

# Labelled transition systems, CCS, and the Mobility Workbench

## *Model-based Design of Distributed and Mobile Systems Lecture 2: Spring 2005*

Mikkel Bundgaard

mikkelbu@itu.dk

Department of Theoretical Computer Science  
IT University of Copenhagen

# Overview of the Lecture

The structure of the lecture:

- Presentation of CCS:
  - Syntax
  - Introduction to inference rules
  - Semantics
  - Examples
- Mobility Workbench
  - Running the workbench
  - Commands
  - An Interactive Session
- Lots of Exercises (for the afternoon)

# Syntax of CCS

Recall, the syntax of a fragment of **CCS**

$P ::= 0$	the inactive process
$\alpha.P$	perform the action $\alpha$ and continue as $P$
$P1 \mid P2$	run $P1$ and $P2$ in parallel
$P1 + P2$	run either as $P1$ or $P2$
$Iden \langle nlist \rangle$	run as the process named $Iden$ instantiated with the names in $nlist$
$(\hat{nlist})P$	restrict the names in $nlist$
$(P)$	parentheses are used for enforcing precedence

- We can also bind an **agent identifier** to an **agent declaration**  $A \stackrel{def}{=} P$
- The meaning of a term  $P$  is a labelled transition system.

$$\llbracket P \rrbracket = (Q, \Sigma, \delta \subseteq Q \times \Sigma \times Q, q \in Q)$$

(We'll write  $P \xrightarrow{\alpha} Q$  for  $(P, \alpha, Q) \in \delta$ )

# Inference rules

- **Inference rules** are widely used in computer science as a scheme for constructing valid inferences (type systems, operational semantics)
- A **rule** consist of a set of formulas called **premises** and an assertion called a **conclusion**
- The intuition is that we can infer new true assertions from other already known ones
- Rules are usually written in the following form

$$\text{RULE} \frac{\textit{Premise1} \quad \textit{Premise2} \quad \dots \quad \textit{Premise}_n}{\textit{Conclusion}}$$

- A rule with no premises is called an **axiom**, which are assertions that are assumed to be true without proof

$$\text{AXIOM} \frac{}{\textit{Conclusion}}$$

# Transition Semantics of CCS

$$\text{ACT} \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{SUM} \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_j} \quad (j \in I)$$

$$\text{COM1} \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{COM2} \frac{P \xrightarrow{\alpha} P'}{Q \mid P \xrightarrow{\alpha} Q \mid P'}$$

$$\text{CON} \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A \stackrel{\text{def}}{=} P) \qquad \text{SYNC} \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{RES} \frac{P \xrightarrow{\alpha} P'}{(\wedge L)P \xrightarrow{\alpha} (\wedge L)P'} \quad (\alpha, \bar{\alpha} \notin L)$$

# Explanation of the Semantics

$$\text{ACT} \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

If the form of our expression is  $\alpha.P$  we can perform the transition  $\alpha$  and become  $P$ .

$$\text{COM1} \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

If left-hand side of a parallel composition can do a transition  $\alpha$  and become  $P'$ , then the entire expression can do the transition, and the left-hand side becomes  $P'$

$$\text{COM2} \frac{P \xrightarrow{\alpha} P'}{Q \mid P \xrightarrow{\alpha} Q \mid P'}$$

Almost as COM1

# Explanation of the Semantics Cont.

$$\text{SUM} \frac{P_j \xrightarrow{\alpha} P'_j}{\Sigma_{i \in I} P_i \xrightarrow{\alpha} P'_j} \quad (j \in I)$$

If either of summands can perform a transition ( $\alpha$ ) to some  $P'$  then the entire expression can perform  $\alpha$  to become  $P'$

$$\text{CON} \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A \stackrel{\text{def}}{=} P)$$

If the defining expression of an agent declaration can make a transition, so can the agent identifier (basically this allows us to replace an identifier with its associated process expression)

# Explanation of the Semantics Cont.

$$\text{SYNC} \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

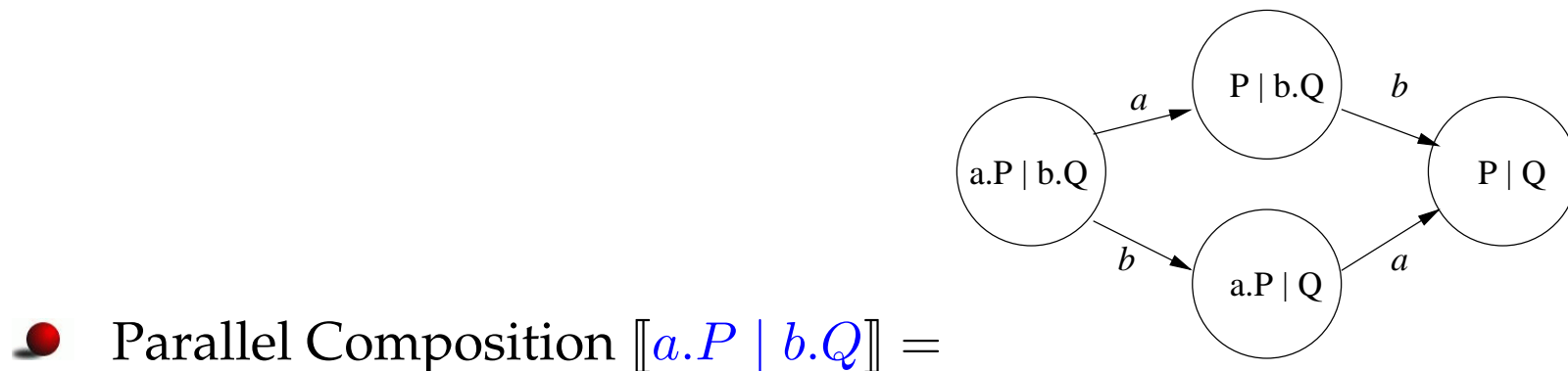
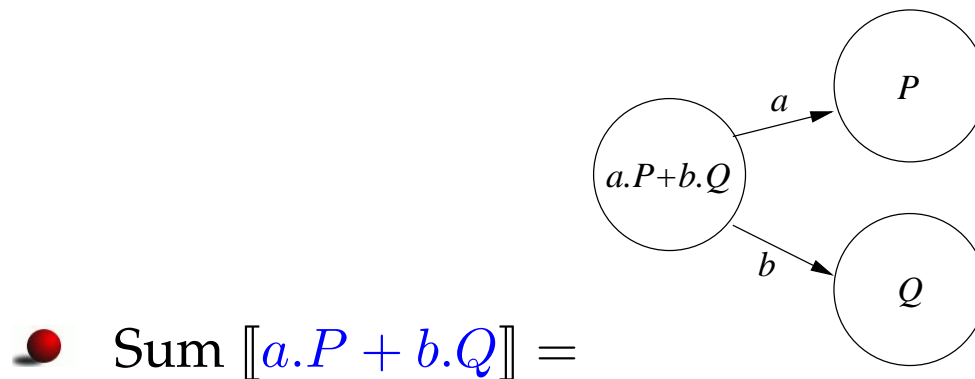
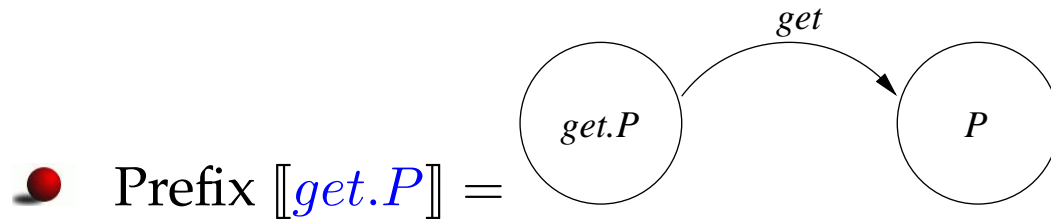
If  $P$  can do an action (input or output) and  $Q$  can do the corresponding co-action, then they can synchronise, resulting in a  $\tau$ -action.

$$\text{RES} \frac{P \xrightarrow{\alpha} P'}{(\wedge L)P \xrightarrow{\alpha} (\wedge L)P'} \quad (\alpha, \bar{\alpha} \notin L)$$

If  $P$  can do an action and this action and the corresponding co-action are not in the **restricted** set, then the entire expression can do the action. Note that the  $\tau$  action cannot be restricted.

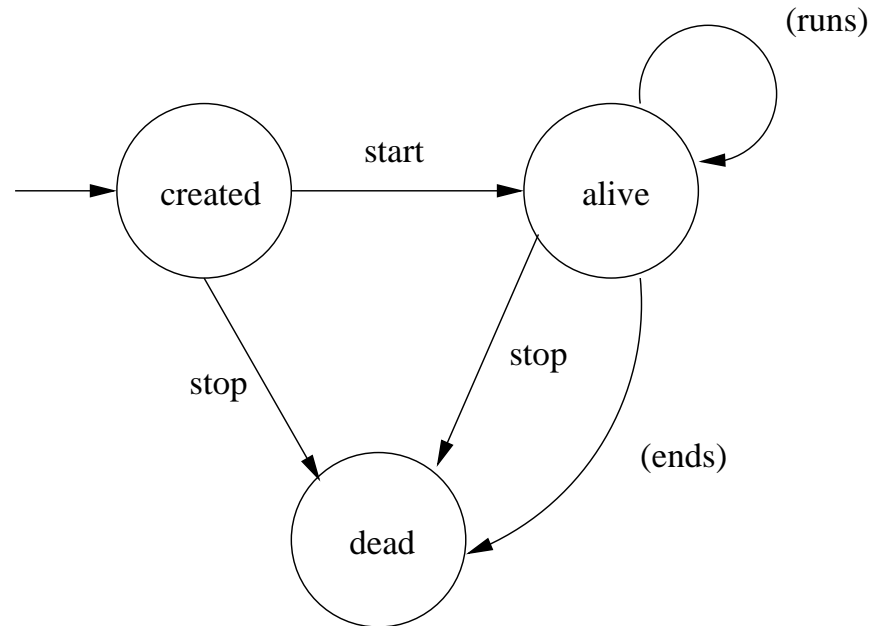
# Processes and LTS

We use a LTS as the **denotation** (meaning) of a process expression



# Traces from LTS

- Consider the following LTS



- Some of the possible **traces** are:
  - stop
  - start, runs, stop
- Give another one ?

# Examples — Derivation Trees

The **inference rules** on slide 5 allows us e.g. to derive the following possible transitions of the process ( $pay3.P + get7.Q$ ):

$$\text{SUM} \frac{\text{ACT} \frac{\text{—————}}{pay3.P \xrightarrow{pay3} P}}{pay3.P + get7.Q \xrightarrow{pay3} P} \quad \text{SUM} \frac{\text{ACT} \frac{\text{—————}}{get7.Q \xrightarrow{get7} Q}}{pay3.P + get7.Q \xrightarrow{get7} Q}$$

Or the possible transitions of  $pay3.P \mid get7.Q$

$$\text{COM1} \frac{\text{ACT} \frac{\text{—————}}{pay3.P \xrightarrow{pay3} P}}{pay3.P \mid get7.Q \xrightarrow{pay3} P \mid get7.Q}$$

$$\text{COM2} \frac{\text{ACT} \frac{\text{—————}}{get7.Q \xrightarrow{get7} Q}}{pay3.P \mid get7.Q \xrightarrow{get7} pay3.P \mid Q}$$

# Examples — Derivation Trees cont.

- A couple more on the board:

$$(pay1.P \mid pay2.Q) \mid pay3.R \xrightarrow{pay2} ?$$

- or if we have defined that  $A \stackrel{def}{=} take.P$  then

$$A \xrightarrow{?} ?$$

- What about restriction ?

$$(\hat{hidden})(hidden.P \mid \overline{hidden}.Q)$$

# Automatic tools

- There exists several automatic tools for CCS and other process calculi, in particular **workbenches**
  - **Mobility Workbench** [Vic95]
  - **Concurrency Workbench** [MS99]
  - ...
- The workbenches allows us to let the computer do the work:
  - in deciding **equivalence** between processes
  - checking properties of processes (**model checking**)
  - **drawing the LTS** of a process
  - **interactively simulating** the processes

We will only use a small fraction of MWB in this lecture.

# Mobility Workbench

- **Mobility Workbench**
  - Primarily developed at Uppsala University
  - [www.it.uu.se/research/group/mobility/mwb](http://www.it.uu.se/research/group/mobility/mwb)
  - Includes model checker and equivalence checker, checking for deadlocks, calculating state space etc.
  - But we will for the moment only use it to execute our processes, and for finding transitions
  - Programmed in Standard ML of New Jersey

# MWB — Starting

- MWB is normally run as an **interactive** session, where you type commands and the workbench responds

```
[mikkelbu@dhcp245 Linux]$ ./mwb.sh
```

```
The Mobility Workbench
```

```
(MWB'99, version 4.135, built Thu Jan...
```

```
MWB>
```

- It is recommended to use **input "sourcecode"** or run the interactive session through emacs

# MWB — Commands

- quit** The command **quit** terminates MWB
- agent** Through the command **agent** **NAME**(arguments) = *expr* we declare a new identifier called **NAME** for the expression *expr*. Later we can use **NAME** in different commands as **clear**, **env**, **step**
- clear** The command **clear** removes all agent declarations, while **clear** *P* only removes the declaration of agent *P*
- env** The command **env** print out all agent declarations, while **env** *P* only prints out the declaration of agent *P*
- input** To read the commands from a file name *filename* use **input** "*filename*"

# MWB — Commands Continued

- step** The **step** command allows us to run the agent definitions step-by-step. At each step MWB shows us the possible transitions and we then chose one.
- transitions** The command **transitions**  $P$  will print out all the possible transitions of the agent  $P$
- set** Using the command **set** we can set various properties of MWB. The only relevant so far is **set debug**  $n$ , where  $n$  is either 0 or 1 (more information)
- size** the commmand **size**  $agent$  gives an approximation on the number of states in the labelled transition system of the  $agent$

# An Interactive Session (input, transitions, size)

Contents of `buffer.ag`:

```
agent Buf0(in,out) = in.Buf1<in,out>
agent Buf1(in,out) = in.Buf2<in,out> + 'out.Buf0<in,out>
agent Buf2(in,out) = 'out.Buf1<in,out>
```

```
MWB>input "buffer.ag"          (*load the contents of buffer.ag*)
```

```
MWB>transitions Buf0<in,out>  (*transitions of Buf0*)
```

Commitments:

```
|>in.(in.Buf2<in,out> + 'out.Buf0<in,out>)
```

```
MWB>transitions Buf1<in,out>  (*transitions of Buf1*)
```

Commitments:

```
|>in.'out.Buf1<in,out>
```

```
|>'out.in.Buf1<in,out>
```

```
MWB>size Buf0<in,out>        (*states in the LTS *)
```

3

```
MWB>size Buf0                (*no name instantiation *)
```

3

# An Interactive Session cont. (step)

Contents of `buffer.ag`:

```
agent Buf0(in,out) = in.Buf1<in,out>
```

```
agent Buf1(in,out) = in.Buf2<in,out> + 'out.Buf0<in,out>
```

```
agent Buf2(in,out) = 'out.Buf1<in,out>
```

```
MWB>step Buf0<in,out> (* simulate Buf0 *)
```

```
* Valid responses are:
```

```
  a number N >= 0 to select the Nth commitment,
```

```
  <CR> to select commitment 0,
```

```
  q to quit.
```

```
0: |>in.(in.Buf2<in,out> + 'out.Buf0<in,out>) (* only 1 choice *)
```

```
Step>0 (* take transition *)
```

```
0: |>in.'out.Buf1<in,out> (* 2 choices *)
```

```
1: |>'out.in.Buf1<in,out>
```

```
Step>0
```

```
0: |>'out.(in.Buf2<in,out> + 'out.Buf0<in,out>)
```

```
Step>q
```

# An Interactive Session cont. (clear, env, quit)

Contents of `buffer.ag`:

```
agent Buf0(in,out) = in.Buf1<in,out>
agent Buf1(in,out) = in.Buf2<in,out> + 'out.Buf0<in,out>
agent Buf2(in,out) = 'out.Buf1<in,out>
```

```
MWB>clear Buf0          (* clear agent declaration of Buf0 *)
MWB>env                 (* show all agent declarations      *)
agent Buf1 = (\in,out)(in.Buf2<in,out> + 'out.Buf0<in,out>)
agent Buf2 = (\in,out)'out.Buf1<in,out>
MWB>env Buf1           (* show agent declaration of Buf1  *)
agent Buf1 = (\in,out)(in.Buf2<in,out> + 'out.Buf0<in,out>)
MWB>clear              (* clear all agent declarations *)
Clearing environment.
MWB>env                (* No more agent declarations *)
MWB>quit               (* quit MWB *)
```

# State Space Problems

- Since Mobility Workbench represents the state-space explicit, and not symbolically, we can only represent finite state spaces
- Also some operations may not terminate

```
MWB>agent U(x) = x.0 | U<x>  
MWB>step U<x>
```

\*Interrupt\*

```
MWB>agent Split(x) = x.(Split<x> | Split<x>)  
MWB>deadlocks Split<x>
```

\*Interrupt\*

# Exercises

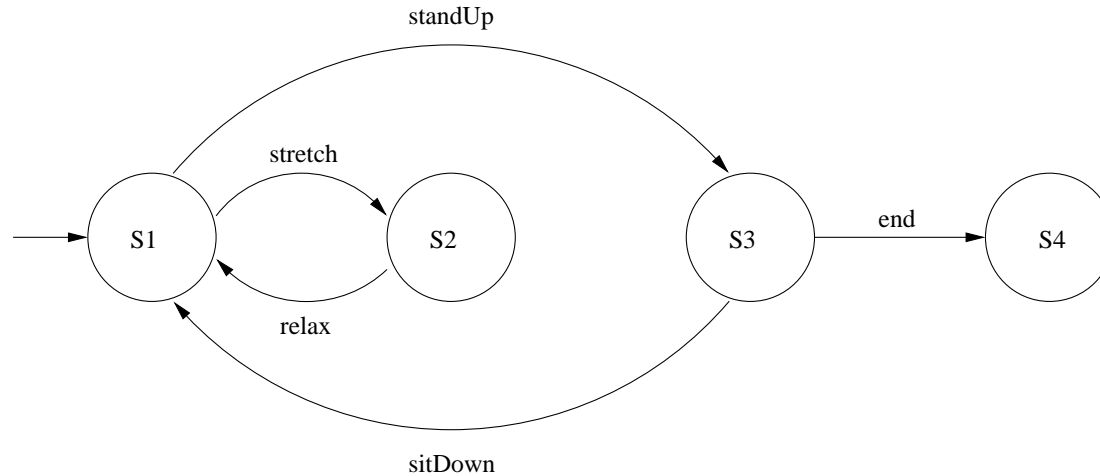
- Questions: please use the **newsgroup** associated with this course
  - Server: `news.itu.dk`
  - Group: `it-c.courses.IMDD`
- Otherwise, come and ask me
- Solve the exercises in groups (except the mandatory exercises)

# Exercises — Continued (Syntax)

- Are the following terms valid **CCS** expressions (if not explain why) ?
  - $O \mid O$
  - $(P1 \mid P2).P3$
  - $take.(P1 + (P2 \mid P3))$
  - $0.0$
  - $P1.P2$
  - $take..give.0$
  - $take$

# Exercises — Continued

- Write a **CCS** expression which denotes the following **LTS**



- It is possible to do the following traces:
  - standUp, sitDown, stretch, relax, standup, end ?
  - standUp, end, end ?
  - standUp, sitDown, standUp, sitDown, ...?

# Exercises — Continued

- Draw the labelled transition system which is the denotation of the following expression  $split.(doA.0 \mid doB.0)$  (hint: it can be drawn using 5 states)

# Exercises — Continued

● Are the following transitions possible in our semantics (if they are then write the derivation tree)

●  $take.P \xrightarrow{take} P$

●  $take.P \xrightarrow{get} P$

●  $0 \xrightarrow{\alpha} 0$

●  $(eat.P + sleep.Q) \mid eyesshut.R \xrightarrow{eat} P \mid eyesshut.R$

●  $(eat.P + sleep.Q) \mid eyesshut.R \xrightarrow{eyesshut} eat.P \mid R$

●  $(take.P \mid get.Q) + nothing.0 \xrightarrow{take} P \mid get.Q$

●  $eat.P \mid \overline{eat}.Q \xrightarrow{\tau} P \mid Q$

●  $eat.P + \overline{eat}.Q \xrightarrow{\tau} P + Q$

# Exercises — Continued

- Create a file called `buffer.ag` containing the definitions from slide 18.
- Follow the same interactive session as in these slides
- If you like, experiment with some of the other commands of MWB
- Reload the definitions in `buffer.ag`
- Now we want to build a larger buffer from the 2-element buffer
  - To this end we use synchronisation and restriction
  - We define the process `FourElem` as  

```
agent FourElem(in,out) =  
    (^mid)(Buf0<in,mid> |Buf0<mid,out>)
```
  - Explain your intuition about the process `FourElem` and check your guesses using MWB

# Exercises — Continued (Beverage Machine)

- Make first a **LTS** and then a **CCS** expression of the following beverage machine:
  - The machine can accept `coin1` (1 kr.) and `coin2` (2 kr.) up to a maximum of 4 kr.
  - When it has received enough money, it must be possible to get either `tea` (1 kr.) or `coffee` (2 kr), the machine must then give correct change back (`change1–change3`), and return to its initial state.
- How many states does the machine have ?
- Is it possible to do the following trace and end up in the initial state:  
`coin1, coin2, getTea, change2`
- Implement the CCS expression in **MWB** and try simulating different runs using `step`.

# Exercises — Continued (Inference rules)

- Given that all the possible traces (to a terminal state) of a process is the following:  $x, y$  and  $x, z$ 
  - does this uniquely define the LTS (hint: non-determinism) ?
    - If yes, then draw the LTS
    - Otherwise, why not ? (give a counter-example with two LTSs containing these traces)
- Find the possible initial transitions of the following process  $trans1.0 + (trans2.0 \mid trans3.0)$  using the transition semantics
- Check the result in MWB with the command `transitions agent`, which prints out all transitions of *agent*

# References

## References

- [MS99] Faron Moller and Perdita Stevens. *Edinburgh Concurrency Workbench User Manual (Version 7.1)*. Laboratory for Foundations of Computer Science, University of Edinburgh, 1999. Available from <http://www.dcs.ed.ac.uk/home/cwb/>.
- [Vic95] Björn Victor. *The Mobility Workbench User's Guide: Polyadic version 3.122*. Department of Information Technology, Uppsala University, 1995.