Bisimulations
Do we need equivalences at all?

- Process algebraists use equivalences because this is the only way for them to verify programs.

- With operational semantics:
  - we translate (well, not to large) programs into graphs
  - we can do visual checking
  - we can do model checking
  - also, equivalences are more expensive than model checking
    - roughly: $O(n \log n)$ vs $O(n)$
  - do we still need equivalences?

- Yes. Equivalences are useful
  - to minimize LTSs (e.g. before visual or model checking)
  - to avoid writing complex temporal logic formulas
  - to check if certain traces are accepted by an LTS
  - to fight state explosion (compositional minimization)
Automata equivalence checks whether two automata accept the same language

- same language = same set of accepted words (or traces)
- this is perfect for regular expressions and compiler scanners

This is not suitable for studying concurrency

- comparing languages is not enough
- two LTS may have the same language but behave differently

Both LTSs recognize the same traces \{a.b, a.c\} but putting them in parallel with a.b generates a deadlock in the 2^{nd} case

‘coffee-vending machine’ example
In the literature, there are nearly 50 different equivalences for LTSs.

In practice, only two or three are needed:

- strong bisimulation: preserves all properties on LTSs (well, not the number of states nor the branching factor)
- weak bisimulation: try to eliminate or collapse sequences of \( \tau \)-transitions which are not observable anyway. Branching bisimulation is a suitable weak bisimulation.
- some divergence-preserving bisimulation

Also useful:
- equivalences taking time and/or probabilities into account
A critical look at CCS
Syntax of CCS

(channel, port) names: \( a, b, c, \ldots \)

corne-names: \( \bar{a}, \bar{b}, \bar{c}, \ldots \)

silent action: \( \tau \)

actions, prefixes: \( \mu ::= a \mid \bar{a} \mid \tau \)

processes: \( P, Q ::= 0 \) inaction

\mid \mu.P \) prefix

\mid P \mid Q \) parallel

\mid P + Q \) (external) choice

\mid \nu a.P \) restriction

\mid \text{rec}_K P \) process \( P \) with definition \( K = P \)

\mid K \) (defined) process name
A very small number of rules

- **Act**
  \[
  \frac{\mu. P \xrightarrow{\mu} P}{\mu. P \xrightarrow{\mu} P}
  \]

- **Res**
  \[
  \frac{P \xrightarrow{\mu} P'}{(\nu a)P \xrightarrow{\mu} (\nu a)P'}
  \]

- **Sum1**
  \[
  \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}
  \]

- **Sum2**
  \[
  \frac{Q \xrightarrow{\mu} Q'}{P + Q \xrightarrow{\mu} Q'}
  \]

- **Par1**
  \[
  \frac{P \xrightarrow{\mu} P'}{P \parallel Q \xrightarrow{\mu} P' \parallel Q}
  \]

- **Par2**
  \[
  \frac{Q \xrightarrow{\mu} Q'}{P \parallel Q \xrightarrow{\mu} P \parallel Q'}
  \]

- **Com**
  \[
  \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}
  \]

- **Rec**
  \[
  \frac{P[\text{rec}_K P/K] \xrightarrow{\mu} P'}{\text{rec}_K P \xrightarrow{\mu} P'}
  \]
Minimality
- appealing in academia, but does not scale up to real problems
- the LOTOS ISO committee added the required extensions

Sequential composition
- CCS action-prefix proved to be a bad language design decision
- see Lecture 3 for a discussion (LOTOS vs LOTOS NT)

Parallel composition
- CCS parallel composition is worse than the one of CSP/LOTOS
- only supports binary rendez-vous (co-names are a mistake)
- even the binary communication is badly designed
CCS parallel composition

- No list of gates on which to synchronize or not
- [Par1] and [Par2]: each parallel process can always evolve alone and ignore the rendez-vous!
- [Com]: the rendezvous is immediately renamed into $\tau$
  impossible to observe in the LTS $\Rightarrow$ verification impossible

$$a . 0 \mid \bar{a} . 0 = \begin{array}{c}
  a \quad \tau \\
  \bar{a} \\
\end{array}$$

a restriction on a is required to force the rendezvous
**Limitation of binary synchronization:**
how to specify \((P || Q) ; R\) ? (LOTOS NT semantics)

This is a 3-party rendez-vous: P and Q wait each other to terminate and R waits to start

CCS requires 2 additional rendezvous \(\delta_1\) and \(\delta_2\):

\[
((P \cdot \delta_1 \mid Q \cdot ('\delta_1 \cdot \delta_2) \ \delta_1 \mid '\delta_2 \cdot R) \ \delta_1 \mid '\delta_2 \cdot R) \ \delta_2
\]

this creates two \(\tau\)-transitions in the LTS (too bad)
The pi-calculus
In ‘classical’ process calculi (CCS, CSP, LOTOS...):
- one often describes a finite set of concurrent actors
- these actors can be (recursively) nested
- the communication topology (i.e., gates) is fixed
- well-adapted to hardware design, data transmission protocols

In fact, ‘classical’ process calculi can do more:
- dynamic creation/destruction of actors and channels
  Example: \( A ; \) hide \( G \) in \((B \ || [G] \ || C) \ ; D\)
- unbounded dynamic creation of actors
  Example: process \( P(N) := \) if \( N=0 \) then \( Q \) else \((P(N-1) \ || \ || Q)\)

(mixing LOTOS and LOTOS NT syntaxes)
‘Mobile process calculi’ : a more radical approach
- dynamically evolving networks
- actors can be created/deleted dynamically
- channels (communication links) also
- actors can discover each other, and then communicate
- often, they are put in relation by a third-party (‘trader’)

Real-life examples:
- plug-and-play devices on a network
- mobile phones and base stations
- object-oriented software
The printer discovery example (J. Parrow):

One approach to mobility: sending channels

- impossible in ‘classical’ process calculi, where offers sent or received on gates only contain data values (but not gates)
- sending processes is similar to sending channels
The pi-calculus

- Proposed by R. Milner, J. Parrow, D. Walker in the early 90s (see References)
- Defined as an extension of CCS
- Two main changes:
  - channels can be sent on channels
  - the restriction operator of CCS is technically modified
- A very influential model in academia:
  - many variants
  - some tools, such as the Mobility Workbench [http://www.it.uu.se/research/group/mobility/mwb](http://www.it.uu.se/research/group/mobility/mwb)
  - some applications - basis for defining BPEL
  - see [http://move.to/mobility](http://move.to/mobility)
### Syntax

**Prefixes**

\[ \alpha ::= \bar{a}x \quad \text{Output (noted a !x in LOTOS)} \]

\[ a(x) \quad \text{Input (noted a ?x in LOTOS)} \]

\[ \tau \quad \text{Silent} \]

**Agents**

\[ P ::= 0 \quad \text{Nil} \]

\[ \alpha . P \quad \text{Prefix} \]

\[ P + P \quad \text{Sum} \]

\[ P | P \quad \text{Parallel} \]

\[ \text{if } x = y \text{ then } P \quad \text{Match} \]

\[ \text{if } x \neq y \text{ then } P \quad \text{Mismatch} \]

\[ (\nu x)P \quad \text{Restriction} \]

\[ A(y_1, \ldots, y_n) \quad \text{Identifier} \]

**Definitions**

\[ A(x_1, \ldots, x_n) \overset{\text{def}}{=} P \quad \text{(where } i \neq j \Rightarrow x_i \neq x_j) \]

- Initially noted \( P \setminus a \) as in CCS
- Also written ‘a<x>

Added later
Static semantics

- A single ‘type’ of data, merging values and channels
- Variables are defined (‘bound’) only at 3 places:
  - \( x \ (y). \ P \) : variable \( y \) contains the data received on \( x \)
    - \( y \) is visible only in \( P \)
  - \( (\nu y) \ P \) : a new channel is created and assigned to variable \( y \)
    - \( y \) is visible only in \( P \), but \( P \) may send \( y \) to other agents
      (this is called ‘scope extrusion’ - tricky rules)
  - \( A \ (x_1, \ldots, x_n) = P \) : parameters \( x_1, \ldots, x_n \) are visible in \( P \)
- \( bn(P) := \) bound variables defined in \( P \) : \( x \ (y) \) or \( (\nu y) \)
- \( fn(P) := \) all other variables used in \( P \) (free variables)
Dynamic semantics

\[ \begin{align*}
\text{TAU} & \quad \tau . P \xrightarrow{\tau} P \\
\text{SUM} & \quad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'} \\
\text{COM} & \quad \frac{P_1 \xrightarrow{x. y} P_1'} {P_2 \xrightarrow{x. y} P_2'} \\ & \quad \frac{P \xrightarrow{\alpha} P'}{P_1 | P_2 \xrightarrow{\tau} P_1' | P_2'} \quad \text{if } x \notin n(\alpha) \\
\text{RES} & \quad \frac{(\nu x) P \xrightarrow{\alpha} (\nu x) P'}{P \xrightarrow{\alpha} P'} \\
\text{MATCH} & \quad \frac{[x = x] P \xrightarrow{\alpha} P'}{P \xrightarrow{\alpha} P'} \\
\text{OUT} & \quad \frac{\overline{x. y} . P \xrightarrow{x. y} P}{\overline{x. y} . P \xrightarrow{x. z} P\{z/y\}} \\
\text{IN} & \quad \frac{x(y) . P \xrightarrow{x. z} P\{z/y\}}{x(y) . P \xrightarrow{x. z} P\{z/y\}} \\
\text{PAR} & \quad \frac{P_1 \xrightarrow{\alpha} P_1'} {P_1 | P_2 \xrightarrow{\alpha} P_1' | P_2'} \quad \text{if } bn(\alpha) \cap fn(P_2) = \emptyset \\
\text{CLOSE} & \quad \frac{P_1 \xrightarrow{x(\overline{y})} P_1'} {P_2 \xrightarrow{x} P_2'} \quad \text{if } y \notin fn(P_2) \\
\text{OPEN} & \quad \frac{\overline{y} \alpha . P \xrightarrow{x. z} P\{z/y\}}{P \xrightarrow{x. y} P'} \quad \text{if } x \neq y, z \notin fn((\nu y)P') \\
\text{MISMATCH} & \quad \frac{[x \neq y] P \xrightarrow{\alpha} P'}{P \xrightarrow{\alpha} P'} \quad \text{if } x \neq y \\
\text{IDE} & \quad \frac{P\{y_1/x_1, \ldots, y_{r(A)}/x_{r(A)}\} \xrightarrow{\alpha} P'}{A(y_1, \ldots, y_{r(A)}) \xrightarrow{\alpha} P'} \quad \text{if } A(x_1, \ldots, x_{r(A)}) \overset{\text{def}}{=} P
\end{align*} \]
Example

Taken from Mateescu-Salaün IFM 2010 paper (see references)

\[
Main = (\nu \text{ req, a, b, c})(\text{Client}(\text{req, a, b, c}) \mid \text{Dispatcher}(\text{req}) \mid \text{Server}(a) \mid \text{Server}(b) \mid \text{Server}(c))
\]

\[
\text{Client}(\text{req, a, b, c}) = (\nu x)(\text{request a} \cdot \text{req}(a, x) \cdot \text{ClientAux}(\text{req, a, a, b, c, x})) + (\nu x)(\text{request b} \cdot \text{req}(b, x) \cdot \text{ClientAux}(\text{req, b, a, b, c, x})) + (\nu x)(\text{request c} \cdot \text{req}(c, x) \cdot \text{ClientAux}(\text{req, c, a, b, c, x}))
\]

\[
\text{ClientAux}(\text{req, k, a, b, c, x}) = x(\text{info}).(\overline{x} \cdot \text{purchase}.\text{purchase k.0} + \overline{x} \cdot \text{refuse}.\text{refuse k}.\text{Client}(\text{req, a, b, c}))
\]

\[
\text{Dispatcher}(\text{req}) = \text{req}(k, x) \cdot \overline{k} \cdot x.\text{Dispatcher}(\text{req})
\]

\[
\text{Server}(k) = k(x) \cdot \overline{x} \cdot \text{info}.x(\text{decision}).\text{Server}(k)
\]
The PIC2LNT tool
A recent translator developed at INRIA Grenoble

**Input language:** PIC
- pi-calculus
- with a machine-readable syntax (from Mobility Workbench)
- extended with data values (= ‘applied pi-calculus’)

**Output:** LOTOS NT program

A script named ‘pic2bcg’ automates the translation
PIC → LOTOS NT → LOTOS → Petri nets → LTS
The PIC language
- defined in the PIC2LNT manual page (see References)
- the data types and value expressions are those of LOTOS NT

The translation approach:
- most pi-calculus tools do symbolic proofs on the terms
- pic2lnt works by state space exploration
  (= explicit-state enumeration = reachability analysis)
- limitation: only works for finite-state models
- ⇒ bounding channels, data types, ‘!’ operator
- BUT enables to study non-trivial mobile programs
A few notes

- Caution: ‘t’ means $\tau$ (contrary to ‘i’ in LOTOS/NT)
- The restriction operator $\nu$ must be written ‘new’
- Emissions $\bar{x}$ have to be noted ‘x’
- Emitted parameters must be bracked with < and > even when there is only a single parameter
- Received parameters must be bracked with ( and ) even when there is only a single parameter
- There are no channel declarations: beware of typos
  - exploit: at any place, you can easily insert a ’debug event
In the LTS obtained, the labels carry extra offers:
- for instance: !FALSE or !TRUE
- this is an artefact of the translation to LOTOS NT
- (perhaps the pic2bcg script could remove them)

The translation implements the creation of new channels by giving unique numbers:
- example: (new y) 'x<y> may generate a transition: X !Y(41)
- don’t worry if the counter is not increasing one by one

Restriction hides the synchronizations 😞
- one cannot observe them in the LTS (only τ-transitions can be seen)
- add extra events if needed
Today’s challenge
Your first pi-calculus program (1/2)

- Find the paper about PIC2LNT published at IFM 2010 (see References below)
- Copy-and-paste in a file named ‘disp.pic’ the pi-calculus example given page 11
- Convert it to machine-readable notations:
  - replace each $\nu$ symbol by the new keyword
  - replace emissions $\overline{x}$ with ’x
  - restore the < and > symbols around emissions of multiple channels; add them for emissions of single channels
  - same with ( and ) for receptions
  - finally, replace the 0 with nil (0 is not documented in the manual page, yet seems to be accepted)
Perform the translation PIC → LOTOS NT → LOTOS → Petri nets → LTS by typing:

- $ \text{pic2bcg disp.pic}$
- if it does not compile properly, fix the mistakes

Visualize the file ‘disp.bcg’ obtained

- $ \text{bcg_edit disp.bcg}$

Compare it to the picture given page 11

Minimize it using strong bisimulation to remove ‘duplicated’ parts of the LTS

- $ \text{bcg_min disp.bcg}$
- $ \text{bcg_edit disp.bcg}$

Send your file ‘disp.pic’ and the PostScript file to Alexander (possibly with comments if you observe a difference with the picture of the paper)

Pi-calculus bibliography


- On-line resources: [http://move.to/mobility](http://move.to/mobility)
Tools for the pi-calculus

PIC2LNT translator, by R. Mateescu and G. Salaün, 2010-12. In your VM, directory $HOME/Desktop/PIC2LNT

- Reference documentation: 
  *The PIC2LNT manual page*
  in your VM, directory $HOME/Desktop/PIC2LNT/man/pdf

- If you want details on the translation: 
  R. Mateescu and G. Salaün. *Translating Pi-Calculus into LOTOS NT*. IFM 2010
  in your VM, directory $HOME/Desktop/PIC2LNT/doc/pdf
  (caution: their version of LOTOS NT is highly simplified)