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

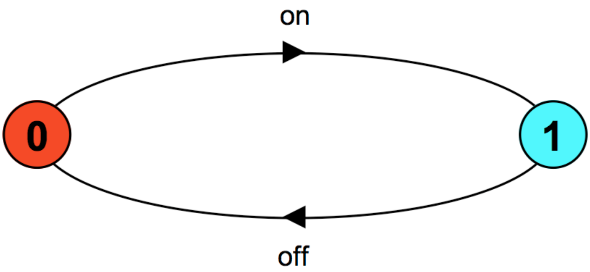
- A family of approaches to formally model concurrent systems: interaction, communication, and synchronization between independent processes (or agents).
- Algebraic laws make it possible to manipulate and reason about these models (in particular in terms of their behavioral equivalence).

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 π-calculus [MPW92, Mil93, Mil99]
- The asynchronous π-calculus [HT91]

Modelling Sequential Processes with LTSs

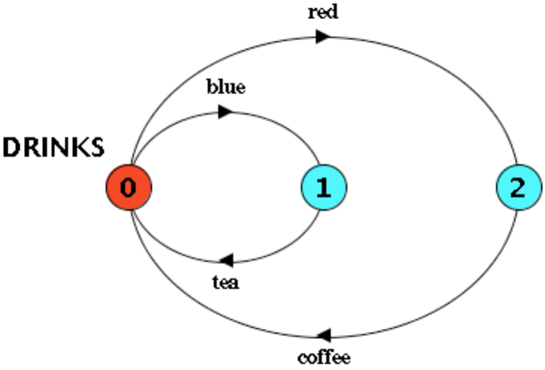
A (sequential) process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions. [MK06]



The corresponding sequence of actions (there is only one) or trace:

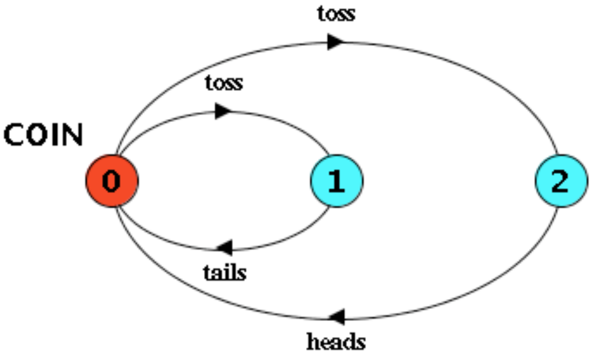
`on -> off -> on -> off -> on -> off ...`

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

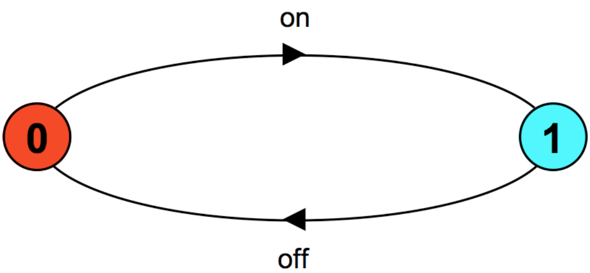



```
toss -> tails -> toss -> tails -> toss -> tails ...
toss -> heads -> toss -> tails -> toss -> tails ...
...
toss -> heads -> toss -> heads -> toss -> heads ...
```

(Basic) FSP Syntax

$$
\begin{array}{lll}
\mathit{ProcessDefinition} & ::= & \mathit{ProcessName} = \mathit{ProcessExpression} \\[1em]
\mathit{ProcessExpression} & ::= & \mathit{ProcessName} \mid \mathit{ActionPrefix} \mid \mathit{Choice} \\[1em]
\mathit{ActionPrefix} & ::= & (\ \mathit{Action} \texttt{ -> } \mathit{ProcessExpression}\ ) \\[1em]
\mathit{Choice} & ::= & (\ \mathit{Action} \texttt{ -> } \mathit{ProcessExpression} \\
 & & \ \ |\ \mathit{Action} \texttt{ -> } \mathit{ProcessExpression}\ )
\end{array}
$$

Recursive Definitions and Action Prefixes

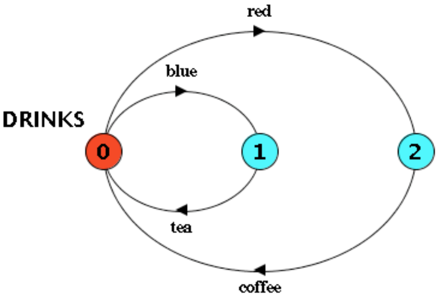



```
SWITCH = OFF, OFF = (on -> ON), ON = (off-> OFF).
```

or

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

Choices




```
DRINKS = ( red -> coffee -> DRINKS
         | blue -> tea -> DRINKS
         ).
```

Syntactic Sugar: Indexed Actions

```
BUFF = (write[i:0..3]->read[i]-> BUFF).
```

is equivalent to:

```
BUFF = ( write[0]->read[0]->BUFF
       | write[1]->read[1]->BUFF
       | write[2]->read[2]->BUFF
       | write[3]->read[3]->BUFF
       ).
```

Note: | is commutative and associative.

Syntactic Sugar: Indexed Processes

```
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]),
BUFF[1] = ( read[1] -> BUFF[1]
          | write[0] -> BUFF[0]
          | write[1] -> BUFF[1]).
```

LTS

Definition

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.

Definition

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 $(q, a, q') \in \Delta$.

Association Rules

The semantics is given by associating an LTS to each process expression: $lts : ProcessExpression \rightarrow LTS$

$$\textsc{Definition}\ \frac{P = E}{lts(P) = lts(E)}$$

$$\textsc{Prefix}\ \frac{lts(E) = < S, A, \Delta, q >}{lts(a \rightarrow E) = < S \cup \{p\}, A \cup \{a\}, \Delta \cup \{(p, a, q)\}, p > \text{ where } p \notin S}$$

$$\textsc{Choice}\ \frac{lts(E_1) = < S_1, A_1, \Delta_1, q_1 > \qquad lts(E_2) = < S_2, A_2, \Delta_2, q_2 >}{\begin{array}{rl} lts(a_1 \rightarrow E_1 \mid a_2 \rightarrow E_2) = & < S \cup \{p\}, A_1 \cup A_2 \cup \{a_1, a_2\}, \\ & \Delta \cup \{(p, a_1, q_1), (p, a_2, q_2)\}, p > \\ & \text{where } p \notin S \end{array}}$$

Parallel Composition

Parallel composition construct: $(\mathit{ProcessName} \mid\mid \mathit{ProcessName})$

```
P = ...
Q = ...
||PQ = (P || Q).
```

Note: in FSP, it is not possible to mix the definition of sequential processes and parallel processes.

Semantics

There is a new association rule for parallel composition:

$$\textsc{ParallelComposition}\ \frac{}{lts(P \mid\mid Q) = lts(P) \mid\mid lts(Q)}$$

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 $L_2 =< S_2, A_2, \Delta_2, q_2 >$.
$L_1 \mid\mid L_2 =< S_1 \times S_2, A_1 \cup A_2, \Delta, (q_1, q_2) >$, where Δ is the smallest relation satisfying the following rules:

$$\frac{L_1 \xrightarrow{a} L_1' \qquad a \notin A_2}{L_1 \mid\mid L_2 \xrightarrow{a} L_1' \mid\mid L_2}$$

$$\frac{L_2 \xrightarrow{a} L_2' \qquad a \notin A_1}{L_1 \mid\mid L_2 \xrightarrow{a} L_1 \mid\mid L_2'}$$

$$\frac{L_1 \xrightarrow{a} L_1' \qquad L_2 \xrightarrow{a} L_2'}{L_1 \mid\mid L_2 \xrightarrow{a} L_1' \mid\mid L_2'} \; a \in A_1 \cup A_2, \text{ it is a shared action}$$

Algebraic laws

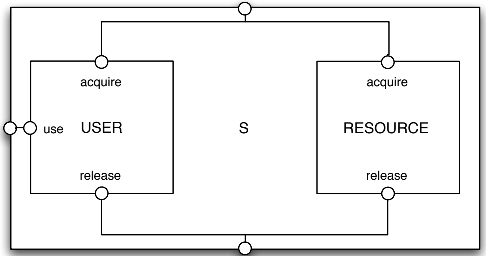
- $||$ is commutative: $P||Q = Q||P$
- $||$ is associative: $(P||Q)||R = P||(Q||R)$

This gives n-ary synchronization on shared actions.

A structural/component view of composition

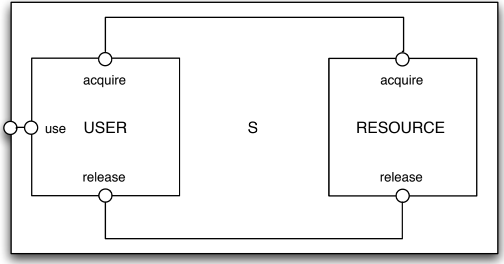
The alphabet of a process is its interface, its definition is its implementation.

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



Hiding actions

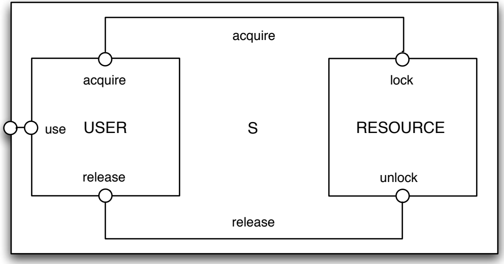
```
||S = (USER || RESOURCE)\{acquire, release}.
```



This creates τ transitions in the underlying LTS.

Relabeling actions

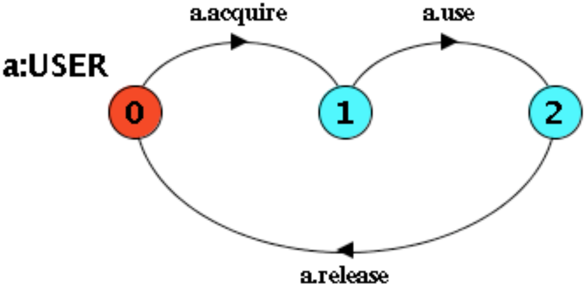
```
RESOURCE = (lock->unlock->RESOURCE).
USER = (acquire->use->release->USER).
||S = (USER || RESOURCE)/{acquire/lock, release/unlock}.
```



Relabeling processes

All the transitions of a sequential process can be prefixed (eg to create some kind of "instances").

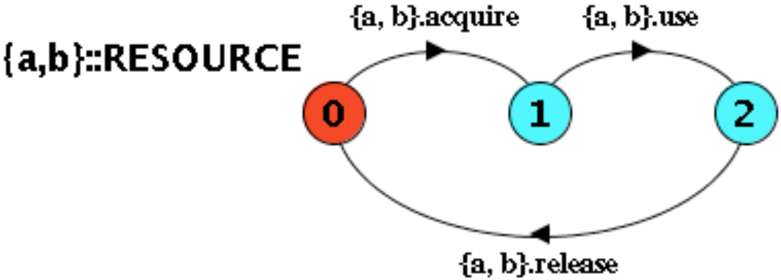
```
RESOURCE = (lock->unlock->RESOURCE).
USER = (acquire->use->release->USER).
||S = (a:USER || b:USER || RESOURCE).
```



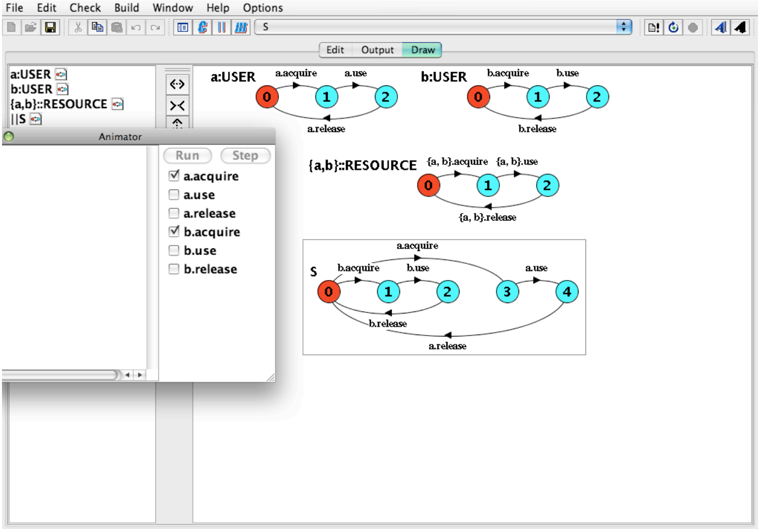
Set prefixing

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).
```



A quick demo?


File Edit Check Build Window Help Options
S
Edit Output Draw
a:USER
b:USER
{a,b}::RESOURCE
||S
Animator
Run Step
a.acquire
a.use
a.release
b.acquire
b.use
b.release
a:USER a.acquire a.use a.release
b:USER b.acquire b.use b.release
{a,b}::RESOURCE {a, b}.acquire {a, b}.use {a, b}.release
S a.acquire b.acquire b.use a.use b.release a.release

Communicating Automata [Mil99]

- Binary synchronization through complementary actions a and $\overline{a}$
- Lean syntax
- Semantics given by either:
  - Transition rules
  - Reaction rules (à la Chemical Abstract Machine)

Syntax

- There is no layering of sequential and parallel processes
- Processes are parameterized by their actions (relabeling)
- `new` restricts the scope of an action (hiding)

$$D ::= A(\overrightarrow{a}) = P_A$$
$$P ::= A\langle \overrightarrow{a} \rangle \mid \sum_{i \in I} \alpha_i . P_i \mid P_1 | P_2 \mid \texttt{new}\ a\ P$$

Labelled Semantics

$$\textsc{Sum}_t \; \frac{}{M + \alpha.P + N \xrightarrow{\alpha} P}$$

$$\textsc{React}_t \; \frac{P \xrightarrow{\lambda} P' \qquad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{L-Par}_t \; \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \textsc{R-Par}_t \; \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\textsc{Res}_t \; \frac{P \xrightarrow{\alpha} P'}{\texttt{new}\ a\ P \xrightarrow{\alpha} \texttt{new}\ a\ P'} \text{ if } \alpha \notin \{a, \overline{a}\}$$

$$\textsc{Ident}_t \; \frac{\{\overrightarrow{b}/\overrightarrow{a}\}P_A \xrightarrow{\alpha} P'}{A < \overrightarrow{b} > \xrightarrow{\alpha} P'} \text{ if } A(\overrightarrow{a}) = P_A$$

Semantics - Structural Congruence

Definition

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 (`new`)
- $P \equiv Q$ modulo reordering choices
- $P \equiv Q$ modulo reordering parallel composition (including $P \mid 0 \equiv P$)
- restrictions
  - $\texttt{new}\ a\ (P|Q) \equiv P|\texttt{new}\ a\ Q$ if a is not free in P
  - $\texttt{new}\ a\ 0 \equiv 0$
  - $\texttt{new}\ a\ (\texttt{new}\ b\ P) \equiv \texttt{new}\ b\ (\texttt{new}\ a\ P)$
- $A < \overrightarrow{b} > \equiv \{\overrightarrow{b}/\overrightarrow{a}\}P_A$ if $A(\overrightarrow{a}) = P_A$

Semantics - Reaction Rules

$$\textsc{Tau}\ \frac{}{\tau.P + M \rightarrow P}$$

$$\textsc{React}\ \frac{}{(a.P + M)|(\overline{a}.Q + N) \rightarrow P|Q}$$

$$\textsc{Par}\ \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\textsc{Res}\ \frac{P \rightarrow P'}{\texttt{new}\ a\ P \rightarrow \texttt{new}\ a\ P'}$$

$$\textsc{Struct}\ \frac{P \rightarrow P'}{Q \rightarrow Q'}\ \text{if}\ P \equiv Q\ \text{and}\ P' \equiv Q'$$

Example [Mil99]

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

$$\dfrac{\dfrac{\dfrac{}{(a.Q_1 + b.Q_2) \mid \overline{a}.0 \rightarrow Q_1 \mid 0}\ \textsc{React}}{(a.Q_1 + b.Q_2) \mid \overline{a} \rightarrow Q_1}\ \textsc{Struct}}{\dfrac{\mathtt{new}\ a\ ((a.Q_1 + b.Q_2) \mid \overline{a}) \rightarrow \mathtt{new}\ a\ Q_1}{\mathtt{new}\ a\ ((a.Q_1 + b.Q_2) \mid \overline{a}) \mid (\overline{b}.R_1 + \overline{a}.R_2) \rightarrow \mathtt{new}\ a\ Q_1 \mid (\overline{b}.R_1 + \overline{a}.R_2)}\ \textsc{Par}}\ \textsc{Res}$$

Example [Mil99]

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

$$
\dfrac{\dfrac{\dfrac{}{(a.Q_1 + b.Q_2) \mid \overline{a}.0 \rightarrow Q_1 \mid 0}\ \textsc{React}}{(a.Q_1 + b.Q_2) \mid \overline{a} \rightarrow Q_1}\ \textsc{Struct}}{\dfrac{\texttt{new}\ a\ ((a.Q_1 + b.Q_2) \mid \overline{a}) \rightarrow \texttt{new}\ a\ Q_1}{\texttt{new}\ a\ ((a.Q_1 + b.Q_2) \mid \overline{a}) \mid (\overline{b}.R_1 + \overline{a}.R_2) \rightarrow \texttt{new}\ a\ Q_1 \mid (\overline{b}.R_1 + \overline{a}.R_2)}\ \textsc{Par}}\ \textsc{Res}
$$

Example [Mil99]

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(a.Q_1 + b.Q_2) \mid \overline{a}.0 \rightarrow Q_1 \mid 0}\ \textsc{React}}{(a.Q_1 + b.Q_2) \mid \overline{a} \rightarrow Q_1}\ \textsc{Struct}}{\mathtt{new}\ a\ ((a.Q_1 + b.Q_2) \mid \overline{a}) \rightarrow \mathtt{new}\ a\ Q_1}\ \textsc{Res}}{\mathtt{new}\ a\ ((a.Q_1 + b.Q_2) \mid \overline{a}) \mid (\overline{b}.R_1 + \overline{a}.R_2) \rightarrow \mathtt{new}\ a\ Q_1 \mid (\overline{b}.R_1 + \overline{a}.R_2)}\ \textsc{Par}$$

Example [Mil99]

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

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{(a.Q_1 + b.Q_2) \mid \overline{a}.0 \rightarrow Q_1 \mid 0}\ \textsc{React}}{(a.Q_1 + b.Q_2) \mid \overline{a} \rightarrow Q_1}\ \textsc{Struct}}{\texttt{new}\ a\ ((a.Q_1 + b.Q_2) \mid \overline{a}) \rightarrow \texttt{new}\ a\ Q_1}\ \textsc{Res}}{\texttt{new}\ a\ ((a.Q_1 + b.Q_2) \mid \overline{a}) \mid (\overline{b}.R_1 + \overline{a}.R_2) \rightarrow \texttt{new}\ a\ Q_1 \mid (\overline{b}.R_1 + \overline{a}.R_2)}\ \textsc{Par}
$$

Linking both semantics

Theorem

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

Bisimulation

Definition

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

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

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.

Weak 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 τ actions.
- A process can be replaced by a process which behaves equivalently up to observable actions.

The π-calculus [Mil99]

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

$$
\begin{array}{llll}
\pi & ::= & x(y) & \text{receive } y \text{ along } x \\
 & | & \overline{x}(y) & \text{send } y \text{ along } x \\
 & | & \tau & \text{unobservable action} \\
P & ::= & \multicolumn{2}{l}{\sum_{i \in I} \pi_i . P_i \mid P_1 | P_2 \mid \texttt{new}\ x\ P \mid\ !P}
\end{array}
$$

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

Semantics (Reaction Rules)

$$\textsc{Tau}\ \frac{}{\tau.P + M \rightarrow P}$$

$$\textsc{React}\ \frac{}{(x(y).P + M)|(\overline{x}(z).Q + N) \rightarrow \{z/y\}P|Q}$$

$$\textsc{Par}\ \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\textsc{Res}\ \frac{P \rightarrow P'}{\texttt{new}\ a\ P \rightarrow \texttt{new}\ a\ P'}$$

$$\textsc{Struct}\ \frac{P \rightarrow P'}{Q \rightarrow Q'}\ \text{if } P{\equiv}Q \text{ and } P'{\equiv}Q'$$

The Asynchronous π-calculus

The asynchronous π-calculus is defined as a subset of the π-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 π-calculus.

Example: the join-calculus [FG96, FG02]

- Syntax

$$
\begin{array}{lll}
P ::= & x\langle u\rangle & \text{message send} \\
\mid & P_1|P_2 & \text{parallel composition} \\
\mid & \mathtt{def}\ x(u)|y(v) \triangleright P_1\ \mathtt{in}\ P_2 &
\end{array}
$$

  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

- Developping 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 π-calculus
  - JoCaml [MM07] (`http://jocaml.inria.fr/`)
  - Cω (http://research.microsoft.com/Comega/)

G. Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
MIT Press, 1986.

Gul Agha and Prasanna Thati.
An algebraic theory of actors and its application to a simple object-based language.
In *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, pages 26–57. Springer-Verlag, 2004.

Luca Cardelli and Andrew D. Gordon.
Mobile ambients.
In Maurice Nivat, editor, *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

Cédric Fournet and Georges Gonthier.
The reflexive CHAM and the join-calculus.
In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg, FL, USA, January 1996. ACM Press.

Cédric Fournet and Georges Gonthier.
The join calculus: A language for distributed mobile programming.
In *Applied Semantics, International Summer School, APPSEM 2000*, pages 268–332. Springer-Verlag, 2002.
Advanced Lectures.

Carl Hewitt, Peter Bishop, and Richard Steiger.
A universal modular actor formalism for artificial intelligence.
In *IJCAI*, pages 235–245, 1973.

Kohei Honda and Mario Tokoro.

An object calculus for asynchronous communication.
In Pierre America, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag.

Robin Milner.
Elements of interaction: Turing award lecture.
Communications of the ACM, 36(1):78–89, 1993.

Robin Milner.
Communicating and Mobile Systems: the π-Calculus.
Cambridge University Press, 1999.

J. Magee and J. Kramer.
Concurrency: State Models and Java.
Wiley, 2nd edition, 2006.

Louis Mandel and Luc Maranget.
The JoCaml language - Documentation and user's manual.

INRIA, July 2007.
Release 3.10.

Robin Milner, Joachim Parrow, and David Walker.
A calculus of mobile processes, I.
Information and Computation, 100(1):1–40, 1992.

Benjamin C. Pierce and David N. Turner.
Pict: A programming language based on the pi-calculus.
In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1997.