURNSTILE Processes

We can now use this to define the two *East* & *West* TURNSTILE Processes:

```
Ornamental Garden: TURNSTILES

|| EAST_TURNSTILE = ( east : TURNSTILE ) .

|| WEST_TURNSTILE = ( west : TURNSTILE ) .

|| TURNSTILES = ( WEST_TURNSTILE || EAST_TURNSTILE )
                  / {  open / Turnstiles.open,
                      close / Turnstiles.close   }   .
```

```
alphabet( EAST_TURNSTILE )
 = { east.{ open, arrive, close,
            counter.acquire, counter.release,
            counter.read[CR], counter.write[CR]   }    }

 = { east.open, east.arrive,  east.close,
     east.counter.acquire,  east.counter.release,
     east.counter.read[CR], east.counter.write[CR]     }
```

Note that "`open / Turnstiles.open`" means relabel both **`east`**`.open` & **`west`**`.open` to `open`, similarly for `close`.

Ornamental Garden: MANAGEMENT process

The `MANAGEMENT` process can check the value of the counter at any time using the `DISPLAY` process to `read` the value of the **counter**, without having to lock & unlock it.

In this case this is acceptable since the `DISPLAY` process does not update the value of the **counter** variable.

The `close` action closes the garden & terminates the system.

```
Ornamental Garden: MANAGEMENT

DISPLAY   = (    counter.read[CR] -> DISPLAY
               | close            -> END      ) +CounterAlpha .

|| MANAGEMENT = ( management : DISPLAY )
                              / { close / management.close } .
```

As with `INCREMENT`, its alphabet must be *extended* with that of the `VISITORS_COUNTER`'s to ensure **no inappropriate actions** are performed by the `VISITORS_COUNTER`.

For example, any of the **counter**`.write[CR]` ones, e.g. **counter**`.write[0]`, **counter**`.write[1]`, etc.

Ornamental Garden: Complete FSP Model

The complete Ornamental Garden system is defined by the `GARDEN` process:

Ornamental Garden: GARDEN

```
|| GARDEN  =  (      TURNSTILES
                 ||  MANAGEMENT
                 ||  CounterUsers :: VISITORS_COUNTER
              ) .
```

which expands to:

Expanded GARDEN

```
|| GARDEN  =
   (
     (west  : TURNSTILE) /{open/west.open, close/west.close}
    ||
     (east  : TURNSTILE) /{open/east.open, close/east.close}
    ||
     ( management : DISPLAY )
    ||
     ( { east, west, management } :: counter : LOCK )
    ||
     ( { east, west, management } :: counter : SAFE_COUNTER )
   ) .
```

Notes on the Ornamental Garden FSP Model

The alphabet for the `TURNSTILE` process is extended with the **`CounterAlpha`** set using the alphabet extension construct **`+{...}`**.

This is to ensure that **no unintended asynchronous actions** are performed within the system.

For example, if a `VISITORS_COUNTER write[nv]` is **not shared** (synchronised) with another process, i.e. a `TURNSTILE` or `DISPLAY` then it can do it **asynchronous**.

`TURNSTILE` **never** does the action "**`counter`**`.write[0]`" since it *always increments the value* it reads.

But, since "**`counter`**`.write[0]`" is in **`CounterAlpha`**, `VISITORS_COUNTER` is **prevented from preforming it asynchronously**.

The `close` action is synchronised on by the turnstiles & the management processes it terminates the system, it is only accepted as an alternative to an `arrive` action.

Note that `VISITORS_COUNTER` is **shared by** the **`east`** & **`west`** `TURNSTILE`s & `MANAGEMENT`.

Ornamental Garden: Structure Diagram

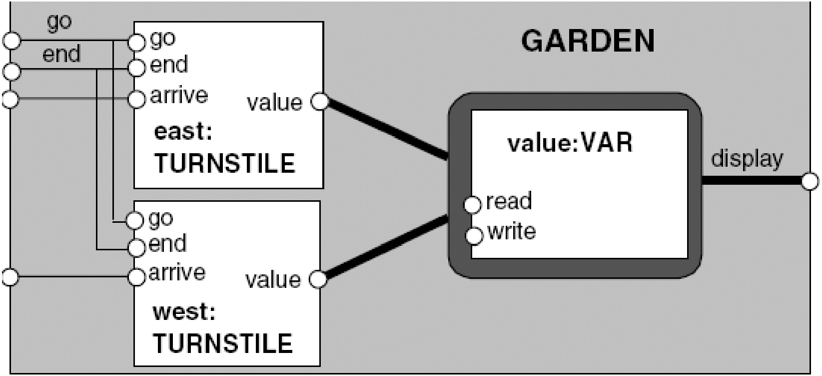

Figure : 11.2 Ornamental Garden Structure Diagram

Notes: in the diagram “value” is **`counter`** & “value:VAR” is VISITORS_COUNTER, i.e. **`counter`**:SAFE_COUNTER.

PART II

Threads Communicating/Interacting by Sharing an Object using Mutual Exclusion

Threads Communicating via a *Shared Object*

One of the simplest ways for two or more Java threads to *“communicate”* &/or *”interact”* with each other is via a *shared object*.

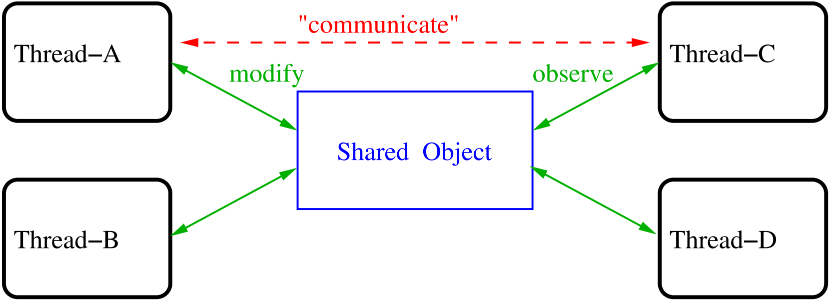

Figure : 11.3 Threads Communicate via a Shared Object

Mutual Exclusion: Context – *Shared Objects*

The *shared object* from Fig. 11.3, that is intended to be used in this way, must have an *interface* that allows this.

That is it must have `public` methods that allow its *state* to be *“modified”* & *“observed”* by these methods.

The collection of threads can then communicate & interact by invoking the shared object’s `public` methods.

Consequently, two threads can **communicate** by one thread *modifying* the state of the shared object & the other thread then *observing* the state.

For example, Thread–A & Thread–C.

Other examples of threads sharing an object, are when a collection of threads *cooperate to update* information encapsulated in a shared object.

Mutual Exclusion: Problem – ***Interference***

We have previously seen that the execution of the instructions from a collection of processes/threads can be interleaved in an arbitrary fashion.

This interleaving can result in incorrect updates to the state of a shared object, resulting in **corrupted data**, this is known as **interference**.

These types of objects are sometimes referred to as *damaged objects*.

This is clearly a situation that must be avoided at all costs.

Mutual Exclusion: Solution – *Java Monitors*

The problem of **interference**, is solved by the enforcement of *mutually exclusive* access to the shared object, by means of a *locking* arrangement.

In Java this means ensuring that the *shared object* is implemented as a **correctly functioning "secure" Java monitor**.

That is, the Java monitor ensures:

- *complete data encapsulation inside the monitor*

  — **all data** is either `private` or `protected`

- *execution of all its* `public` *methods are mutually exclusive*

  — **all** `public` **methods** are `synchronized`.

PART III

Translating FSP into Java

Translating FSP into Java

An FSP program, i.e. a collection of FSP processes, is used to model a program.

In a given FSP model of a system each:

Shared object: is modelled as an *FSP process*,

Process: is also modelled as an *FSP process*.

However, FSP processes (that make up a program), do not distinguish between:

- *“Passive” entities* (shared objects: Java monitors).
- *“Active” entities* (processes: Java threads).

They are both modelled as finite state machines.

This uniform treatment facilitates analysis.

Therefore, in translating an FSP program into a Java program *we must decide* what kind of Java *“object”* each individual FSP process is modelling.

Mutual Exclusion in Java

In Java usually represent *"shared objects"* as *Java monitors*.

At the Java monitor level of abstraction, we can ignore the details of *locks* & *mutual exclusion* (as provided by the use of `synchronized` methods).

Concurrent calls of a Java `synchronized` method are *mutually exclusive*.

So we can translate the FSP `SAFE_COUNTER` process into a Java monitor & make the `increment` method **`synchronized`**.

```
class SAFE_COUNTER          // a ``monitor''
{
  private int value = 0 ;

  SAFE_COUNTER( int initial ){ value = initial ;  }

  public synchronized void increment()
  {
     int temp = value ;     // "read[ x : CR ]"
     value    = temp + 1 ;  // "write[ x + 1]"
  }
}

SAFE_COUNTER counter = new SAFE_COUNTER( 0 ) ;  // "US_COUNT[ 0 ]"
```

Java Monitor Semantics Ensures Mutual Exclusion

Java associates a *lock* with every object.

The Java compiler inserts code to *acquire the lock* before executing the body of a `synchronized` method & code to *release the lock* before the method returns.

Concurrent threads *trying to access the monitor* are **blocked** until the lock is *released*.

Since only one thread at a time may hold the lock, only one thread may be executing the `synchronized` method.

If this is the only method, as in the example, *mutual exclusion* to the shared object is ensured.

If an object has *more than one method*, to ensure mutually exclusive access to the state of the object, **all the methods should be `synchronized`**.

Access to an object may also be made *mutually exclusive* by using the `synchronized` statement, but this is **not recommended**, as it is too *"unstructured"* & prone to errors.

Summary

We have discussed *thread interaction* via *shared objects*.

The Ornamental Garden example served to demonstrate that uncontrolled interleaving of method instructions leads to **interference**, i.e. **destructive update** of the state of the shared object.

Interference can be avoided by giving each concurrent method activation *mutually exclusive access to the shared state*.

In Java, this is achieved by making such methods `synchronized`.

`synchronized` methods **acquire a lock** associated with the object before accessing the object state & **release the lock** after access.

Since only one thread at a time can acquire the lock, `synchronized` methods obtain *mutually exclusive access* to the object state.

The answer is to: **ensure that all the methods of objects shared between threads are `synchronized`**.

They can then be treated as *"atomic actions"* for modelling purposes.

Note that **interference bugs** in real concurrent programs are notoriously difficult to find.

Condition Synchronisation: Semaphores

We illustrate *condition synchronisation* using a simple version of Dijkstra's semaphores as an example.

Recall that a semaphore `s` is an integer variable that can take only non-negative values.

Once `s` has been given an initial value, the only operations permitted on `s` are `s.claim()` & `s.release()` defined as follows:

```
s.claim(): when ( s > 0 ) { decrement s ; }
           when ( s == 0 ) { wait/suspend ; }

s.release(): increment s ;
```

In the following, we describe how *semaphores* can be modelled by an *FSP process* & how that can be implemented using a *Java monitor*.

However, in practice, semaphores are a low-level mechanism & are sometimes used to implement the higher-level monitor construct, rather than vice versa.

FSP Semaphore Model & Process

First step in modelling a system: decide which *events* or *actions* are of interest.

For a semaphore there are only two actions:

- **`claim`**
- **`release`**.

Second step: identify the *processes* – the *users* of the shared variable.

The control requirements for a semaphore's `claim` action are modelled using the FSP *guarded action construct*:

```
when ( B ) action
```

FSP Semaphore: SEMAPHORE

The FSP process to model this simple view of a semaphore is:

SEMAPHORE

```
const MAX_NAT = 3
range NAT     = 0..MAX_NAT

SEMAPHORE( N = 0 ) = SEMA[ N ] ,

SEMA[ s : NAT ] = (  when ( s > 0 ) claim -> SEMA[ s - 1 ]
                   | release                -> SEMA[ s + 1 ] ),

SEMA[ MAX_NAT + 1 ] = ERROR .
```

FSP can **only model finite concurrent systems**, so we only model semaphores that take a *finite range of values*, i.e. 0 – 3.

SEMAPHORE allows a maximum of three **claim** actions to be accepted before a **release** action must occur.

If a **release** action results in a semaphore value of 4, then it is treated as an error & is defined as the error process ERROR.

Behaviour of the FSP Semaphore Process

The semaphore's behaviour is illustrated in the LTSA diagram Fig. 11.4.

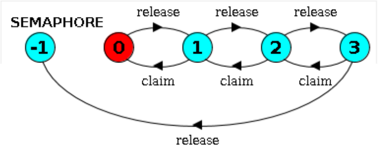

Figure : 11.4 SEMAPHORE LTS

This directly models the first (simple) definition for a semaphore given above.

`claim` is only accepted when $(\mathbf{s} > 0)$ in the `SEMAPHORE`.

`release` is **not guarded** (or the guard is just `true`).

`SEMAPHORE` has values in the range `0..MAX_NAT` & has an initial value `N`.

If a **`release`** causes `MAX_NAT` to be exceeded then `SEMAPHORE` moves to the `ERROR` state **−1**.

When `SEMAPHORE` is used in a larger model, we must ensure that this `ERROR` state **does not occur**.

FSP Semaphore Example

We use a semaphore **mutex**, shared by three processes `mp[1..3]`, to ensure *mutually exclusive* access to some resource.

Each process performs the action **mutex.claim** to get exclusive access & **mutex.release** to release it.

Access to the resource (via critical section), is modelled by the action `critical[i]`.

The composite process `SEMADEMO` , which combines the three processes `mp[i]:MutexProcess` & a semaphore is:

SEMADEMO

```
range IDS = 1..3

MutexProcess( ID = 0 ) = ( mutex.claim ->
                               critical[ID] ->
                           mutex.release -> MutexProcess ) .

|| MutexProcesses = ( forall [ i : IDS ]
                          ( mp[ i ] : MutexProcess(i) ) ) .

||SEMADEMO  = (     MutexProcesses
                 || { mp[IDS] } :: mutex : SEMAPHORE(1) ) .
```

Ensuring Mutual Exclusion Using the Semaphore

To achieve *mutual exclusion* use a *binary semaphore* initially **unlocked**, i.e. 1.

The first process that tries to execute its critical action, performs a `mutex.claim` making the value of mutex zero.

No further process can perform `mutex.claim` until the original process releases mutual exclusion by `mutex.release`.

Condition Synchronisation in Java (Monitors)

Java *monitor objects* have a *synchronisation lock* & a thread *wait set*.

Recall the following methods provided by the `Object` class:

- `wait()` – place current thread in the wait set.
- `notify()` – wake up a single thread from wait set.
- `notifyAll()` – wake up all threads in wait set.

The operations **fail** if called by a thread that does **not currently "own" the monitor**, i.e. has the synchronisation lock.

A thread enters a monitor when it **acquires the mutual exclusion lock** associated with the monitor & exits the monitor when it **releases the lock**.

A thread calling `wait()` *exits the monitor*.

This allows other threads to enter the monitor &, when the appropriate condition is satisfied, to call `notify()` or `notifyAll()` to awake waiting threads.

Condition Synchronisation in FSP

The basic format for modelling a *guarded action* for some condition **cond** & action **action** using FSP is shown below:

```
FSP:    when ( cond ) action -> NEWSTATE
```

The corresponding format for implementing the *guarded action* for condition **cond** & action **action** using Java is as follows:

```
public synchronized  void action( )
       throws  InterruptedException
{
    while ( !cond )    // "when ( cond )"
    {
        wait() ;       // cond = false
    }
    // cond = true

    // "action -> NEWSTATE"
    // modify monitor data "action -> NEWSTATE"

    notifyAll() ;
}
```

Notes on the Translation

The `while` loop is necessary to ensure that **`cond`** is indeed satisfied when a *thread re-enters the monitor*.

Although the thread invoking `wait()` may have been notified that **`cond`** is satisfied, thereby releasing it from the monitor wait set.

But **`cond`** may be **invalidated by another thread** that runs between the time that the waiting thread is awakened & the time it re-enters the monitor (by acquiring the lock).

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.

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.

Translation General Rules

General Rules: for guiding the translation of an *FSP process model* into a *Java monitor* are as follows:

1. In general if an FSP process that represents a shared object offers a *choice* ("`|`") between several *unguarded actions* & *guarded actions*, then as a first attempt at translation each one should be implemented as a separate `synchronized` method.

2. The *choice* ("`|`") is represented in the monitor by the fact that a user of the shared resource (monitor) is *"offered the choice"* of calling any one of the `synchronized` methods.

3. Each *guarded action* in the FSP model of a monitor is implemented as a:

   - `synchronized` method
   - that uses a `while` loop & `wait()` to implement the guard.

4. The `while` loop **condition** is the **logical negation** (`!`) of the **model guard condition**.

Translating the FSP Semaphore into Java

There is of course Java's pre-defined semaphore class:

`java.util.concurrent.Semaphore`

that we briefly looked at in a previous lecture.

But we shall provide our own explicitly defined Java Semaphore class.

Semaphores are *passive* objects that react to `claim` & `release` actions; they do **not initiate actions**.

Consequently, we implement a semaphore as a Java *monitor* class.

The actions `claim` & `release` become `synchronized` methods of our monitor semaphore class.

Translating the FSP Semaphore into Java: Guard

The guard: "**when (v > 0)**", on the `claim` action in the FSP process model, is implemented using *conditional synchronisation*.

That is, we must use the *monitor guard* `while` loop:

```
while ( BExp )
{
  wait() ;
}
```

The `claim` changes the state of the monitor by decrementing its value `s`.

However, we do **not use** `notify()` to *signal the change in state*.

This is because threads only wait for the value of the semaphore to be *incremented*, they **do not wait** for the value to be *decremented*.

Implementing FSP's SEMAPHORE Process using a Java Semaphore class

Note that this is a very simple implementation of a semaphore.

```
public class Semaphore
{
  private int value ;

  public Semaphore ( int initial ) {  value = initial ;  }

  public synchronized  void claim( ) throws InterruptedException
  {
    while ( value == 0 )
    {
        wait() ;
    }
    value-- ;
  }

  public synchronized void release( )
  {
     value++ ;
     notify() ;
  }
}
```

Implementing FSP's `MutexProcess` using a Java Thread

The FSP `MutexProcess` process is implemented as a Java `Thread` class.

Each one performs the action **`mutex.claim()`** to get exclusive access & **`mutex.release()`** to release it.

```
class  MutexProcess extends Thread
{
  private Semaphore mutex ;

  MutexProcess( Semaphore sema )
  {
    super( "MutexProcess" ) ;   // Thread constructor
    mutex = sema ;
  }

  public void run()
  {
    try {   while ( true )
            {
               mutex.claim() ;        // get ME
               // critical section
               mutex.release() ;      // release ME
            }
    } catch (InterruptedException e){}
  }
} // MutexProcess
```

Implementing FSP's SEMADEMO Process using a Java Thread class

The Java version of the composite process SEMADEMO, which combines the three processes MutexProcess & a semaphore is:

```
class SEMADEMO
{
  public static void main( String args[] )
  {
    final int UNLOCKED = 1 ;

    Semaphore mutex = new Semaphore( UNLOCKED ) ;

    Thread mp[] = new Thread[3] ;

    // create 3 MutexProcess
    mp[0] = new MutexProcess( mutex ) ;
    mp[1] = new MutexProcess( mutex ) ;
    mp[2] = new MutexProcess( mutex ) ;

    // start  MutexProcess
    mp[0].start() ;
    mp[1].start() ;
    mp[2].start() ;
  }
} // SEMADEMO
```

The End.