PL 2008 - Punta Arenas
A (Gentle?) Introduction to Process Calculi

Jacques Noyé
OBASCO - Ecole des Mines de Nantes/INRIA, LINA
Jacques.Noye@emn.fr

11-12 November 2008
Process Calculi

- A family of approaches to formally model concurrent systems: interaction, communication, and synchronization between independent processes (or agents).
- Algebraic laws make it possible to manipulate and reason about these models (in particular in terms of their behavioral equivalence).
The Agenda

- FSP (Finite State Processes) [MK06]
  - Processes are modelled graphically by labelled transitions systems (LTS) and textually by FSP
  - LTSA (Labelled Transition System Analyzer) translates FSPs into LTSs and provides model animation and model checking of safety and liveness properties.
- Communicating automata (revised version of CCS - a Calculus of Communicating Systems)
- The $\pi$-calculus [MPW92, Mil93, Mil99]
- The asynchronous $\pi$-calculus [HT91]
A \textit{(sequential) process} is the execution of a \textit{sequential program}. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions. [MK06]

The corresponding sequence of actions (there is only one) or \textit{trace}:

\textit{on} $\rightarrow$ \textit{off} $\rightarrow$ \textit{on} $\rightarrow$ \textit{off} $\rightarrow$ \textit{on} $\rightarrow$ \textit{off} $\ldots$
Example 2 - Several Traces

blue -> tea -> blue -> tea -> blue -> tea ...
red -> coffee -> blue -> tea -> blue -> tea ...
... 
red -> coffee -> red -> coffee -> red -> coffee ...
Example 3 - Nondeterministic

toss -> tails -> toss -> tails -> toss -> tails ...
toss -> heads -> toss -> tails -> toss -> tails ...
...
toss -> heads -> toss -> heads -> toss -> heads ...
(Basic) FSP Syntax

\[\text{ProcessDefinition} ::= \text{ProcessName} = \text{ProcessExpression}\]

\[\text{ProcessExpression} ::= \text{ProcessName} \mid \text{ActionPrefix} \mid \text{Choice}\]

\[\text{ActionPrefix} ::= (\text{Action} \rightarrow \text{ProcessExpression})\]

\[\text{Choice} ::= (\text{Action} \rightarrow \text{ProcessExpression} \mid \text{Action} \rightarrow \text{ProcessExpression})\]
Recursive Definitions and Action Prefixes

SWITCH = OFF, OFF = (on -> ON), ON = (off-> OFF).

or

SWITCH = OFF, OFF = (on -> (off -> OFF)).
SWITCH = (on -> off -> SWITCH). % -> is right-associative
Choices

\[
\text{DRINKS} = ( \text{red} \rightarrow \text{coffee} \rightarrow \text{DRINKS} \\
| \text{blue} \rightarrow \text{tea} \rightarrow \text{DRINKS} \\
).
\]
BUFF = (write[i:0..3]→read[i]→BUFF).

is equivalent to:

BUFF = (write[0]→read[0]→BUFF
).

Note: | is commutative and associative.
range R = 0..1
BUFF = BUFF[0],
BUFF[old:R] = ( read[old] -> BUFF[old]
    | write[new:R] -> BUFF[new]).

is equivalent to:

BUFF = BUFF[0],
BUFF[0] = ( read[0] -> BUFF[0]
    | write[0] -> BUFF[0]
    | write[1] -> BUFF[1]),
    | write[0] -> BUFF[0]
    | write[1] -> BUFF[1]).
A (finite) **LTS** is a quadruple \(< S, A, \Delta, q >\) where:

- \(S\) is a *finite* set of states
- \(A\) is the *alphabet* of the LTS (a set of labels)
- \(\Delta \subseteq (S \times A \times S)\) is the *transition relation* of LTS
- \(q\) is the initial state of the LTS.

That is, a nondeterministic automaton without accepting states.

An LTS \(L = < S, A, \Delta, q >\) **transits** with action \(a \in A\) into and LTS \(L', L \xrightarrow{a} L'\) if: \(P' = < S, A, \Delta, q' >\), where \((q, a, q') \in \Delta\).
The semantics is given by associating an LTS to each process expression: \( \text{lts} : \text{ProcessExpression} \rightarrow \text{LTS} \)

\[
\begin{align*}
\text{Definition} & \quad \frac{P = E}{\text{lts}(P) = \text{lts}(E)} \\
\text{Prefix} & \quad \frac{\text{lts}(E) = \langle S, A, \Delta, q \rangle}{\text{lts}(a \rightarrow E) = \langle S \cup \{p\}, A \cup \{a\}, \Delta \cup \{(p, a, q)\}, p \rangle \text{ where } p \notin S} \\
\text{Choice} & \quad \frac{\text{lts}(E_1) = \langle S_1, A_1, \Delta_1, q_1 \rangle \quad \text{lts}(E_2) = \langle S_2, A_2, \Delta_2, q_2 \rangle}{\text{lts}(a_1 \rightarrow E_1 \mid a_2 \rightarrow E_2) = \langle S \cup \{p\}, A_1 \cup A_2 \cup \{a_1, a_2\}, \Delta \cup \{(p, a_1, q_1), (p, a_2, q_2)\}, p \rangle \text{ where } p \notin S}
\end{align*}
\]
Parallel Composition

Parallel composition construct: $(\text{ProcessName} \ || \ \text{ProcessName})$

$P = \ldots$

$Q = \ldots$

$\ || P Q = (P \ || \ Q)$.

Note: in FSP, it is not possible to mix the definition of sequential processes and parallel processes.
There is a new association rule for parallel composition:

\[ \text{PARALLEL\text{-}COMPOSITION} \quad lts(P || Q) = lts(P) || lts(Q) \]

This requires to define the parallel composition of LTSs.
Composing LTSs

Let us consider $L_1 = <S_1, A_1, \Delta_1, q_1>$ and $L_2 = <S_2, A_2, \Delta_2, q_2>$.
$L_1 \parallel L_2 = <S_1 \times S_2, A_1 \cup A_2, \Delta, (q_1, q_2)>$, where $\Delta$ is the smallest relation satisfying the following rules:

$$L_1 \xrightarrow{a} L_1' \quad a \notin A_2$$

$$\frac{L_1 \parallel L_2 \xrightarrow{a} L_1' \parallel L_2}{L_1 \parallel L_2 \xrightarrow{a} L_1 \parallel L_2'}$$

$$L_2 \xrightarrow{a} L_2' \quad a \notin A_1$$

$$\frac{L_2 \xrightarrow{a} L_2' \quad a \notin A_1}{L_1 \parallel L_2 \xrightarrow{a} L_1 \parallel L_2'}$$

$$L_1 \xrightarrow{a} L_1' \quad L_2 \xrightarrow{a} L_2'$$

$$\frac{L_1 \parallel L_2 \xrightarrow{a} L_1' \parallel L_2'}{a \in A_1 \cup A_2, \text{ it is a shared action}}$$
Algebraic laws

- \( P || Q = Q || P \) is commutative.
- \( (P || Q) || R = P || (Q || R) \) is associative.

This gives \textit{n-ary synchronization} on shared actions.
A structural/component view of composition

The alphabet of a process is its interface, its definition is its implementation.

RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).
||S = (USER || RESOURCE).
Hiding actions

\[ ||S = (USER || RESOURCE)\{acquire, release}\. \]

This creates \( \tau \) transitions in the underlying LTS.
**Relabeling actions**

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

\[
\mid | S = (\text{USER} \mid | \text{RESOURCE})/\{\text{acquire}/\text{lock}, \text{release}/\text{unlock}\}.
\]
Relabeling processes

All the transitions of a sequential process can be prefixed (e.g. to create some kind of “instances”).

\[
\text{RESOURCE} = (\text{lock} \rightarrow \text{unlock} \rightarrow \text{RESOURCE}). \\
\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}). \\
||S = (\text{a:USER} \parallel \text{b:USER} \parallel \text{RESOURCE}).
\]
Set prefixing

Instead of a single prefix, a set can be used. This creates a process that is not structurally equivalent to the initial one.

RESOURCE = (lock->unlock->RESOURCE).
USER = (acquire->use->release->USER).
||S = (a:USER || b:USER || {a,b}::RESOURCE).

![Diagram of process execution](image-url)
A quick demo?
Communicating Automata [Mil99]

- **Binary synchronization** through complementary actions $a$ and $\bar{a}$
- Lean syntax
- Semantics given by either:
  - Transition rules
  - Reaction rules (à la Chemical Abstract Machine)
There is no layering of sequential and parallel processes

- Processes are parameterized by their actions (relabeling)
- `new` restricts the scope of an action (hiding)

\[
D ::= A(\overrightarrow{a}) = P_A \\
P ::= A(\overrightarrow{a}) \mid \sum_{i \in I} \alpha_i.P_i \mid P_1|P_2 \mid \text{new } a P
\]
Labelled Semantics

\[ \text{SUM}_t \quad M + \alpha.P + N^\alpha \rightarrow P \]

\[ \text{REACT}_t \quad \frac{P^\lambda \rightarrow P'}{P \mid Q^\tau \rightarrow P' \mid Q} \]

\[ \text{L-PAR}_t \quad \frac{P^\alpha \rightarrow P'}{P \mid Q^\alpha \rightarrow P' \mid Q} \]

\[ \text{R-PAR}_t \quad \frac{Q^\alpha \rightarrow Q'}{P \mid Q^\alpha \rightarrow P' \mid Q} \]

\[ \text{RES}_t \quad \frac{P^\alpha \rightarrow P'}{\text{new } a \ P^\alpha \rightarrow \text{new } a \ P'} \quad \text{if } \alpha \notin \{a, \bar{a}\} \]

\[ \text{IDENT}_t \quad \frac{\{\overrightarrow{b / \bar{a}}\} \ P_A^\alpha \rightarrow P'}{A < \overrightarrow{b} \rightarrow \alpha \rightarrow P'} \quad \text{if } A(\overrightarrow{a}) = P_A \]
Two processes $P$ and $Q$ are **structurally congruent**, $P \equiv Q$, if they are identical up to structure. Structural congruence is the least equivalence relation preserved by the process constructs and the following rules:

- $P \equiv Q$ modulo alpha-conversion of bound variables (\texttt{new})
- $P \equiv Q$ modulo reordering choices
- $P \equiv Q$ modulo reordering parallel composition (including $P | 0 \equiv P$)
- restrictions
  - $\text{new } a (P | Q) \equiv P | \text{new } a Q$ if $a$ is not free in $P$
  - $\text{new } a 0 \equiv 0$
  - $\text{new } a (\text{new } b P) \equiv \text{new } b (\text{new } a P)$
- $A < \vec{b} > \equiv \{ \vec{b} / \vec{a} \} P_A$ if $A(\vec{a}) = P_A$
Semantics - Reaction Rules

\[ \text{TAU} \frac{\tau.P + M \rightarrow P}{\text{REACT} \ \frac{(a.P + M)|(\bar{a}.Q + N) \rightarrow P|Q}} \]

\[ \text{PAR} \ \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q} \]

\[ \text{RES} \ \frac{P \rightarrow P'}{\text{new } a \ P \rightarrow \text{new } a \ P'} \]

\[ \text{STRUCT} \ \frac{P \rightarrow P'}{Q \rightarrow Q'} \text{ if } P \equiv Q \text{ and } P' \equiv Q' \]
Example [Mil99]

Let us consider $P = \text{new } a((a.Q_1 + b.Q_2) \mid \bar{a}) \mid (\bar{b}.R_1 + \bar{a}.R_2)$.

$$
\begin{array}{c}
(a.Q_1 + b.Q_2) \mid \bar{a}.0 \rightarrow Q_1 \mid 0 \\
(a.Q_1 + b.Q_2) \mid \bar{a} \rightarrow Q_1 \\
\text{new } a ((a.Q_1 + b.Q_2) \mid \bar{a}) \rightarrow \text{new } a Q_1 \\
\text{new } a ((a.Q_1 + b.Q_2) \mid \bar{a}) \mid (\bar{b}.R_1 + \bar{a}.R_2) \rightarrow \text{new } a Q_1 \mid (\bar{b}.R_1 + \bar{a}.R_2)
\end{array}
$$
Example [Mil99]

Let us consider $P = \text{new } a((a.Q_1 + b.Q_2) | \bar{a}) | (\bar{b}.R_1 + \bar{a}.R_2)$.

\[
\begin{align*}
&\text{new } a ((a.Q_1 + b.Q_2) | \bar{a}) \rightarrow \text{new } a Q_1 \\
&\text{new } a ((a.Q_1 + b.Q_2) | \bar{a}) | (\bar{b}.R_1 + \bar{a}.R_2) \rightarrow \text{new } a Q_1 | (\bar{b}.R_1 + \bar{a}.R_2)
\end{align*}
\]
Example [Mil99]

Let us consider $P = \text{new } a((a.Q_1 + b.Q_2) \mid \bar{a}) \mid (\bar{b}.R_1 + \bar{a}.R_2)$.

\[
\begin{align*}
\text{React} & : (a.Q_1 + b.Q_2) \mid \bar{a}.0 \rightarrow Q_1 \mid 0 \\
\text{Struct} & : (a.Q_1 + b.Q_2) \mid \bar{a} \rightarrow Q_1 \\
\text{Res} & : \text{new } a ((a.Q_1 + b.Q_2) \mid \bar{a}) \rightarrow \text{new } a Q_1 \\
\text{Par} & : \text{new } a ((a.Q_1 + b.Q_2) \mid \bar{a}) \mid (\bar{b}.R_1 + \bar{a}.R_2) \rightarrow \text{new } a Q_1 \mid (\bar{b}.R_1 + \bar{a}.R_2)
\end{align*}
\]
Let us consider $P = \text{new } a((a.Q_1 + b.Q_2) \mid \bar{a}) \mid (\bar{b}.R_1 + \bar{a}.R_2)$.

\[
\begin{array}{c}
\frac{(a.Q_1 + b.Q_2) \mid \bar{a}.0 \rightarrow Q_1 \mid 0}{(a.Q_1 + b.Q_2) \mid \bar{a} \rightarrow Q_1} \\
\frac{\text{React}}{
\frac{\text{Struct}}{
\frac{\text{Res}}{
\text{Par}}}
\frac{\text{Par}}{
\text{Res}}}
\end{array}
\]
Process Calculi

Variants

Communicating Automata

Linking both semantics

Theorem

Reaction agrees with $\tau$-transition: $P \xrightarrow{\tau} P'$ if and only if $P \rightarrow P'$
Bisimulation

**Definition**

A binary relation $\mathcal{R}$ over processes is a **strong simulation** if, whenever $P \mathcal{R} Q$:

- if $P \xrightarrow{\alpha} P'$, then there exists $Q'$ such that $Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{R} Q'$.

Intuition: $P$ “simulates” $Q$, it is able to “follow” its transitions.

**Definition**

A **strong bisimulation** $\mathcal{R}$ is a simulation whose converse relation $\mathcal{R}^{-1}$ is also a simulation.

Example: Structural congruence is a strong bisimulation.
The definition of weak bisimulation is essentially the same as that of strong simulation except that the transition relation is replaced by a relation which makes it possible to ignore internal $\tau$ actions.

A process can be replaced by a process which behaves equivalently up to observable actions.
Actions are not only used to synchronize processes, they are also used as channels of communication, communicating values that are themselves channels:

- The structure of the system is dynamic.
- The expressive power is completely different: for instance, it is possible to encode the λ-calculus.
Syntax

\[
\pi ::= x(y) \quad \text{receive } y \text{ along } x \\
| \bar{x}(y) \quad \text{send } y \text{ along } x \\
| \tau \quad \text{unobservable action}
\]

\[
P ::= \sum_{i \in I} \pi_i.P_i \mid P_1 | P_2 \mid \text{new } x \mid P \mid !P
\]

Mutually recursive definitions are replaced by repetition (in the basic π-calculus): \(!P \equiv P|!P\).
Semantics (Reaction Rules)

\[
\begin{align*}
\text{TAU} & \quad \frac{}{\tau \cdot P + M \rightarrow P} \\
\text{REACT} & \quad \frac{\left(x(y).P + M \right) \vert \left(\overline{x}(z).Q + N \right) \rightarrow \{z/y\} P \vert Q}{(x(y).P + M) \vert (\overline{x}(z).Q + N) \rightarrow \{z/y\} P \vert Q} \\
\text{PAR} & \quad \frac{P \rightarrow P'}{P \vert Q \rightarrow P' \vert Q} \\
\text{RES} & \quad \frac{P \rightarrow P'}{\text{new } a \ P \rightarrow \text{new } a \ P'} \\
\text{STRUCT} & \quad \frac{P \rightarrow P'}{Q \rightarrow Q'} \quad \text{if } P \equiv Q \text{ and } P' \equiv Q'
\end{align*}
\]
The asynchronous $\pi$-calculus is defined as a subset of the $\pi$-calculus where:

- There is no output prefixing (a process may only output a value and stop).
- There is no output in choices (in order to avoid synchronization, in particular in a distributed setting, at the implementation level).

It is “almost” as expressive as the $\pi$-calculus.
Example: the join-calculus [FG96, FG02]

■ Syntax

\[
P ::= x(u) \quad \text{message send}
\]

\[
P_1 | P_2 \quad \text{parallel composition}
\]

\[
def x(u) | y(v) \triangleright P_1 \text{ in } P_2
\]

A process and its channels are jointly defined in a construct that looks like a function definition (the scope of \( u \) and \( v \) is \( P_1 \), the scope of \( x \) and \( y \) the whole definition).

■ Informal semantics: the reception of a message on both \( u \) and \( v \) (join pattern) spawns a process \( P_1 \) and proceeds with \( P_2 \).
Some other interesting topics

- **Higher-order** vs first-order process calculi (it is possible to send process expressions rather than simply names over channels)
- Reconciling the actor model [HBS73, Agh86] and process calculi [AT04]
- **The ambient calculus** [CG98] (the focus is on movement rather than communication)
Current Research

- Developing new calculi that better capture (some aspects of) computation
- Improving the capabilities for reasoning on processes:
  - “Well-behaved” subcalculi (with stronger properties)
  - Behavioral theory
  - Specific logics
- Understanding the relative expressivity of process calculi (using encodings)
What can we do with all this?

- **Analysis**: extract the behavior of an existing system and analyze its properties.
- **Synthesis (model-driven development)**: model new systems and derive their implementation (with an objective of correction by construction)
- **Program language design**: improve current support for concurrency; reduce the gap between the models and the implementation. Examples:
  - Pict [PT97], based on the $\pi$-calculus
  - JoCaml [MM07] (http://jocaml.inria.fr/)
  - C$\omega$ (http://research.microsoft.com/Comega/)
G. Agha. 
*Actors: A Model of Concurrent Computation in Distributed Systems.*

Gul Agha and Prasanna Thati. 
An algebraic theory of actors and its application to a simple object-based language. 

Luca Cardelli and Andrew D. Gordon. 
Mobile ambients. 


Kohei Honda and Mario Tokoro.
An object calculus for asynchronous communication.

Robin Milner.
Elements of interaction: Turing award lecture.

Robin Milner.
*Communicating and Mobile Systems: the π-Calculus.*

J. Magee and J. Kramer.
*Concurrency: State Models and Java.*

Louis Mandel and Luc Maranget.
*The JoCaml language - Documentation and user’s manual.*
INRIA, July 2007.
Release 3.10.
