Chapter 2

Processes & Threads
We structure complex systems as sets of simpler activities, each represented as a *sequential process*. Processes can overlap or be concurrent, so as to reflect the concurrency inherent in the physical world, or to offload time-consuming tasks, or to manage communications or other devices.

Designing concurrent software can be complex and error prone. A rigorous engineering approach is essential.

**Concept of a process as a sequence of actions.**

**Model processes as finite state machines.**

**Program processes as threads in Java.**
**Concepts:** processes - units of sequential execution.

**Models:** finite state processes (FSP) to model processes as sequences of actions. labelled transition systems (LTS) to analyse, display and animate behavior.

**Practice:** Java threads
2.1 Modeling Processes

Models are described using state machines, known as Labelled Transition Systems \textbf{LTS}. These are described textually as finite state processes (\textbf{FSP}) and displayed and analysed by the \textbf{LTSA} analysis tool.

- \textbf{LTS} - graphical form
- \textbf{FSP} - algebraic form
modeling processes

A process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions.

Can finite state models produce infinite traces?
If \( \mathbf{x} \) is an action and \( \mathbf{P} \) a process then \((\mathbf{x} \rightarrow \mathbf{P})\) describes a process that initially engages in the action \( \mathbf{x} \) and then behaves exactly as described by \( \mathbf{P} \).

\[
\text{ONESHOT} = (\text{once} \rightarrow \text{STOP}).
\]

**ONESHOT state machine**
(terminating process)

**Convention:** actions begin with lowercase letters

PROCESSES begin with uppercase letters
Repetitive behaviour uses recursion:

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

Substituting to get a more succinct definition:

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

And again:

\[
\text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}).
\]
The LTSA animator can be used to produce a trace.

Ticked actions are eligible for selection.

In the LTS, the last action is highlighted in red.
FSP - action prefix

FSP model of a traffic light:

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

LTS generated using \text{LTSA}:

Trace:

\[ \text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{orange} \rightarrow \text{red} \rightarrow \text{orange} \rightarrow \text{green} \ldots \]
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 the first action has occurred, the subsequent behavior is described by $P$ if the first action was $x$ and $Q$ if the first action was $y$.

Who or what makes the choice?

Is there a difference between input and output actions?
FSP - choice

FSP model of a drinks machine:

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

LTS generated using \textit{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 \(Q\).

\[
\text{COIN} = (\text{toss} \rightarrow \text{HEADS} \mid \text{toss} \rightarrow \text{TAILS}), \\
\text{HEADS} = (\text{heads} \rightarrow \text{COIN}), \\
\text{TAILS} = (\text{tails} \rightarrow \text{COIN}).
\]

Tossing a coin.

Possible traces?
Modeling failure

How do we model an unreliable communication channel which accepts in actions and if a failure occurs produces no output, otherwise performs an out action?

Use non-determinism...

\[
\text{CHAN} = (\text{in} \to \text{CHAN} \mid \text{in} \to \text{out} \to \text{CHAN})
\]
FSP - indexed processes and actions

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

\[ \text{BUFF}(N=3) = (\text{in}[i:0..N] \rightarrow \text{out}[i] \rightarrow \text{BUFF}) . \]
FSP - indexed processes and actions

Local indexed process definitions are equivalent to process definitions for each index value.

Index expressions to model calculation:

\[
\begin{align*}
\text{const } N &= 1 \\
\text{range } T &= 0..N \\
\text{range } R &= 0..2*N \\
\text{SUM} &= (\text{in}[a:T][b:T]\rightarrow\text{TOTAL}[a+b]), \\
\text{TOTAL}[s:R] &= (\text{out}[s]\rightarrow\text{SUM}).
\end{align*}
\]
The choice \texttt{(when B x -> P | y -> Q)} means that when the guard B is true then the actions x and y are both eligible to be chosen, otherwise if B is false then the action x cannot be chosen.

\texttt{COUNT (N=3) = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->\texttt{COUNT[i+1]})
| when(i>0) dec->\texttt{COUNT[i-1]})}.

0 \quad 1 \quad 2 \quad 3

\begin{align*}
\text{inc} & \quad \text{dec} \\
\text{inc} & \quad \text{dec} \\
\text{inc} & \quad \text{dec}
\end{align*}
FSP - guarded actions

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

\[
\text{COUNTDOWN (N=3)} = (\text{start} \rightarrow \text{COUNTDOWN}[N]), \\
\text{COUNTDOWN}[i:0..N] = \\
\quad (\text{when}(i>0) \text{ tick} \rightarrow \text{COUNTDOWN}[i-1] \\
\quad \mid \text{when}(i==0) \text{ beep} \rightarrow \text{STOP} \\
\quad \mid \text{stop} \rightarrow \text{STOP})
\]

Concurrent actions:
- \text{start}
- \text{tick}
- \text{beep}
- \text{stop}
What is the following FSP process equivalent to?

\[
\text{const False } = 0 \\
P = (\text{when (False) doanything->P}).
\]

Answer:

STOP
FSP - process alphabets

The alphabet of a process is the set of actions in which it can engage.

Process alphabets are implicitly defined by the actions in the process definition.

The alphabet of a process can be displayed using the LTSA alphabet window.

Process: COUNTDOWN
Alphabet:
{ beep, start, stop, tick }
FSP - process alphabet extension

Alphabet extension can be used to extend the implicit alphabet of a process:

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

Alphabet of \text{WRITER} is the set \{\text{write}[0..3]\}
(we make use of alphabet extensions in later chapters)
Revision & Wake-up Exercise

In FSP, model a process **FILTER**, that exhibits the following repetitive behavior:

inputs a value \( v \) between 0 and 5, but only outputs it if \( v \leq 2 \), otherwise it discards it.

\[
\begin{align*}
\text{FILTER} &= (\text{in}[v:0..5] \rightarrow \text{DECIDE}[v]), \\
\text{DECIDE}[v:0..5] &= (\ ? \ ).
\end{align*}
\]
2.2 Implementing processes

Modeling processes as finite state machines using FSP/LTS.

Implementing threads in Java.

Note: to avoid confusion, we use the term process when referring to the models, and thread when referring to the implementation in Java.
A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) threads of control, it has multiple stacks, one for each thread.
**threads in Java**

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.

The Thread class executes instructions from its method `run()`. The actual code executed depends on the implementation provided for `run()` in a derived class.

```java
class MyThread extends Thread {
    public void run() {
        //......
    }
}
```

Creating a thread object:

```java
Thread a = new MyThread();
```
threads in Java

Since Java does not permit multiple inheritance, we often implement the `run()` method in a class not derived from Thread but from the interface Runnable.

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

class MyRun implements Runnable{
    public void run() {
        //....
    }
}

Creating a thread object:

```
Thread b = new Thread(new MyRun());
```

thread life-cycle in Java

An overview of the life-cycle of a thread as state transitions:

- **Created**
  - `new Thread()`
  - `start()` causes the thread to call its `run()` method.
- **Alive**
  - `start()`
  - `stop()`, or `run()` returns
- **Terminated**

The predicate `isAlive()` can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted (cf. mortals).
thread alive states in Java

Once started, an alive thread has a number of substates:

- Runnable
- Non-Runnable

Also, `wait()` makes a Thread Non-Runnable, and `notify()` makes it Runnable (used in later chapters).
Java thread lifecycle - an FSP specification

<table>
<thead>
<tr>
<th>THREAD</th>
<th>= CREATED,</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATED</td>
<td>= (start</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>RUNNING</td>
<td>= ({suspend,sleep}--&gt;NON_RUNNABLE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>RUNNABLE</td>
<td>= (suspend</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NON_RUNNABLE</td>
<td>= (resume</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>TERMINATED</td>
<td>= STOP.</td>
</tr>
</tbody>
</table>
end, run, dispatch are not methods of class Thread.

States 0 to 4 correspond to CREATED, TERMINATED, Runnable, Running, and NON-RUNNABLE respectively.
CountDown timer example

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

Implementation in Java?
The class `CountDown` derives from `Applet` and contains the implementation of the `run()` method which is required by `Thread`.

Concurrency: processes & threads
CountDown class

```java
public class CountDown extends Applet
    implements Runnable {

    Thread counter; int i;
    final static int N = 10;
    AudioClip beepSound, tickSound;
    NumberCanvas display;

    public void init() {...}
    public void start() {...}
    public void stop() {...}
    public void run() {...}
    private void tick() {...}
    private void beep() {...}
}
```
CountDown class - start(), stop() and run()

```java
public void start() {
    counter = new Thread(this);
    i = N; counter.start();
}

public void stop() {
    counter = null;
}

public void run() {
    while(true) {
        if (counter == null) return;
        if (i>0) { tick(); --i; }
        if (i==0) { beep(); return;}
    }
}
```

**COUNTDOWN Model**

**start** -> CD[3]

**run** -> CD[i:0..3] =
(while (i>0) tick -> CD[i-1] |when (i==0) beep -> STOP).

**STOP** -> [predefined in FSP to end a process]

**CD[i] process**

- **recursion** transformed into **while** loop

**STOP** when run() returns
CountDown execution

```
countdown.start()
```

- **CountDown**
  - `start()`
  - `new Thread(this)`
  - `counter.start()`
  - `target.run()`

- **Counter thread**
  - `init()`
  - `created`
  - `tick()`
  - `alive`
  - `beep()`
  - `terminated`
CountDown

start()

new Thread(this)

counter.start()

tick()

tick()

tick()

tick()

target.run()

stop()

counter=null

counter thread

created

alive

terminated

Concurrency: processes & threads

©Magee/Kramer 2nd Edition
Summary

◆ Concepts
  ● process - unit of concurrency, execution of a program

◆ Models
  ● LTS to model processes as state machines - sequences of atomic actions
  ● FSP to specify processes using prefix “- >”, choice “ | ” and recursion.

◆ Practice
  ● Java threads to implement processes.
  ● Thread lifecycle - created, running, runnable, non-runnable, terminated.