2. Java and Concurrency

Prof. O. Nierstrasz

Selected material © Magee and Kramer
Roadmap

> Modelling Concurrency
  — Finite State Processes
  — Labelled Transition Systems

> Java Threads
  — Thread creation
  — Thread lifecycle

> Java synchronization
  — wait and notify
Roadmap

> Modelling Concurrency
  — Finite State Processes
  — Labelled Transition Systems

> Java Threads
  — Thread creation
  — Thread lifecycle

> Java synchronization
  — wait and notify
Because concurrent systems are *non-deterministic*, it can be difficult to build them and reason about their properties.

A **model** is an *abstraction of the real world* that makes it easier to focus on the points of interest.

**Approach:**

Model concurrent systems as *sets of sequential finite state processes*
Finite State Processes

**FSP** is a *textual notation* for specifying a finite state process:

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

**LTS** is a *graphical notation* for interpreting a process as a labelled transition system:

The *meaning* of a process is a set of possible traces:

\[
\text{on } \rightarrow \text{ off } \rightarrow \text{ on } \rightarrow \text{ off } \rightarrow \text{ on } \rightarrow \text{ off } \rightarrow \text{ on}.\]
# FSP Summary

| **Action prefix** | (x→P)          | **Parallel composition** | (P||Q)          |
|-------------------|----------------|--------------------------|-----------------|
| **Choice**        | (x→P|y→Q)      | **Replicator**           | forall [I:1..N] P(I) |
| **Guarded Action**| (when B x→P|y→Q) | **Process labelling**    | a:P             |
| **Alphabet extension** | P + S        | **Process sharing**      | {a₁,…,aₙ}::P     |
| **Conditional**   | If B then P else Q | **Priority High**         | ||C=(P||Q)<<{a₁,…,aₙ} |
| **Relabelling**   | /{new₁/old₁,…} | **Priority Low**          | ||C=(P||Q)>>{a₁,…,aₙ} |
| **Hiding**        | \{a₁,…,aₙ}    | **Safety property**      | property P      |
| **Interface**     | @{a₁,…,aₙ}    | **Progress property**    | progress P = {a₁,…,aₙ} |

We will encounter and use these features in the lectures to come …
If $x$ is an action and $P$ a process then $(x \rightarrow P)$ is a process that initially engages in the action $x$ and then behaves like $P$.

**ONESHOT** = $(\text{once} \rightarrow \text{STOP})$.

**Convention:**
Processes start with **UPPERCASE**, actions start with **lowercase**.
Repetitive behaviour uses recursion:

\[
\begin{align*}
  \text{SWITCH} &= \text{OFF}, \\
  \text{OFF} &= (\text{on} \rightarrow \text{ON}), \\
  \text{ON} &= (\text{off} \rightarrow \text{OFF}).
\end{align*}
\]
If $x$ and $y$ are actions then $(x \rightarrow P \mid y \rightarrow Q)$ is a process which initially engages in *either* of the actions $x$ or $y$.

If $x$ occurs, the process then behaves like $P$; otherwise, if $y$ occurs, it behaves like $Q$.

**DRINKS** =

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

**What are the possible traces of DRINKS?**
FSP — Non-determinism

\((x \to P \mid x \to Q)\) performs \(x\) and then behaves as either \(P\) or \(Q\).

\[\text{COIN} = (\text{toss} \to \text{heads} \to \text{COIN} \mid \text{toss} \to \text{tails} \to \text{COIN})\]
FSP — Guarded actions

(when \( B \ x \rightarrow P \ \mid\ y \rightarrow Q \)) means that when the guard \( B \) is true then either \( x \) or \( y \) may be chosen; otherwise if \( B \) is false then only \( y \) may be chosen. (default case is optional)

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

> Modelling Concurrency
  — Finite State Processes
  — Labelled Transition Systems

> Java Threads
  — Thread creation
  — Thread lifecycle

> Java synchronization
  — wait and notify
Java

> Concurrency model based on *monitors*
  — synchronized keyword
  — wait() and notify() methods
  — Thread class and Runnable interface

> java.util.concurrent package (Java 1.5)
  — Implements many common concurrency idioms
A Java Thread has a run method defining its behaviour:

```
class SimpleThread extends Thread {
    public static void main (String[] args) { ... }
    public SimpleThread(String str) {
        super(str);    // Call Thread constructor
    }
    public void run() {   // What the thread does
        for (int i=0; i<5; i++) {
            System.out.println(i + " " + getName());
            try { sleep((int)(Math.random()*1000));
            } catch (InterruptedException e) { } }
        System.out.println("DONE! " + getName());
    }
}
```
SimpleThread can be modelled as a single, sequential, finite state process:

\[
\text{Simple} = ([1] \rightarrow [2] \rightarrow [3] \rightarrow [4] \rightarrow \text{done} \rightarrow \text{STOP}).
\]

Or, more generically:

\[
\begin{align*}
\text{const } N &= 5 \\
\text{Simple} &= \text{Print}[1], \\
\text{Print}[n:1..N] &= \left( \text{when}(n<N) \ [n] \rightarrow \text{Print}[n+1] \right. \\
&\hspace{1cm} \left. \mid \text{when}(n==N) \ \text{done} \rightarrow \text{STOP} \right).
\end{align*}
\]
Multiple Threads ...

A Thread’s run method is never called directly but is executed when the Thread is *started*:

```java
class SimpleThread {
    public static void main (String[] args) {
        // Instantiate a Thread, then start it:
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
    ...
}
```
Running the TwoThreadsDemo

In this implementation of Java, the execution of the two threads is interleaved.

This is not guaranteed for all implementations!

Why are the output lines never garbled?

0 Jamaica
0 Fiji
1 Jamaica
1 Fiji
2 Fiji
3 Fiji
2 Jamaica
4 Fiji
3 Jamaica
DONE! Fiji
4 Jamaica
DONE! Jamaica
We can *relabel* the transitions of *Simple* and concurrently *compose* two copies of it:

```
||TwoThreadsDemo = ( fiji:Simple
    || jamaica:Simple
).
```

What are all the possible traces?
FSP — Composition

If we restrict ourselves to two steps, the composition will have nine states:
The state space of two composed processes is (at most) the Cartesian product of the individual state spaces.
A Java thread can either *inherit* from java.lang.Thread, or *contain* a Runnable object:

```java
public class java.lang.Thread
    extends java.lang.Object
    implements java.lang.Runnable
{
    public Thread();
    public Thread(Runnable target);
    public Thread(Runnable target, String name);
    public Thread(String name);
    ...
```
A thread must be created, and then started:

```java
...
    public void run();
    public synchronized void start();
    public static void sleep(long millis)
        throws InterruptedException;
    public static void yield();
    public final String getName();
...
}
```

NB: `suspend()`, `resume()` and `stop()` are deprecated!
Since Java does not support multiple inheritance, it is impossible to inherit from both `Thread` and another class. Instead, simply define:

```java
class MyStuff extends UsefulStuff
    implements Runnable ...
```

and instantiate:

```java
new Thread(new MyStuff).start();
```
Transitions between Thread States

Thread

- start()
- sleep()
- wait()
- i/o block
- yield()
- time elapsed
- notify() / notifyAll()
- i/o completed
- run() exits
- Not Runnable
- Runnable
LTS for Threads

Thread = ( start -> Runnable ),
Runnable =
    ( yield -> Runnable
    | {sleep, wait, blockio} -> NotRunnable
    | stop -> STOP ),
NotRunnable =
    ( {awake, notify, unblockio} -> Runnable ).
Creating Threads

This Clock application uses a thread to update the time:

```java
public class Clock extends Canvas implements Runnable {
    private Thread clockThread = null;

    public Clock() {
        super();
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
}
```
Creating Threads ...

```java
... public void run() {
   // stops when clockThread is set to null
   while(Thread.currentThread() == clockThread) {
      repaint();
      try {clockThread.sleep(1000); }
      catch (InterruptedException e){ } }
...
... And stopping them

```java
... public void stopThread() {
    clockThread = null;
}

public void paintComponent(Graphics g) {
    ...
    String time = dateFormat.format(new Date());
    g2d.drawString(...);
}
...
Modelling Concurrency
- Finite State Processes
- Labelled Transition Systems

Java Threads
- Thread creation
- Thread lifecycle

Java synchronization
- wait and notify
Without synchronization, an arbitrary number of threads may run at any time within the methods of an object.

— Class invariant may not hold when a method starts!
— So can’t guarantee any post-condition!

A solution: consider a method to be a critical section which locks access to the object while it is running.

This works as long as methods cooperate in locking and unlocking access!
Synchronized methods

Either: declare an entire method to be synchronized with other synchronized methods of an object:

```java
public class PrintStream extends FilterOutputStream {
    ...
    public synchronized void println(String s);
    public synchronized void println(char c);
    ...
}
```
Synchronized blocks

Or: synchronize an individual block within a method with respect to some object:

```java
public Object aMethod() {
    // unsynchronized code
    ...
    synchronized(resource) { // lock resource
        ...
    } // unlock resource
    ...
}
```
wait and notify

Synchronization must sometimes be interrupted:

```java
class Account {
    protected int assets = 0;
    public synchronized void withdraw(int amount) {
        while (amount > assets) {
            try {
                wait();
            } catch(InterruptedException e) { }
        }
        assets -= amount;
    }
    public synchronized void deposit(int amount) {
        assets += amount;
        notifyAll();
    }
}
```

NB: you must either catch or throw InterruptedException
final Account myAccount = new Account();
new Thread() {
    public void run() {
        int amount = 100;
        System.out.println("Waiting to withdraw " + amount + " units ...");
        myAccount.withdraw(amount);
        System.out.println("I withdrew " + amount + " units!");
    }
}.start();
try { Thread.sleep(1000); } catch (InterruptedException e){ }
new Thread() {
    public void run() {
        int amount = 200;
        System.out.println("Depositing " + amount + " units ...");
        myAccount.deposit(amount);
        System.out.println("I deposited " + amount + " units!");
    }
}.start();

Waiting to withdraw 100 units ...
Depositing 200 units ...
I deposited 200 units!
I withdrew 100 units!
NB: `wait()` and `notify()` are methods rather than keywords:

```java
public class java.lang.Object {
    ...
    public final void wait()
        throws InterruptedException;
    public final void notify();
    public final void notifyAll();
    ...
}
```
What you should know!

- What are finite state processes?
- How are they used to model concurrency?
- What are traces, and what do they model?
- How can the same FSP have multiple traces?
- How do you create a new thread in Java?
- What states can a Java thread be in?
- How can it change state?
- What is the Runnable interface good for?
- What is a critical section?
- When should you declare a method to be synchronized?
Can you answer these questions?

> How would you specify an FSP that repeatedly performs hello, but may stop at any time?
> How many states and how many possible traces does the full TwoThreadsDemo FSP have?
> When should you inherit from Thread?
> How can concurrency invalidate a class invariant?
> What happens if you call wait or notify outside a synchronized method or block?
> When is it better to use synchronized blocks rather than methods?
> How would you model synchronization in FSP?
License

Attribution-ShareAlike 3.0 Unported

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.

http://creativecommons.org/licenses/by-sa/3.0/