COMP31212: Concurrency

Concurrency in FSP (topic 2.2) & Java Threads (topic 3.1)
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
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

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

\[
\begin{align*}
\text{LAUGH} & \quad = \quad \text{(laugh} \rightarrow \text{STOP}). \\
\text{CONVERSE} & \quad = \quad \text{(think} \rightarrow \text{talk} \rightarrow \text{STOP}).
\end{align*}
\]

\[\parallel \text{ CONVERSE_LAUGH} \quad = \quad \text{(LAUGH} \parallel \text{ CONVERSE}).\]

★ Possible traces?? ★
Interleaving Actions

- **LAUGH**
  - States: 0, 1
  - Transitions: laugh

- **CONVERSE**
  - States: 0, 1, 2
  - Transitions: think, talk

2 states

3 states
Interleaving Actions

LAUGH

\[
\begin{array}{c}
0 \\
\text{laugh} \\
1
\end{array}
\]

2 states

CONVERSE

\[
\begin{array}{c}
0 \\
\text{think} \\
1 \\
\text{talk} \\
2
\end{array}
\]

3 states

★ Composite LTS?? ★
Example: Clock Radio

\[
\text{CLOCK} = (\text{tick} \rightarrow \text{CLOCK}).
\]

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

\[
||\text{CLOCK} \_ \text{RADIO} || = (\text{CLOCK} || \text{RADIO}).
\]
Example: Clock Radio

\[
\text{CLOCK} = (\text{tick} \rightarrow \text{CLOCK}).
\]
\[
\text{RADIO} = (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}).
\]
\[
\parallel \text{CLOCK} \mathbin{\parallel} \text{RADIO} = (\text{CLOCK} \mathbin{\parallel} \text{RADIO}).
\]

★ LTS? Traces? Number of states?? ★
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
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
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:

\[
\text{MAKERv2} = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKERv2}).
\]
\[
\text{USERv2} = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USERv2}).
\]

\[
\text{MAKER} \parallel \text{USERv2} = (\text{MAKERv2} \parallel \text{USERv2}).
\]

★ LTS? ★
Handshake

Handshake is an action acknowledged by another:

\[
\begin{align*}
\text{MAKER}v2 & = (\text{make} \to \text{ready} \to \text{used} \to \text{MAKER}v2). \\
\text{USER}v2 & = (\text{ready} \to \text{use} \to \text{used} \to \text{USER}v2).
\end{align*}
\]

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

★ LTS? ★
Handshake

Handshake is an action acknowledged by another:

\[
\begin{align*}
\text{MAKER}v2 &= (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}v2).
\text{USER}v2 &= (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}v2).
\| \text{MAKER} \_ \text{USER}v2 &= (\text{MAKER}v2 \| \text{USER}v2).
\end{align*}
\]

★ LTS? ★
Handshake

Handshake is an action acknowledged by another:

\[
\text{MAKERv2} = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKERv2}). \\
\text{USERv2} = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USERv2}). \\
\|
\text{MAKER}_\cup \text{USERv2} = (\text{MAKERv2} \| \text{USERv2}).
\]

★ 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? ★
Multiple Processes

Multi-party synchronisation:

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

★ 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? ★
Multiple Processes

Multi-party synchronisation:

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

★ LTS? ★
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
Process Labelling - I

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

Two instances of a switch process:

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

\[
\|\text{TWO\_SWITCH} = (a : \text{SWITCH} \| b : \text{SWITCH}).
\]
Process Labelling - I

a : P prefixes each action label in the alphabet of P with a.

Two instances of a switch process:

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

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

An array of instances of the switch process:

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

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

These are equivalent expressions.
Process Labelling - I

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

Two instances of a switch process:

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

\[
\| \text{TWO\_SWITCH} = (a : \text{SWITCH} \| b : \text{SWITCH}).
\]

An array of instances of the switch process:

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

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

These are equivalent expressions.

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

\(\{a_1, \ldots, ax\} :: P\) replaces every action label \(n\) in the alphabet of \(P\) with the labels \(a_1.n, \ldots, ax.n\). Further, every transition \(n\rightarrow X\) in the definition of \(P\) is replaced with the collection of transitions \(\{a_1.n, \ldots, ax.n\}\rightarrow X\).
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_1n, \ldots, axn. \text{ Further, every transition } n\rightarrow X \text{ in the definition of } P \text{ is replaced with the collection of transitions } \{a_1n, \ldots, axn\} \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}).
\end{align*}

To run TWO users and the resource in parallel:

\[\| \text{RESOURCE}_\text{SHARE} = (a : \text{USER} \| b : \text{USER} \| \{a, b\} :: \text{RESOURCE})\].
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_1n, \ldots, axn. \text{ Further, every transition } n -> X \text{ in the definition of } P \text{ is replaced with the collection of transitions } \{a_1n, \ldots, axn\} -> X.

Process prefixing is useful for modeling shared resources:

\[
\text{RESOURCE} = (\text{acquire} -> \text{release} -> \text{RESOURCE}). \\
\text{USER} = (\text{acquire} -> \text{use} -> \text{release} -> \text{USER}).
\]

To run TWO users and the resource in parallel:

\[
\text{RESOURCE\_SHARE} = (a : \text{USER} \mid \mid b : \text{USER} \mid \mid \{a, b\} :: \text{RESOURCE}).
\]

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

★ LTS for RESOURCE_SHARE? ★
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
Action Relabelling - I

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

\[
/\{newlabel_1/oldlabel_1, \ldots, newlabel_n/oldlabel_n\}.
\]
Action Relabelling - I

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

\[
\{\text{newlabel}_1 / \text{oldlabel}_1, \ldots, \text{newlabel}_n / \text{oldlabel}_n\}.
\]

Relabelling to ensure that composed processes synchronize on particular actions:

\[
\text{CLIENT} = (\text{call} \rightarrow \text{wait} \rightarrow \text{continue} \rightarrow \text{CLIENT}).
\]
\[
\text{SERVER} = (\text{request} \rightarrow \text{service} \rightarrow \text{reply} \rightarrow \text{SERVER}).
\]
\[
\text{CLIENT} \parallel \text{SERVER} = (\text{CLIENT} \parallel \text{SERVER})
\]
\[
\{\text{call} / \text{request}, \text{reply} / \text{wait}\}.
\]
Client Server LTSs
Alternative formulation of the client server system is described below using qualified or prefixed labels:

\[
\text{SERVER}v2 = (\text{accept.request} \\
\text{\quad} \rightarrow \text{service} \rightarrow \text{accept.reply} \rightarrow \text{SERVER}v2). \\
\text{CLIENT}v2 = (\text{call.request} \\
\text{\quad} \rightarrow \text{call.reply} \rightarrow \text{continue} \rightarrow \text{CLIENT}v2). \\
\|
\text{CLIENT \_ SERVER}v2 = (\text{CLIENT}v2 \|	ext{SERVER}v2) \\
/\{\text{call/accept}\}.
\]
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 \( \text{tau} \) (the Greek letter \( \tau \) is standard in process algebras for this ‘internal’ or ‘silent’ action).

Tau actions are special (not just another action name), in particular one cannot synchronize on tau actions: tau actions in different processes are not shared.
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 $\tau$ (the Greek letter $\tau$ is standard in process algebras for this ‘internal’ or ‘silent’ action).

Tau actions are special (not just another action name), in particular one cannot synchronize on tau actions: tau 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 $\oplus\{a_1..a_x\}$ hides all the actions in the alphabet of $P$ not labelled in the set $a_1..a_x$. 
The following definitions are equivalent:

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

\[
\text{USER} \quad = \quad (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \quad @\{\text{acquire, release}\}.
\]
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
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 $P@\{c, d\}$
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
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();
...
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
Terminal Counter

class Counter implements Runnable {

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

    Counter () {
        counter = new Thread(this);
        i = N;
        counter.start();
    }
}
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
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
Thread Synchronisation

synchronized (obj) statements
synchronized typename methodname () statement

This provides \textit{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

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

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

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


Car Park Departures class

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) {} 
    }
}

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
Thread Lifecycle

```
new Thread()
  start()
  stop()
  yield()
  sleep()
  suspend()

Created
  Running
    Runnable
      Non-Runnable
        suspend()
        resume()
        stop()
        or run() returns
```

Terminated
Thread Lifecycle

- New Thread()
- Start()
- Created
- Stop()
- Running
- Sleep()
- Yield()
- Runnable
- Resume()
- Non-Runnable
- Suspends
- Alive
- IsAlive()
- Terminated
- Or run() returns
Thread Lifecycle
FSP for Thread Lifecycle

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