Chapter 4

Shared Objects & Mutual Exclusion
Shared Objects & Mutual Exclusion

**Concepts:** process interference. mutual exclusion.

**Models:** model checking for interference modeling mutual exclusion

**Practice:** thread interference in shared Java objects mutual exclusion in Java *(synchronized objects/methods).*
4.1 Interference

Ornamental garden problem:
People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.

The concurrent program consists of two concurrent threads and a shared counter object.
The **Turnstile** thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the **increment()** method of the counter object.
The **Counter** object and **Turnstile** threads are created by the `go()` method of the Garden applet:

```java
private void go() {
    counter = new Counter(counterD);
    west = new Turnstile(westD, counter);
    east = new Turnstile(eastD, counter);
    west.start();
    east.start();
}
```

Note that **counterD**, **westD** and **eastD** are objects of **NumberCanvas** used in chapter 2.
Turnstile class

class Turnstile extends Thread {
    NumberCanvas display;
    Counter people;

    Turnstile(NumberCanvas n, Counter c)
    {
        display = n; people = c;
    }

    public void run() {
        try{
            display.setvalue(0);
            for (int i=1; i<=Garden.MAX; i++) {
                Thread.sleep(500); // 0.5 second between arrivals
                display.setvalue(i);
                people.increment();
            }
        }
        catch (InterruptedException e) {} }
}

The run() method exits and the thread terminates after Garden.MAX visitors have entered.
Counter class

class Counter {
    int value=0;
    NumberCanvas display;

    Counter(NumberCanvas n) {
        display=n;
        display.setvalue(value);
    }

    void increment() {
        int temp = value; //read value
        Simulate.HWinterrupt();
        value=temp+1; //write value
        display.setvalue(value);
    }
}

Hardware interrupts can occur at arbitrary times.

The counter simulates a hardware interrupt during an increment(), between reading and writing to the shared counter value. Interrupt randomly calls Thread.yield() to force a thread switch.
After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. **Why?**
concurrent method activation

Java method activations are not atomic - thread objects **east** and **west** may be executing the code for the increment method at the same time.

```
<table>
<thead>
<tr>
<th>west</th>
<th>shared code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>increment:</td>
</tr>
<tr>
<td>program counter</td>
<td>read value</td>
</tr>
<tr>
<td></td>
<td>write value + 1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>east</th>
<th>program counter</th>
</tr>
</thead>
</table>
```

©Magee/Kramer
Process \texttt{VAR} models read and write access to the shared counter \texttt{value}.

Increment is modeled inside \texttt{TURNSTILE} since Java method activations are not atomic i.e. thread objects \texttt{east} and \texttt{west} may interleave their \texttt{read} and \texttt{write} actions.
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} } 

VAR = VAR[0],
VAR[u:T] = (read[u]  ->VAR[u]
            |write[v:T]->VAR[v]).

TURNSTILE = (go    -> RUN),
RUN = (arrive-> INCREMENT
      |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
          -> value.write[x+1]->RUN
     )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
          || { east,west,display} ::value:VAR)
         /{ go /{ east,west} .go,
            end/{ east,west} .end} .

The alphabet of process VAR is declared explicitly as a set constant, VarAlpha.

The alphabet of TURNSTILE is extended with VarAlpha to ensure no unintended free actions in VAR ie. all actions in VAR must be controlled by a TURNSTILE.
Scenario checking - use animation to produce a trace.

Is this trace correct?
Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

\[
\begin{align*}
\text{TEST} & = \text{TEST}[0], \\
\text{TEST}[v:T] & = \\
& \quad \left( \text{when}\ (v<N) \{\text{east.arrive, west.arrive}\} \rightarrow \text{TEST}[v+1] \right. \\
& \quad \left| \text{end} \rightarrow \text{CHECK}[v] \right), \\
\text{CHECK}[v:T] & = \\
& \quad \left( \text{display.value.read}[u:T] \rightarrow \\
& \quad \quad \left( \text{when}\ (u==v) \ \text{right} \rightarrow \text{TEST}[v] \\
& \quad \quad \ | \text{when}\ (u!=v) \ \text{wrong} \rightarrow \text{ERROR} \right) \right) \\
& \quad \left. \right) + \left\{ \text{display.VarAlpha} \right\}.
\end{align*}
\]

Like \textsc{STOP}, \textsc{ERROR} is a predefined FSP local process (state), numbered -1 in the equivalent LTS.
ornamental garden model - checking for errors

\[ \text{TESTGARDEN} = (\text{GARDEN} \mid \mid \text{TEST}) . \]

Use **LTSA** to perform an exhaustive search for **ERROR**.

Trace to property violation in TEST:

- go
- east.arrive
- east.value.read.0
- west.arrive
- west.value.read.0
- east.value.write.1
- west.value.write.1
- end
- display.value.read.1
- wrong

**LTSA** produces the shortest path to reach **ERROR**.
Interference and Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed interference.

Interference bugs are extremely difficult to locate. The general solution is to give methods mutually exclusive access to shared objects. Mutual exclusion can be modeled as atomic actions.
4.2 Mutual exclusion in Java

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword `synchronized`.

We correct `COUNTER` class by deriving a class from it and making the increment method `synchronized`:

```java
class SynchronizedCounter extends Counter {
    SynchronizedCounter(NumberCanvas n) {
        super(n);
    }

    synchronized void increment() {
        super.increment();
    }
}
```
Java associates a *lock* with every object. The Java compiler inserts code to acquire the lock before executing the body of the synchronized method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.
Access to an object may also be made mutually exclusive by using the `synchronized` statement:

```java
synchronized (object) {
    statements
}
```

A less elegant way to correct the example would be to modify the `Turnstile.run()` method:

```java
synchronized (counter) {
    counter.increment();
}
```

Why is this “less elegant”?

To ensure mutually exclusive access to an object, all **object methods** should be synchronized.
4.3 Modeling mutual exclusion

To add locking to our model, define a LOCK, compose it with the shared VAR in the garden, and modify the alphabet set:

\[
\text{LOCK} = (\text{acquire} \rightarrow \text{release} \rightarrow \text{LOCK}).
\]
\[
\text{LOCKVAR} = (\text{LOCK} \mid | \mid \text{VAR}).
\]
\[
\text{set VarAlpha} = \{\text{value.}\{\text{read}[T],\text{write}[T],\text{acquire}, \text{release}\}\}
\]

Modify TURNSTILE to acquire and release the lock:

\[
\text{TURNSTILE} = (\text{go} \rightarrow \text{RUN}),
\]
\[
\text{RUN} = (\text{arrive} \rightarrow \text{INCREMENT}
\mid \text{end} \rightarrow \text{TURNSTILE}),
\]
\[
\text{INCREMENT} = (\text{value.acquire}
\rightarrow \text{value.read}[x:T]\rightarrow\text{value.write}[x+1]
\rightarrow \text{value.release}\rightarrow\text{RUN}
) + \text{VarAlpha}.
\]
Revised ornamental garden model - checking for errors

A sample animation execution trace

go
east.arrive
east.value.acquire
east.value.read.0
east.value.write.1
east.value.release
west.arrive
west.value.acquire
west.value.read.1
west.value.write.2
west.value.release
end
display.value.read.2
right

Use TEST and LTSA to perform an exhaustive check.

Is TEST satisfied?
To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

For SynchronizedCounter we hide read, write, acquire, release actions.

```
const N = 4
range T = 0..N
VAR = VAR[0],
VAR[u:T] = ( read[u]->VAR[u]  
LOCK = (acquire->release->LOCK).
INCREMENT = (acquire->read[x:T]  
|  when (x<N) write[x+1]  
|  release->increment->INCREMENT  
|  ) + {read[T],write[T]}.
COUNTER = (INCREMENT||LOCK||VAR){increment}.
```
We can give a more abstract, simpler description of a COUNTER which generates the same LTS:

\[
\text{COUNTER} = \text{COUNTER}[0] \\
\text{COUNTER}[v:T] = (\text{when } (v<N) \text{ increment } \rightarrow \text{COUNTER}[v+1]).
\]

This therefore exhibits “equivalent” behavior i.e. has the same observable behavior.
Summary

◆ Concepts
  • process interference
  • mutual exclusion

◆ Models
  • model checking for interference
  • modeling mutual exclusion

◆ Practice
  • thread interference in shared Java objects
  • mutual exclusion in Java (synchronized objects/methods).