A Concert Hall Booking System

A central computer connected to remote terminals via communication links is used to automate seat reservations for a concert hall.

To book a seat, a client chooses a free seat and the clerk enters the number of the chosen seat at the terminal and issues a ticket, if it is free.

A system is required which avoids double bookings of the same seat whilst allowing clients free choice of the available seats.

Construct an abstract model of the system and demonstrate that your model does not permit double bookings.

```fsharp
const False = 0
const True = 1
range Bool = False..True


range Seats = 1..2 ||SEATS = (seat[Seats]:SEAT).
```

Like STOP, ERROR is a predefined FSP local process (state), numbered -1 in the equivalent LTS.
Concert Hall Booking System

TERMINAL = (choose[s:Seats] -> seat[s].query[reserved:Bool] -> if (!reserved) then (seat[s].reserve -> TERMINAL) else TERMINAL).

set Terminals = {a,b}

||CONCERT = (Terminals:TERMINAL || Terminals::SEATS).

Does this system allow double booking of a seat?

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.

||CONCERT = (Terminals:TERMINAL || Terminals::SEATS || Terminals::LOCK).

Would locking at the seat level permit more concurrency?

ornamental garden Program - class diagram

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.
ornamental garden program

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.

Counter class

```java
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.sleep()` to force a thread switch.

Turnstile class

```java
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.

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.

**east**
- PC
- program counter
- increment:
  - read value
  - write value + 1

**west**
- PC
- program counter
- shared code

Process **VAR** models read and write access to the shared counter **value**.

Increment is modeled inside **TURNSTILE** since Java method activations are not atomic i.e. thread objects **east** and **west** may interleave their **read** and **write** actions.

---

ornamental garden Model

```
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 shared process **VAR** is declared explicitly as a set constant, **VarAlpha**.

The **TURNSTILE** alphabet is extended with **VarAlpha** to ensure no unintended free (autonomous) actions in **VAR** such as **value.write[0]**.

All actions in the shared **VAR** must be controlled (shared) by a **TURNSTILE**.

---

checking for errors - animation

Scenario checking - use animation to produce a trace.

Is this trace correct?
checking for errors - exhaustive analysis

Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

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

Like STOP, ERROR is a predefined FSP local process (state), numbered -1 in the equivalent LTS.

ornamental garden model - checking for errors

||TESTGARDEN = (GARDEN || 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.

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, which uses a lock on the object.

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();
    }
}
```

Interference and Mutual Exclusion (mutex)

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 (often referred to as “mutex”) can be modeled as atomic actions.
mutual exclusion - the ornamental garden

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.

Java synchronized statement

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 (people) {people.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:

```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).
set VarAlpha = {value.{read[T],write[T], acquire, release}}
```

Modify TURNSTILE to acquire and release the lock:

```
TURNSTILE = (go -> RUN),
RUN = (arrive-> INCREMENT |end -> TURNSTILE),
INCREMENT = (value.acquire -> value.read[x:T]->value.write[x+1] -> value.release->RUN )+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 LTS to perform an exhaustive check. Is TEST satisfied?

COUNTER: Abstraction using action hiding

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]
| write[v:T]->VAR[v]).
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.

Minimized LTS:

We can give a more abstract, simpler description of a COUNTER which generates the same LTS:

COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment -> COUNTER[v+1]).

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

4.4 Java Concurrency Utilities Package

Java SE 5 introduced a package of advanced concurrency utilities in java.util.concurrent, later extended in JSE8.

This includes many additional, explicit mechanisms such as atomic variables, a task scheduling framework (for thread and thread pool instantiation and control), and synchronizers such as semaphores (later), mutexes, barriers and explicit locks with timeout.

Explicit locks are more dangerous and less efficient than synchronized. Use only if required for non-block structured situations requiring flexibility (such as chain locking: acquire lock on A, then on B, then release A and acquire C, ...) or for other advanced features such as timed or polled lock acquisition. (At the memory level, volatile can be used to force reads/writes on a shared variable to main memory rather than cached thread-locally).

class LockedCounter extends Counter {
    final ReentrantLock inclock = new ReentrantLock();
    LockedCounter(NumberCanvas n)
    {super(n);}
    public void increment()throws InterruptedException {
        inclock.lock();
        try {
            super.increment();
        } finally {inclock.unlock();}
    }
}
Summary

◆ Concepts
  • process interference
  • mutual exclusion and locks

◆ Models
  • model checking for interference
  • modelling mutual exclusion

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