COMP30112: Concurrency

Topics 2.2: Concurrency in FSP & 3.1: Java Threads

Howard Barringer

Room KB2.20: email: Howard.Barringer@manchester.ac.uk

February 2009
Outline

Topic 2.2: Modelling Processes with FSP - II
  Composing Processes in Parallel
  Interaction
  Labelling Processes
  Relabelling & Hiding
  Summary

Topic 3.1: Java Threads: Revision
  Threads
  Example
  Synchronisation
  Example
  Thread Lifecycle
Parallel Composition

★ Consider ‘DAY’ processes? ★

\((P \parallel Q)\) represents the concurrent execution of \(P\) and \(Q\).
The operator \(\parallel\) is the parallel composition operator.

\[
\text{LAUGH} = (\text{laugh} \rightarrow \text{STOP}).
\]
\[
\text{CONVERSE} = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}).
\]
\[
\parallel \text{CONVERSE} \text{_LAUGH} = (\text{LAUGH} \parallel \text{CONVERSE}).
\]

★ Possible Traces? ★
Interleaving Actions

**LAUGH**

0 ➔ laugh ➔ 1

2 states

**CONVERSE**

0 ➔ think ➔ talk ➔ 2

3 states

★ Composite LTS? ★
Example: Clock Radio

\[
\begin{align*}
\text{CLOCK} &= (\text{tick} \rightarrow \text{CLOCK}). \\
\text{RADIO} &= (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}). \\
\text{\| CLOCK } \text{\| RADIO} &= (\text{CLOCK} \| \text{RADIO}).
\end{align*}
\]

★ LTS? ★ ★ Traces? ★ ★ Number of states? ★
Shared Actions

- processes in a composition may have common actions: *shared* actions (i.e. process *alphabets* intersect)
- shared actions model process interaction
- unshared actions: arbitrary interleaving
- shared actions: must be executed at the same time by all processes

MAKER = (make→ready→MAKER).  
USER = (ready→use→USER).

|| MAKER_USER = (MAKER || USER).

★ LTS? ★ ★ Traces? ★ ★ Number of states? ★
Handshake

Handshake is an action acknowledged by another:

\[
\text{MAKER}v2 = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}v2).
\]

\[
\text{USER}v2 = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}v2).
\]

\[
\text{|| MAKER \_ USER}v2 = (\text{MAKER}v2 \ || \ \text{USER}v2).
\]

★ LTS? ★
Multiple Processes

Multi-party synchronisation:

\[
\begin{align*}
\text{MAKE}_A & = (\text{makeA} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_A). \\
\text{MAKE}_B & = (\text{makeB} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_B). \\
\text{ASSEMBLE} & = (\text{ready} \rightarrow \text{assemble} \rightarrow \text{used} \rightarrow \text{ASSEMBLE}). \\
|| \text{FACTORY} & = (\text{MAKE}_A || \text{MAKE}_B || \text{ASSEMBLE}).
\end{align*}
\]

★★ LTS? ★★
Process Labelling - I

\( \mathbf{a} : \mathbf{P} \) prefixes each action label in the alphabet of \( \mathbf{P} \) with \( \mathbf{a} \).

Two instances of a switch process:

\[
\text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}).
\]

\[
\parallel \text{TWO} \_ \text{SWITCH} = (\mathbf{a} : \text{SWITCH} \parallel \mathbf{b} : \text{SWITCH}).
\]

An array of instances of the switch process:

\[
\parallel \text{SWITCHES}(N = 3) = (\text{forall}[i : 1..N] \ s[i] : \text{SWITCH}).
\]

\[
\parallel \text{SWITCHES}(N = 3) = (s[i : 1..N] : \text{SWITCH}).
\]

★ Write \( \text{SWITCHES}(3) \) in basic FSP? ★
Process Labelling - II

\{a_1, \ldots, a_x\} :: P \text{ replaces every action label } n \text{ in the alphabet of } P \text{ with the labels } a_1.n, \ldots, a_x.n. \text{ Further, every transition } n \rightarrow X \text{ in the definition of } P \text{ is replaced with the transitions } \{a_1.n, \ldots, a_x.n\} \rightarrow X.

Process prefixing is useful for modeling shared resources:

\begin{align*}
\text{RESOURCE} & = (\text{acquire} \rightarrow \text{release} \rightarrow \text{RESOURCE}) . \\
\text{USER} & = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) . \\
\| \text{RESOURCE\_SHARE} & = (a : \text{USER} \| b : \text{USER} \\
& \| \{a, b\} :: \text{RESOURCE}) .
\end{align*}

★ Write \text{RESOURCE\_SHARE} \text{ in basic FSP?} ★
Process Prefix Labels For Shared Resources

★ LTS for RESOURCE_SHARE? ★
Relabelling functions are applied to processes to change the names of action labels. The general form of the relabelling function is:

\[
/\{newlabel_1/oldlabel_1, \ldots, newlabel_n/oldlabel_n\}.
\]

Relabelling to ensure that composed processes synchronize on particular actions:

\[
\begin{align*}
\text{CLIENT} &= (\text{call} \rightarrow \text{wait} \rightarrow \text{continue} \rightarrow \text{CLIENT}). \\
\text{SERVER} &= (\text{request} \rightarrow \text{service} \rightarrow \text{reply} \rightarrow \text{SERVER}). \\
\text{CLIENT} \parallel \text{SERVER} &= (\text{CLIENT} \parallel \text{SERVER}) \backslash \{\text{call}/\text{request, reply}/\text{wait}\}.
\end{align*}
\]
Client Server LTSs

CLIENT

0 \rightarrow 1 \rightarrow 2 \rightarrow 0

continue

call \rightarrow reply

SERVER

0 \rightarrow 1 \rightarrow 2 \rightarrow 0

reply

call \rightarrow service

CLIENT_SERVER

0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0

continue

call \rightarrow service \rightarrow reply

Alternative formulation of the client server system is described below using qualified or prefixed labels:

\[
\begin{align*}
\text{SERVER}v2 &= (\text{accept}.\text{request} \\
&\quad \rightarrow \text{service} \rightarrow \text{accept}.\text{reply} \rightarrow \text{SERVER}v2). \\
\text{CLIENT}v2 &= (\text{call}.\text{request} \\
&\quad \rightarrow \text{call}.\text{reply} \rightarrow \text{continue} \rightarrow \text{CLIENT}v2). \\
\big|\big| \text{CLIENT} \cdot \text{SERVER}v2 &= (\text{CLIENT}v2 \big|\big| \text{SERVER}v2) \\
&/\{\text{call}/\text{accept}\}.
\end{align*}
\]
Action Hiding

When applied to a process $P$, the hiding operator $\{a_1..a_x\}$ removes the action names $a_1..a_x$ from the alphabet of $P$ and makes these concealed actions “silent”. These silent actions are labelled $\text{tau}$. Silent actions in different processes are not shared.

Sometimes it is more convenient to specify the set of labels to be exposed...

When applied to a process $P$, the interface operator $@\{a_1..a_x\}$ hides all the actions in the alphabet of $P$ not labelled in the set $a_1..a_x$. 
The following definitions are equivalent:

\[ \text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \setminus \{\text{use}\} . \]

\[ \text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \odot \{\text{acquire, release}\} . \]
Summary

- process composition e.g. \((P \parallel Q \parallel R)\)
  - shared actions synchronised
  - unshared actions interleaved
- process labeling e.g. \(a : P\) and \(\{a, b\} :: P\)
- action relabelling e.g. \(P/\{a/b, c/d\}\)
- action hiding e.g. \(P\{a, b\}\) and \(P@\{c, d\}\)
The Thread Class

class MyThread extends Thread {
    int i;

    public void run() {
        i = 0;
        while (i <= 5) {
            System.out.println("" + i);
            i++;
        }
    }

    ...  
    MyThread mt = new MyThread();
    mt.start();
    ...
}
The Runnable Interface

```java
public interface Runnable{
    public abstract void run();
}

class MyRun implements Runnable {
    int i;

    public void run() {
        i = 0;
        while (i <= 5) {
            System.out.println("" + i);
            i++;
        }
    }
}

MyRun mr = new MyRun();
Thread mt = new Thread(mr);
mt.start();
...
class Counter implements Runnable {

    Thread counter;    int i;
    final static int N = 3;

    Counter () {
        counter = new Thread(this);
        i = N;
        counter.start();
    }
}
Terminal Counter - cont.

```java
public void run () {
    while (true) {
        if (counter == null) return;
        if (i>0) {
            System.out.println("Count is " + i);
            try { Thread.sleep(500); } 
            catch (InterruptedException e) {}  
            i--;
        }
        if (i==0) {
            System.out.println("Count is done");
            return;
        }
    }  //while
}  //run  }  //class
```
class CounterText {

    public static void main (String [] args) {

        System.out.println("Main starting counter...");
        Counter count = new Counter();
        System.out.println("Main Done");

    }//main

}//class CounterText
Thread Synchronisation

synchronized (obj) statements
synchronized typename methodname () statement

This provides *mutually exclusive* access to methods/objects.
• each object has a single lock associated with it
• any synchronized code applied to an object must first obtain the lock for the object
• if the lock is already taken, then thread will block until lock is available
• exception releases lock
• scheduling policy for waiting threads
wait() and notify()

- **wait()**: suspend and block thread; release lock
- **notify()**: unblock a suspended thread
- **notifyAll()**: unblock all suspended threads
- **wait sets**
Condition Synchronization

```java
synchronized type1 Method1 (...) throws InterruptedException {
    while (!firstCondition){ wait(); }
    ...
}
```

```java
synchronized type2 Method2 (...) {
    ...
    firstCondition = true;
    notify(); // or notifyAll()
    ...
}
```

Note: *signal-and-continue* semantics for `notify()`
Car Park

![Car Park Diagram](image-url)
Car Park Control Class

class CarParkControl {

    protected int spaces;
    protected int capacity;

    CarParkControl(int n) {
        capacity = spaces = n;
    }

    synchronized void arrive() throws InterruptedException {
        while (spaces==0) wait();
        --spaces;
        notifyAll();
    }

    synchronized void depart() throws InterruptedException{
        while (spaces==capacity) wait();
        ++spaces;
        notifyAll();
    }

}
Car Park Arrivals class

class Arrivals implements Runnable {

    CarParkControl carpark;

    Arrivals(CarParkControl c) {
        carpark = c;
    }

    public void run() {
        try {
            while(true) {
                ThreadPanel.rotate(330);
                carpark.arrive();
                ThreadPanel.rotate(30);
            }
        } catch (InterruptedException e){}
    }
}
class Departures implements Runnable {

    CarParkControl carpark;

    Departures(CarParkControl c) {
        carpark = c;
    }

    public void run() {
        try {
            while(true) {
                ThreadPanel.rotate(180);
                carpark.depart();
                ThreadPanel.rotate(180);
            }
        } catch (InterruptedException e){}
    }
}
Thread Lifecycle
FSP for Thread Lifecycle

\[
\begin{align*}
\text{THREAD} & \quad = \quad \text{CREATED}, \\
\text{CREATED} & \quad = \quad (\text{start} \rightarrow \text{RUNNING} \\
& \quad \vert \text{stop} \rightarrow \text{TERMINATED}), \\
\text{RUNNING} & \quad = \quad (\{\text{suspend, sleep}\} \rightarrow \text{NON\_RUNNABLE} \\
& \quad \vert \text{yield} \rightarrow \text{RUNNABLE} \\
& \quad \vert \{\text{stop, end}\} \rightarrow \text{TERMINATED} \\
& \quad \vert \text{run} \rightarrow \text{RUNNING}), \\
\text{RUNNABLE} & \quad = \quad (\text{suspend} \rightarrow \text{NON\_RUNNABLE} \\
& \quad \vert \text{dispatch} \rightarrow \text{RUNNING} \\
& \quad \vert \text{stop} \rightarrow \text{TERMINATED}), \\
\text{NON\_RUNNABLE} & \quad = \quad (\text{resume} \rightarrow \text{RUNNABLE} \\
& \quad \vert \text{stop} \rightarrow \text{TERMINATED}), \\
\text{TERMINATED} & \quad = \quad \text{STOP}.
\end{align*}
\]
LTS for Thread Lifecycle