Concurrent Execution

Alexandre David
adavid@cs.aau.dk

Credits for the slides:
Claus Brabrand
Jeff Magee & Jeff Kramer
Repetition

➢ **Concepts:** We adopt a model-based approach for the design and construction of concurrent programs.

➢ **Models:** We use finite state models to represent concurrent behaviour (Finite State Processes and Labelled Transition Systems).

➢ **Practice:** We use Java for constructing concurrent programs (and later C).
Repetition

Model = simplified representation of the real world.

➢ Based on Labelled Transition Systems (LTS)

*Focuses on concurrency aspects (of the program)*
- everything else abstracted away

Aka. *Finite State Machine* (FSM)

➢ Described textually as Finite State Processes (FSP)

$$\text{EngineOff} = (\text{engineOn} \rightarrow \text{EngineOn}), \quad \text{EngineOn} = (\text{engineOff} \rightarrow \text{EngineOff} \mid \text{speed} \rightarrow \text{EngineOn}).$$
# Repetition

- **Finite State Processes (FSP):**

<table>
<thead>
<tr>
<th>P</th>
<th>STOP</th>
<th>// termination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(x ↦ P)</td>
<td>// action prefix</td>
</tr>
<tr>
<td></td>
<td>(when (...) x ↦ P)</td>
<td>// guard</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>P’</td>
</tr>
<tr>
<td></td>
<td>P + { ... }</td>
<td>// alphabet extension</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>// process variable</td>
</tr>
</tbody>
</table>

- **action indexing**
  
  \[ x[i:1..N] \rightarrow P \text{ or } x[i] \rightarrow P \]

- **process parameters**

  \[ P[i:1..N] = \ldots \]

- **constant definitions**

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

- **range definitions**

  \[ \text{range } R = 0..N \]
Repetition

➢ Subclassing `java.lang.Thread`:

```java
class MyThread extends Thread {
    public void run() {
        // ...
    }
}
Thread t = new MyThread();
t.start();
// ...
```

➢ Implementing `java.langRunnable`:

```java
class MyRun implements Runnable {
    public void run() {
        // ...
    }
}
Thread t = new Thread(new MyRun());
t.start();
// ...
```
Concurrent Execution

**Concepts:** processes - concurrent execution and interleaving.
  process interaction.

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

**Practice:** multi-threaded Java programs
Definition: Parallelism

Parallelism (aka. “Real” Concurrent Execution)

- Physically simultaneous processing
- Involves multiple processing elements (PEs) and/or independent device operations
Definition: Concurrency

**Concurrency** (aka. *Pseudo-Concurrent Execution*)

- Logically simultaneous processing

- Does not imply multiple processing elements (PEs)

- Requires *interleaved* execution on a single PE
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).
How do we model concurrency?

Arbitrary relative order of actions from different processes – **interleaving** but preservation of each process order.
Modeling Concurrency

➢ How should we model process execution speed?

➢ We abstract away time: arbitrary speed!

-+: we can say nothing of real-time properties

+: independent of architecture, processor speed, scheduling policies, …
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.

$$ITCH = (\text{scratch} \rightarrow \text{STOP}).$$
$$CONVERSE = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}).$$

$$||CONVERSE_ITCH = (ITCH || CONVERSE).$$

Possible traces as a result of action interleaving?

• scratch $\rightarrow$ think $\rightarrow$ talk
• think $\rightarrow$ scratch $\rightarrow$ talk
• think $\rightarrow$ talk $\rightarrow$ scratch
Parallel Composition – Action Interleaving

Parallel composition =

ITCH

| scratch
| 0 |
| 1 |

CONVERSE

| think
| 0 |
| talk
| 1 |
| 2 |

2 states

3 states

Cartesian product =

from ITCH

from CONVERSE

2 x 3 states

2 x 3 states
Parallel Composition – Algebraic Laws

Commutative: \( (P \parallel Q) = (Q \parallel P) \)

Associative: \( (P \parallel (Q \parallel R)) = ((P \parallel Q) \parallel R) \)
\[= (P \parallel Q \parallel R). \]

Clock radio example:

\[
\text{CLOCK} = (\text{tick}\rightarrow\text{CLOCK}).
\]
\[
\text{RADIO} = (\text{on}\rightarrow\text{off}\rightarrow\text{RADIO}).
\]
\[
\text{||CLOCK\_RADIO} = (\text{CLOCK} \parallel \text{RADIO}).
\]

\text{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.
Modeling Interaction - Example

MAKE1 = (make->ready->STOP).
USE1 = (ready->use->STOP).

||MAKE1_USE1 = (MAKE1 || USE1).

3 states
3 states

3 x 3 states?
**Modeling Interaction - Example**

\[
\text{MAKE1} = (\text{make} \rightarrow \text{ready} \rightarrow \text{STOP}). \\
\text{USE1} = (\text{ready} \rightarrow \text{use} \rightarrow \text{STOP}). \\
\text{MAKE1}_\text{USE1} = (\text{MAKE1} || \text{USE1}).
\]

3 states
3 states
3 x 3 states?

No…!
**MAKE1** = (make->ready->STOP).
**USE1** = (ready->use->STOP).

\[ | | \text{MAKE1}_\text{USE1} = (\text{MAKE1} \lor \text{USE1}). \]

**Interaction constrains the overall behaviour.**

3 states
3 states

4 states!
MAKER = (make -> ready -> MAKER).
USER = (ready -> use -> USER).

||MAKER_USER = (MAKER || USER).
MAKER = (make->ready->MAKER).
USER = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
MAKER = (make→ready→MAKER).
USER = (ready→use→USER).

||MAKER_USER = (MAKER || USER).
MAKER = (make->ready->MAKER).
USER = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
Modeling Interaction - Example

\[
\text{MAKER} = (\text{make} \rightarrow \text{ready} \rightarrow \text{MAKER}). \\
\text{USER} = (\text{ready} \rightarrow \text{use} \rightarrow \text{USER}). \\
\text{||MAKER_USER} = (\text{MAKER} \mid \mid \text{USER}).
\]
MAKER = (make→ready→MAKER).
USER = (ready→use→USER).

||MAKER_USER = (MAKER || USER).
MAKER = (make->ready->MAKER).
USER = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
A handshake is an action acknowledged by another:

\[
\text{MAKER}_v^2 = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}_v^2).
\]

\[
\text{USER}_v^2 = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}_v^2).
\]

\[
\text{MAKE}_R \_ \text{USE}_R^v = (\text{MAKE}_R^v \mid \mid \text{USE}_R^v^2).
\]
**Multi-party synchronization:**

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

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

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

\[
|| \text{FACTORY} = (\text{MAKE}_A || \text{MAKE}_B || \text{ASSEMBLE}).
\]
A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

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

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

Substitution of def’n of \text{MAKERS}:

\[
\text{FACTORY} = ((\text{MAKE}_A \mid \mid \text{MAKE}_B) \mid \mid \text{ASSEMBLE}).
\]
A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

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

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

Substitution of definition of MAKERS

\[
\text{FACTORY} = ((\text{MAKE}_A \parallel \text{MAKE}_B) \parallel \text{ASSEMBLE}).
\]

Further simplification?

\[
\text{FACTORY} = (\text{MAKE}_A \parallel \text{MAKE}_B \parallel \text{ASSEMBLE}).
\]

Associativity!
**Process Labeling**

\[ \textcolor{red}{a:P} \] prefixes each action label in the alphabet of \( P \) with \( a \).

*Two instances* of a switch process:

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

\[
\text{||TWO_SWITCH} = \text{(a:SWITCH || b:SWITCH)}.
\]

**LTS?** *(a:SWITCH)*
**Process Labeling**

\( a: P \) prefixes each action label in the alphabet of \( P \) with \( a \).

**Two instances** of a switch process:

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

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

Diagram:

- **a:SWITCH**
  - States: 0, 1
  - Transitions: a.on, a.off, b.on, b.off

- **b:SWITCH**
  - States: 0, 1
  - Transitions: b.on, b.off

- **SWITCH**
  - States: 0, 1, 2, 3
  - Transitions:
    - 0: off off
    - 1: off on
    - 2: on on
    - 3: on off
**Process Labeling**

**a:P** prefixes each action label in the alphabet of P with a.

*Two *instances* of a switch process:*

```
SWITCH = (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

*An array of *instances* of the switch process:*

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```
Process prefixing is useful for modeling *shared* resources:

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

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

\[
\text{RESOURCE SHARE} = (a: \text{USER} \parallel b: \text{USER} \parallel \{a, b\} :: \text{RESOURCE}) .
\]
How does the model ensure that the user who acquires the resource is the one to release it?
Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

\[/{\text{newlabel}_1/\text{oldlabel}_1, \ldots \text{newlabel}_n/\text{oldlabel}_n}\].

Relabeling to ensure that composed processes synchronize on particular actions:

CLIENT = (call->wait->continue->CLIENT).

SERVER = (request->service->reply->SERVER).
### Action Relabeling

<table>
<thead>
<tr>
<th>CLIENT</th>
<th>SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(call→wait→continue→CLIENT)</td>
<td>(request→service→reply→SERVER)</td>
</tr>
<tr>
<td>( C = (\text{CLIENT} /{\text{reply}/\text{wait}}) )</td>
<td>( S = (\text{SERVER} /{\text{call}/\text{request}}) )</td>
</tr>
</tbody>
</table>

\[ \models C_S = (C \parallel S) \]

---

**Diagram:**

- **C**
  - States: 0, 1, 2
  - Edges: call, wait, reply, continue

- **S**
  - States: 0, 1, 2
  - Edges: request, call, service, reply

- **C_S**
  - States: 0, 1, 2, 3
  - Edges: call, service, reply, continue
An alternative formulation of the client server system is described below using qualified or prefixed labels:

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

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

\[
\text{CLIENT}_v2 \| \text{SERVER}_v2 = (\text{CLIENT}_v2 \| \text{SERVER}_v2)
\]

\[
\text{//}\{\text{call/accept}\}.
\]
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.

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

![Diagram of USER process]

- States: 0, 1, 2
- Transitions: acquire, release, $\tau$
When applied to a process \( P \), the interface operator @ \{a1..ax\} hides all actions in the alphabet of \( P \) not labeled in the set a1..ax.

\[
\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \\
\quad @\{\text{acquire, release}\}.
\]
The following definitions are equivalent:

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

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

Minimization removes hidden \textbf{tau} actions to produce an LTS with equivalent observable behaviour.
The following definitions are equivalent:

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

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

Minimization removes hidden \text{tau} actions to produce an LTS with equivalent observable behaviour.
Process $T$ with alphabet $\{a,b\}$. 

Structure Diagrams
Process $T$ with alphabet \{a,b\}.

Parallel Composition

$(P \parallel Q) / \{m/a, m/b, c/d\}$
Process $T$ with alphabet \{a,b\}.

Parallel Composition

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

Composite process

$A \parallel S = (A \parallel B) @ \{x, y\}$
We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabeling and hiding.

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

\[
||\text{TWOBUFF} = (a:\text{BUFF} \parallel b:\text{BUFF}) \\
\left\{\text{in}/a.\text{in}, a.\text{out}/b.\text{in}, \text{out}/b.\text{out}\right\} \\
\left\{\text{in}, \text{out}\right\}.
\]
Structure diagram for **CLIENT_SERVER**

Structure diagram for **CLIENT_SERVERv2**

Structure Diagrams
RESOURCE = (acquire->release->RESOURCE).
USER = (printer.acquire->use->printer.release->USER).

PRINTER_SHARE = (a:USER || b:USER || {a,b}::printer:RESOURCE).

Structure Diagrams – Resource Sharing
THREAD = OFF,

OFF = (toggle->ON
    | abort->STOP),

ON = (toggle->OFF
    | output->ON
    | abort->STOP).

||THREAD_DEMO = (a:THREAD || b:THREAD)

/ {
 | stop/{a,b}.abort
 |}. 

Interpret:
**toggle**, **abort**
as inputs

**output**
as output
```java
class MyThread extends Thread {
    private boolean on;
    MyThread() {
        on = false;
    }
    void toggle() { on = !on; }
    void abort() { interrupt(); }
    private void output() { System.out.println("output"); }
    public void run() {
        try {
            while (true) {
                if (on) output();
                sleep(1000);
            }
        } catch (InterruptedException _) {
            System.out.println("Done!");
        }
    }
}
```
class MyThread extends Thread {
    private boolean on;
    MyThread() {
        on = false;
    }
    void toggle() { on = !on; }
    void abort() { interrupt(); }
    private void output() { System.out.println("output"); }
    public void run() {
        try {
            while (true) {
                if (on) output();
                sleep(1000);
            }
        } catch (InterruptedException _) {
            System.out.println("Done!");
        }
    }
}
```java
class ThreadDemo {
    private stop(MyThread a, MyThread b) {
        a.abort();
        b.abort();
    }
    public static void main(String[] args) {
        MyThread a = new MyThread();
        MyThread b = new MyThread();
        a.start(); b.start();
        while (true) {
            switch (readChar()) {
                case 'a': a.toogle(); break;
                case 'b': b.toogle(); break;
                case 'i': stop(a,b); return;
            }
        }
    }
}
```
Summary

➢ **Concepts:** concurrent processes and process interaction.

➢ **Models:**

  - asynchronous (*arbitrary speed*) & interleaving (*arbitrary order*)
  
  - parallel composition (*finite state process with action interleaving*)
  
  - process interaction (*shared actions*)
  
  - process labeling, action relabeling, and hiding
  
  - structure diagrams

➢ **Practice:** multiple threads in Java.