COMP31212: Concurrency and Process Algebra
Introduction to the Course and to FSP

David Rydeheard

Room KB2.111: email: David.Rydeheard@manchester.ac.uk
Acknowledgements

This course owes much to previous versions prepared by Alan Williams and Howard Barringer. Others involved in early versions were Donal Fellows and Peter Aczel.
Course materials

This course is based on the following textbook:

**Concurrency: State Models and Java Programs**  

We will follow both the presentation of topics and also the exercises in this book. The authors have produced an LTSA (Labelled Transition Systems Analysis) tool to allow us to animate models of concurrent systems and test models for various properties. The tool is available from the course website or from the authors’ website.

If you take this course, you will need (1) to have this book available and (2) to download and use the LTSA tool.

The course website is packed with supporting materials: slides, exercises, answers, notes, links, examples, etc. **Look at it and use it!**
Contents

Topic 1: Introduction
  General Background
  On Concurrency
  Examples
  Implementation

Topic 2: Modelling Processes with FSP - I
  Process Algebras
  Labelled Transition Systems
  FSP: Basic Elements
  Summary
Course Topics

1. Introduction and Overview
2. Process algebras: FSP and Modelling Processes
3. Properties: Safety, Liveness, Deadlock, Livelock and Fairness
4. Process Equivalence
5. Java Threads - implementing concurrent processes
6. Concurrency Patterns:
   - Mutual Exclusion
   - Monitors+Semaphores
   - Producers/Consumers
   - Readers/Writers
   - GUls
   - Termination
7. Revision
General Comments

- Some concepts familiar from previous courses on distributed system and on operating systems,
- We follow much of Magee and Kramer, including their modelling approach through the process algebra, FSP,
- Java used to illustrate and implement — but this is not a programming course
Supporting and Background Material

Books


LTSA: Magee and Kramer’s modelling and analysis tool,
Exercises: offline and in lectures,
Lecture Slides: in lectures and on course website,
Notes on FSP.

All exercises, notes, lecture slides, and other material are available from the course website (accessed from the syllabus page).
Assessment

Two-hour examination (answer 3 questions from 5). See previous years’ examinations for sample questions.
What is Concurrency?

Simple account: A set of sequential programs (processes) executed “in parallel”.

The notion of “parallel” needs careful analysis, as does the way that the programs communicate and/or share information. Some issues:

- threads of control
- multi-threading
- parallel processing
- multi-processing
- multi-tasking
- shared memory
- message-passing: point-to-point or broadcast, synchronous or asynchronous
What is Concurrency?

Key concepts:

- ‘Atomicity of actions’ and interference,
- Non-determinism,
- Hierarchical structure of concurrent systems?
- Difficulty of reasoning about concurrency, rules for reasoning and building correct systems.
Why use Concurrency?

- often a natural concept when systems consist of more than one process
- computationally, sequencing is often over specified, and concurrency is natural, e.g. in the development of algorithms
- performance issues:
  - increased speed of performance (e.g. using multiprocessor and multicore machines)
  - functionally: increased responsiveness and throughput
Why is concurrency hard?

- development of concurrent algorithms: problems, including difficulties with abstraction
- efficiency and performance
- in general, behaviour is non-deterministic: consequences for testing and simulation
- correctness:
  - multiple threads to follow but their interleaving - hence overall behaviour - is undetermined,
  - there are properties not present in sequential systems e.g. deadlock, livelock, fairness, liveness, etc.
Example: Mutual Exclusion

Design a basic control protocol to ensure two processes never execute some “critical” region of program together.

Is it OK? At first it looks like a possible solution - but it can deadlock (actually livelock).
This example indicates many of the problems in developing and reasoning about concurrent processes:

- the large number of possible traces of execution that may arise through the interleaving of actions, and considering whether we have analysed sufficient to establish correctness,
- behaviours not present in sequential programming, such as deadlock and livelock
- restricting access to resources and interference,
- fairness of interaction, etc.
Example: The Firing Squad Synchronisation Problem

- On the command “FIRE”, the chain of control units must mutually synchronise to fire each gun simultaneously.
- The control units must be identical and work for any size chain of artillery.

How to do it?
Example: The Firing Squad Synchronisation Problem - continued

This example illustrates further aspects that arise in the description and analysis of concurrent processes:

- this is an example of interaction via message passing (the previous example had interaction via shared memory),
- the problem is one of synchronization, which is a special case of consensus amongst processes.
Modelling Concurrency

We consider how to describe concurrent processes using a process algebra called FSP:

- representing the real world at a suitable level of abstraction?
- modelling before implementing
- model captures interesting aspects: concurrency
- animation
- analysis

- Model Description Language: FSP (Finite State Processes)
- Models: LTS (Labelled Transition Systems)
Example: Cruise Control System

- Does it do what we expect? Is it safe? How do we model and implement the required behaviour?
FSP: Animation

INPUTSPEED = ( engineOn -> CHECKSPEED ),
CHECKSPEED = ( speed -> CHECKSPEED
| engineOff -> INPUTSPEED ).
set Sensors = {engineOn, engineOff, on, off, resume, brake, accelerator}
set Engine = {engineOn, engineOff}
set Prompts = {clearSpeed, recordSpeed, enableControl, disableControl}
Implementation in Java

- **Thread** class; **Runnable** interface
- starting, stopping, suspending threads
- mutual exclusion: *synchronized* methods and code blocks
- monitors, condition synchronization
- **wait**, **notify**, **notifyAll**
- **sleep**, **interrupt**
- **suspend**, **resume**, **stop**
- properties: safety, liveness
Process algebras

A process algebra is a notation and calculus for describing concurrent systems, based on:

- two types of elements, processes and actions, and the construct $a.P$ (otherwise written as $a \rightarrow P$) meaning perform action $a$ and then become process $P$,
- a variety of process constructors, including $||$ for the parallel composition of processes,
- processes are defined by (recursive) equations,
- evaluation is via rewriting of expressions using rules,
- synchronisation and communication is via shared actions.

Can consider process definitions as specifications, as runnable prototypes, or as models.
There are several varieties of process algebra, depending on how communication is handled, whether data is included, exactly what process constructors are incorporated, etc.
Labelled Transition Systems

What is an LTS?:

\[
\text{LTS} = (S, A, \sigma, s_0)
\]

where

- \( S \) : set of states
- \( A \) : alphabet \( A \subseteq Act \)
- \( \sigma \) : transition relation \( \sigma \subseteq (S \times A \times S) \)
- \( s_0 \) : initial state \( s_0 \in S \)

\( Act \) is the set of transition labels or (as we shall call them) ‘actions’. We model concurrency so that these are atomic events which cannot be interrupted or interfered with, and so are the lowest level of granularity.
First Exercise

**DAY1:** A Day In the Life Of:

- Get up — action: up,
- then have a cup of tea — action: tea
- then work — action: work

★ LTS for DAY1? ★

**DAY2:** Now repeat the Day

★ LTS for DAY2? ★

**DAY3:** Now be able to choose coffee — action: coffee — instead of tea

★ LTS for DAY3? ★
FSP: A Textual Representation for LTS

FSP - Finite State Processes
What FSP constructs Aae required?

- sequencing of actions
- STOP
- process definition, with recursion
- choice

Later, we consider the concurrent/parallel running of processes as a program construct.
Action Prefix

If \( x \) is an action and \( P \) is a process then \((x\rightarrow P)\) describes a process that initially engages in the action \( x \) and then behaves exactly as described by \( P \).

\((\text{once}\rightarrow \text{STOP})\).

Convention:

- actions begin with a lower case letter
- PROCESS NAMES begin with an upper case letter
- \textbf{STOP} is a specially pre-defined FSP process name.

★ FSP for DAY1? ★
Process Definition

Basic form:

\[ \text{ProcId} = \text{process_expression} \]

The meaning of \text{ProcId} (‘process identifier’) will be given by the meaning of \text{process_expression}.
\text{ProcId} should start with an upper-case letter.

(more complex forms possible — see later...

★ FSP for DAY2? ★
Choice

- If $x$ and $y$ are actions then $(x \rightarrow P \mid y \rightarrow Q)$ describes a process which initially engages in either of the actions $x$ or $y$.
- After that the subsequent behaviour is described by
  - $P$ if the first action was $x$,
  - $Q$ if the first action was $y$.

★ FSP for DAY3? ★
Example: Various Switches

Repetitive behaviour is via recursion:
\[
\text{SWITCH} \ = \ \text{OFF}, \\
\text{OFF} \ = \ (\text{on} \rightarrow \text{ON}), \\
\text{ON} \ = \ (\text{off} \rightarrow \text{OFF}).
\]

Substituting to get a more concise definition:
\[
\text{SWITCH} \ = \ \text{OFF}, \\
\text{OFF} \ = \ (\text{on} \rightarrow \text{off} \rightarrow \text{OFF}).
\]

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

★ Are these FSP SWITCH definitions the same? ★
Example: Traffic Light

FSP model of a traffic light (USA sequence!):

\[ \text{TRAFFICLIGHT} = (\text{red} \rightarrow \text{amber} \rightarrow \text{green} \rightarrow \text{amber} \rightarrow \text{TRAFFICLIGHT}) \].

★ LTS generated using LTSA? ★

★ Trace? ★

red \rightarrow \text{amber} \rightarrow \text{green} \rightarrow \text{amber} \rightarrow \text{red} \rightarrow \text{amber} \rightarrow \text{green} \rightarrow \ldots
Example: Vending Machine

FSP model of a drinks machine:

\[
\text{DRINKS} = (\text{red} \rightarrow \text{coffee} \rightarrow \text{DRINKS} \\
\quad | \\
\quad \text{blue} \rightarrow \text{tea} \rightarrow \text{DRINKS})
\]

★ LTS generated using LTSA? ★

★ Possible traces? ★
Non-Deterministic Choice

Process \((x \rightarrow P \mid x \rightarrow Q)\) describes a process which engages in \(x\) and then behaves as either \(P\) or \(Q\).

\[
\begin{align*}
\text{COIN} & \quad = \quad (\text{toss} \rightarrow \text{HEADS} \mid \text{toss} \rightarrow \text{TAILS}), \\
\text{HEADS} & \quad = \quad (\text{heads} \rightarrow \text{COIN}), \\
\text{TAILS} & \quad = \quad (\text{tails} \rightarrow \text{COIN}).
\end{align*}
\]

Tossing a coin

★ Possible traces? ★
Indexed Processes and Actions

Single slot buffer that inputs a value in the range 0 to 3 and then outputs a value:

$$\text{BUFF} = (\text{in}[i : 0..3] \rightarrow \text{out}[i] \rightarrow \text{BUFF}).$$

equivalent to

$$\text{BUFF} = (\text{in}[0] \rightarrow \text{out}[0] \rightarrow \text{BUFF}$$
$$| \text{in}[1] \rightarrow \text{out}[1] \rightarrow \text{BUFF}$$
$$| \text{in}[2] \rightarrow \text{out}[2] \rightarrow \text{BUFF}$$
$$| \text{in}[3] \rightarrow \text{out}[3] \rightarrow \text{BUFF}$$
$$).$$
or using a constant and indexed process `BUFF[i]`:

\[
\begin{align*}
\text{const } N &= 3 \\
\text{BUFF} &= (\text{in}[i : 0..N] \rightarrow \text{BUFF}[i]), \\
\text{BUFF}[i : 0..N] &= (\text{out}[i] \rightarrow \text{BUFF}).
\end{align*}
\]

or using a process parameter with default value:

\[
\text{BUFF}(N = 3) = (\text{in}[i : 0..N] \rightarrow \text{out}[i] \rightarrow \text{BUFF}).
\]
Guarded Actions

The choice \((\text{when } B \ x \rightarrow P \mid y \rightarrow Q)\) describes a process that is like \((x \rightarrow P \mid y \rightarrow Q)\) except that the action \(x\) can only be chosen \textbf{when} the guard \(B\) is true.
Example: A Counter

\[
\begin{align*}
\text{COUNT}(N = 3) & = \text{COUNT}[0], \\
\text{COUNT}[i : 0..N] & = (\text{when } (i < N) \text{ inc} \rightarrow \text{COUNT}[i + 1] \\
& \quad | \text{when } (i > 0) \text{ dec} \rightarrow \text{COUNT}[i - 1] \\
& ).
\end{align*}
\]
Example: A Countdown Timer

A countdown timer which beeps after \( N \) ticks, or can be stopped.

\[
\begin{align*}
\text{COUNTDOWN}(N = 3) &= (\text{start} \to \text{COUNTDOWN}[N]), \\
\text{COUNTDOWN}[i : 0..N] &= (\text{when } (i > 0) \text{ tick} \\
&\quad \to \text{COUNTDOWN}[i - 1] \\
&\quad | \text{when } (i == 0) \text{ beep} \to \text{STOP} \\
&\quad | \text{stop} \to \text{STOP}
\end{align*}
\]

★ LTS? ★
Example: What is this?

★ What is the following FSP process equivalent to? ★

const False = 0

P = (when (False) doanything -> P).
Constant and Range Declarations

Index expressions to model a calculation:

const N = 1
range T = 0..N
range R = 0..2*N

SUM = (in[a:T][b:T] -> TOTAL[a+b]),
TOTAL[s:R] = (out[s] -> SUM).

★ Write SUM using basic FSP? ★
Process Alphabets

- The alphabet of a process is the set of actions in which it is allowed to engage.
- This is usually determined implicitly as the actions in which it can engage.
- But the implicit alphabet can be extended:

\[
\text{WRITER} = (\text{write}[1] \rightarrow \text{write}[3] \rightarrow \text{WRITER}) + \{\text{write}[0..3]\}.
\]

The alphabet of \text{WRITER} is the set \{\text{write}[0..3]\}; i.e. the set \{\text{write}[0], \text{write}[1], \text{write}[2], \text{write}[3]\}.
# FSP: Summary

<table>
<thead>
<tr>
<th>Forms of process expression</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix action</td>
<td>(coffee-&gt;DRINKS)</td>
</tr>
<tr>
<td>guarded action</td>
<td>(when (i == 0) beep-&gt;STOP)</td>
</tr>
<tr>
<td>deterministic choice</td>
<td>(red-&gt;COFFEE</td>
</tr>
<tr>
<td>non-deterministic choice</td>
<td>(toss-&gt;HEADS</td>
</tr>
<tr>
<td>dependent process</td>
<td>(out[i]-&gt;BUFF)</td>
</tr>
<tr>
<td>indexed choice</td>
<td>(in[i : 0..3]-&gt;BUFF[i])</td>
</tr>
<tr>
<td>process name</td>
<td>{ DRINKS, BUFF[i] }</td>
</tr>
</tbody>
</table>
Process equation: \[ \text{process} \_\text{name} = \text{process} \_\text{expression} \]

Process definition:
\[
\begin{align*}
\text{declarations} \\
\text{main} \_\text{process} \_\text{equation}, \\
\text{local} \_\text{process} \_\text{equation}, \\
\vdots \\
\text{local} \_\text{process} \_\text{equation}, \\
\text{local} \_\text{process} \_\text{equation}.
\end{align*}
\]