Chapter 3

Concurrent Execution
Concurrent execution

Concepts: processes - concurrent execution and interleaving.

Models: parallel composition of asynchronous processes
- interleaving
interaction - shared actions
process labelling, and action relabelling and hiding
structure diagrams

Practice: Multithreaded Java programs
Definitions

◆ **Concurrency**
  - *Logically* simultaneous processing.
  - Does not imply multiple processing elements (PEs). Requires interleaved execution on a single PE.

◆ **Parallelism**
  - *Physically* simultaneous processing.
  - Involves multiple PEs and/or independent device operations.

Both concurrency and parallelism require controlled access to shared resources. We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-concurrent execution.
3.1 Modeling Concurrency

How should we model process execution speed?
- arbitrary speed
  (we abstract away time)

How do we model concurrency?
- arbitrary relative order of actions from different processes
  (interleaving but preservation of each process order)

What is the result?
- provides a general model independent of scheduling
  (asynchronous model of execution)
parallel composition - action interleaving

If $P$ and $Q$ are processes then $(P||Q)$ represents the concurrent execution of $P$ and $Q$. The operator $||$ is the parallel composition operator.

\[
\text{ITCH} = (\text{scratch} \rightarrow \text{STOP}). \\
\text{CONVERSE} = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}). \\
||\text{CONVERSE}_\text{ITCH} = (\text{ITCH} || \text{CONVERSE}).
\]

Possible traces as a result of action interleaving:

- think$\rightarrow$talk$\rightarrow$scratch
- think$\rightarrow$scratch$\rightarrow$talk
- scratch$\rightarrow$think$\rightarrow$talk

Disjoint alphabets
parallel composition - action interleaving

ITCH

2 states

CONVERSE

3 states

ITCH

2 states

CONVERSE

3 states

CONVERSE_ITCH

2 x 3 states

from ITCH

from CONVERSE
parallel composition - algebraic laws

**Commutative:** \((P||Q) = (Q||P)\)

**Associative:** \((P||(Q||R)) = ((P||Q)||R) = (P||Q||R)\).

Clock radio example:

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

*LTS? Traces? Number of states?*
If processes in a composition have actions in common, these actions are said to be **shared**. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

\[
\text{MAKER} = (\text{make} \to \text{ready} \to \text{MAKER}). \\
\text{USER} = (\text{ready} \to \text{use} \to \text{USER}). \\
\text{MAKER_USER} = (\text{MAKER} \parallel \text{USER}).
\]

**LTS? Traces? Number of states?**

2015 Concurrency: concurrent execution
modelling interaction - handshake

A handshake is an action acknowledged by another:

\[
\text{MAKER}_v2 = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}_v2).
\]

\[
\text{USER}_v2 = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}_v2).
\]

\[
| \ | \text{MAKER}\_\text{USER}_v2 = (\text{MAKER}_v2 \ | \ | \text{USER}_v2).
\]

Interaction constrains the overall behaviour.

3 states
3 states
3 x 3 states?

4 states
Modelling interaction - multiple processes

Multi-party synchronization:

\[
\text{MAKE}_A = (\text{makeA} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_A).
\]

\[
\text{MAKE}_B = (\text{makeB} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_B).
\]

\[
\text{ASSEMBLE} = (\text{ready} \rightarrow \text{assemble} \rightarrow \text{used} \rightarrow \text{ASSEMBLE}).
\]

\[
\text{FACTORY} = (\text{MAKE}_A \mid \mid \text{MAKE}_B \mid \mid \text{ASSEMBLE}).
\]
composite processes

A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

\[
||\text{MAKERS} = (\text{MAKE}_A \mid\mid \text{MAKE}_B) .
\]

\[
||\text{FACTORY} = (\text{MAKERS} \mid\mid \text{ASSEMBLE}) .
\]

Substituting the definition for \text{MAKERS} in \text{FACTORY} and applying the \textbf{commutative} and \textbf{associative} laws for parallel composition results in the original definition for \text{FACTORY} in terms of primitive processes.

\[
||\text{FACTORY} = (\text{MAKE}_A \mid\mid \text{MAKE}_B \mid\mid \text{ASSEMBLE}) .
\]
Example: a roller coaster model

A roller coaster control system only permits its car to depart when it is full.

Passengers arriving at the departure platform are registered by a turnstile. The controller signals the car to depart when there are enough passengers on the platform to fill the car to its maximum capacity of $M$ passengers. The car then goes around the roller coaster track and then waits for another $M$ passengers. A maximum of $M$ passengers may occupy the platform.

The roller coaster consists of three interacting processes TURNSTILE, CONTROL and CAR.
Example: an abstract roller coaster model

\[
\text{const } M = 3
\]

//turnstile simulates passenger arrival
\[
\text{TURNSTILE} = ( \ldots \rightarrow \text{TURNSTILE}).
\]

//control counts passengers and signals when full
\[
\text{CONTROL} = \text{CONTROL}[0],
\]
\[
\text{CONTROL}[i:0..M] = ( \ldots \rightarrow \text{CONTROL}[i+1]
\]
\[
\mid \ldots \rightarrow \text{CONTROL}[0]
\]
\[
).\]

//car departs when signalled
\[
\text{CAR} = ( \ldots \rightarrow \text{CAR}).
\]

||\text{ROLLERCOASTER} = ( \ldots ).\]
process instances and labelling

a:P creates an instance of process P and prefixes each action label in the alphabet of P with a.

Two instances of a switch process:

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

An array of instances of the switch process:

\[
||\text{SWITCHES(N=3)} = (\forall i:1..N \ s[i]:\text{SWITCH}).
\]
\[
||\text{SWITCHES(N=3)} = (s[i:1..N]:\text{SWITCH}).
\]
Relabelling functions are applied to processes to change the names of action labels. The general form of the relabeling 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.

CLIENT = (\text{call} \rightarrow \text{wait} \rightarrow \text{continue} \rightarrow \text{CLIENT}).
SERVER = (\text{request} \rightarrow \text{service} \rightarrow \text{reply} \rightarrow \text{SERVER}).

Note that both newlabel and oldlabel can be sets of labels.
action relabelling

||CLIENT_SERVER = (CLIENT || SERVER)

/{call/request, reply/wait}. 
Process prefixing is useful for modelling **shared** resources:

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

\[
\text{||RESOURCE\_SHARE} = (a:USER \mid \mid b:USER \mid \mid \{a,b\}::\text{RESOURCE}) .
\]
How does the model ensure that the user that acquires the resource is the one to release it?

Can this be achieved using relabelling rather than sharing? How?
An alternative formulation of the client server system is described below using qualified or prefixed labels:

\[
\text{SERVER}_v2 = (\text{accept}. \text{request} \\
\quad \rightarrow \text{service} \\
\quad \rightarrow \text{accept}. \text{reply} \\
\rightarrow \text{SERVER}_v2).
\]

\[
\text{CLIENT}_v2 = (\text{call}. \text{request} \\
\quad \rightarrow \text{call}. \text{reply} \\
\rightarrow \text{continue} \\
\rightarrow \text{CLIENT}_v2).
\]

\[
| | \text{CLIENT}_v2 \text{SERVER}_v2 = (\text{CLIENT}_v2 | | \text{SERVER}_v2) \\
/\{\text{call/accept}\}.
\]
action hiding - abstraction to reduce complexity

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 labeled $\tau$. 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 actions in the alphabet of $P$ not labeled in the set $a_1..a_x$. 
action hiding

The following definitions are equivalent:

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

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

\textbf{Minimization} removes hidden \texttt{tau} actions to produce an LTS with equivalent observable behaviour.
Structure diagrams – systems as interacting processes

Process P with alphabet \{a,b\}.

Parallel Composition
\((P \parallel Q) / \{m/a,m/b,c/d\}\)

Composite process
\(||S = (P \parallel Q) @ \{x,y\}\)
structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators: *parallel composition*, *relabeling* and *hiding*.

\[
\text{range } T = 0..3 \\
\text{BUFF} = (\text{in}[i:T] \rightarrow \text{out}[i] \rightarrow \text{BUFF}). \\
||\text{TWOBUF} = ?
\]
structure diagrams

Structure diagram for CLIENT_SERVER

Structure diagram for CLIENT_SERVERv2
RESOURCE = (acquire->release->RESOURCE).
USER = (printer.acquire->use
         ->printer.release->USER)\{use\}.

||PRINTER_SHARE
   = (a:USER| |b:USER| |{a,b}::printer:RESOURCE).
3.2 Multi-threaded Programs in Java

Concurrency in Java occurs when more than one thread is alive. ThreadDemo has two threads which rotate displays.
ThreadDemo model

ROTATOR = PAUSED,
PAUSED = (run->RUN | pause->PAUSED | interrupt->STOP),
RUN = (pause->PAUSED | {run,rotate}->RUN | interrupt->STOP).

||THREAD_DEMO = (a:ROTATOR || b:ROTATOR)
/{stop/{a,b}.interrupt}.

Interpret run, pause, interrupt as inputs, rotate as an output.
ThreadDemo implementation in Java - class diagram

ThreadDemo creates two ThreadPanel displays when initialized. ThreadPanel manages the display and control buttons, and delegates calls to rotate() to DisplayThread. Rotator implements the runnable interface.
Rotator class

```java
class Rotator implements Runnable {
    public void run() {
        try {
            while(true) ThreadPanel.rotate();
        } catch(InterruptedException e) {} 
    }
}
```

Rotator implements the runnable interface, calling ThreadPanel.rotate() to move the display.

run() finishes if an exception is raised by Thread.interrupt().
ThreadPanel class

public class ThreadPanel extends Panel {

    // construct display with title and segment color c
    public ThreadPanel(String title, Color c) {...}

    // rotate display of currently running thread 6 degrees
    // return value not used in this example
    public static boolean rotate() throws InterruptedException {...}

    // create a new thread with target r and start it running
    public void start(Runnable r) {
        thread = new DisplayThread(canvas,r,...);
        thread.start();
    }

    // stop the thread using Thread.interrupt()
    public void stop() {thread.interrupt();}
}

ThreadPanel manages the display and control buttons for a thread.

Calls to rotate() are delegated to DisplayThread.

Threads are created and started by the start() method, and terminated by the stop() method.
ThreadDemo class

```java
public class ThreadDemo extends Applet {
    ThreadPanel A; ThreadPanel B;

    public void init() {
        A = new ThreadPanel("Thread A",Color.blue);
        B = new ThreadPanel("Thread B",Color.blue);
        add(A); add(B);
    }

    public void start() {
        A.start(new Rotator());
        B.start(new Rotator());
    }

    public void stop() {
        A.stop();
        B.stop();
    }
}
```

ThreadDemo creates two ThreadPanel displays when initialized and two threads when started. ThreadPanel is used extensively in later demonstration programs.
## 3.3 Java Concurrency Utilities Package

Java SE 5 introduced a package of advanced concurrency utilities in `java.util.concurrent` (more later). This was extended in Java SE 7 to include additional constructs to separate thread creation and management from the rest of the application using `executors`, `thread pools`, and `fork/join`.

| Executor interface: | replacement for thread creation, usually using existing thread: 
replace `(new Thread(r)).start();` runnable object `r`
with `e.execute(r);` Executor object `e` |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecutorService:</td>
<td>manage termination; return a <code>Future</code> for tracking thread status</td>
</tr>
</tbody>
</table>
| Thread Pools:       | used to minimize the overhead of thread creation /termination 
`ExecutorService newFixedThreadPool(int nThreads)` |
|                     | - creates a fixed number with at most `nThreads` active threads 
- tasks are allocated from a shared unbounded queue |
| Fork/Join:          | for recursive decomposition of tasks using thread pools |
Summary

◆ Concepts
  ● concurrent processes and process interaction

◆ Models
  ● Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  ● Parallel composition as a finite state process with action interleaving.
  ● Process interaction by shared actions.
  ● Process labeling and action relabeling and hiding.
  ● Structure diagrams

◆ Practice
  ● Multiple threads in Java.