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.**
processes and threads

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 LTS. These are described textually as finite state processes (FSP) and displayed and analysed by the LTSA analysis tool.

♦ LTS - graphical form
♦ 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.

A light switch

\[ \text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \text{off} \rightarrow \ldots \]

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

**ONESHOT** = (once $\rightarrow$ 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:

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

LTS generated using 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 \((x-> P \mid y-> Q)\) 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 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 \texttt{in} actions and if a failure occurs produces no output, otherwise performs an \texttt{out} action?

Use non-determinism...

\[
\text{CHAN} = (\text{in} \rightarrow \text{CHAN} \\
\mid \text{in} \rightarrow \text{out} \rightarrow \text{CHAN})
\]
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 \text{process parameter} with default value:

\[
\text{BUFF}(N=3) = (\text{in}[i:0..N] \rightarrow \text{out}[i] \rightarrow \text{BUFF}).
\]
FSP - constant & range declaration

index expressions to model 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).
FSP - guarded actions

The choice \((\text{when } B \ x \to \ P \ | \ y \to \ Q)\) 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.

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

\begin{center}
\begin{tikzpicture}

\node[fill=red] (0) at (0,0) {0};
\node[fill=blue] (1) at (2,0) {1};
\node[fill=green] (2) at (4,0) {2};
\node[fill=blue!50!cyan] (3) at (6,0) {3};

\draw[->] (0) to [out=180,in=90] node [above] {inc} (1);
\draw[->] (1) to [out=0,in=90] node [above] {inc} (2);
\draw[->] (2) to [out=0,in=90] node [above] {inc} (3);
\draw[->] (3) to [out=0,in=90] node [above] {inc} (0);
\draw[<->] (0) to node [below] {dec} (3);
\end{tikzpicture}
\end{center}
FSP - guarded actions

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

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

What is the following FSP process equivalent to?

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

**Answer:**

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

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 \textbf{WRITER} is the set \{\text{write}[0..3]\} (we make use of alphabet extensions in later chapters)
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) \textbf{threads of control}, it has multiple stacks, one for each thread.
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.

class MyThread extends Thread {
    public void run() {
        //......
    }
}
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() {
        //.....
    }
}
```
**thread life-cycle in Java**

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

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**
  - **start()**
  - **yield()**
  - **dispatch**
  - **suspend()**

- **Non-Runnable**
  - **stop()**, or **run()** returns
  - **resume()**

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

| THREAD      | CREATED, |
|            | |
| CREATED    | (start          ->RUNNING |
|            | stop            ->TERMINATED), |
| RUNNING    | ({suspend,sleep}->NON_RUNNABLE |
|            | yield          ->RUNNABLE |
|            | {stop,end}     ->TERMINATED |
|            | run            ->RUNNING), |
| RUNNABLE   | (suspend      ->NON_RUNNABLE |
|            | dispatch       ->RUNNING |
|            | stop          ->TERMINATED), |
| NON_RUNNABLE | (resume     ->RUNNABLE |
|            | stop          ->TERMINATED), |
| TERMINATED | STOP.        |
end, run, dispatch are not methods of class Thread.

States 0 to 4 correspond to **CREATED, TERMINATED, RUNNING, NON–RUNNABLE**, and **RUNNABLE** respectively.

Concurrency: processes & threads
CountDown timer example

COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
   (when(i>0) tick->COUNTDOWN[i-1]
   | when(i==0) beep->STOP
   | stop->STOP
   )

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

The class **NumberCanvas** provides the display canvas.
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() {...}
}

©Magee/Kramer
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 →

stop →

COUNTDOWN[i] process
recursion as a while loop
STOP
when(i>0) tick → CD[i-1]
when(i==0)beep → STOP

STOP when run() returns
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.