SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 11

Prof. Daniele Gorla

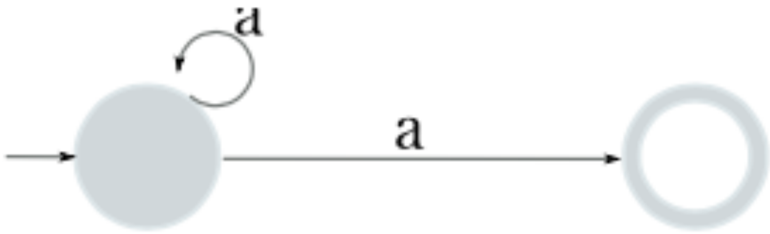
LTSs and Bisimulation

Behaviour of a concurrent system

→ set of traces (hystories)

→ set of traces + branching structure

Ex.:



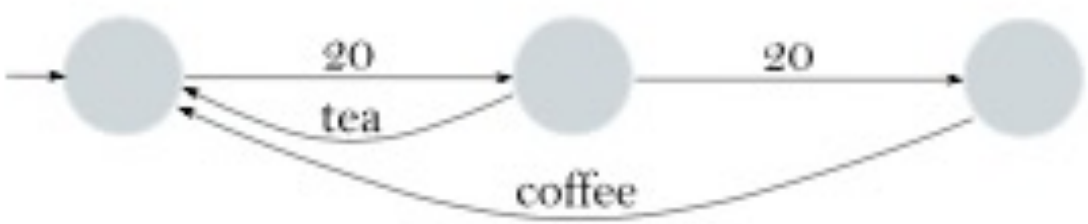
A (finite non-deterministic) automaton is a quintuple M = (Q,Act,q0,F,T), where:

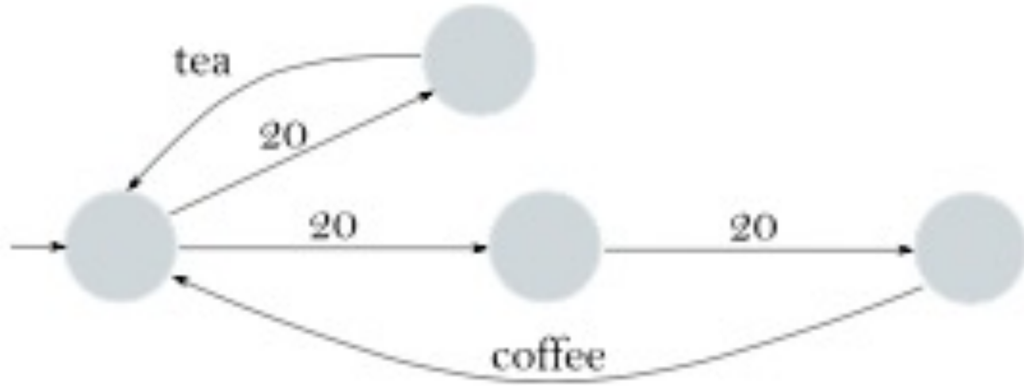
- Q is the set of states,
- Act is the set of actions,
- q0 is the starting state,
- F is the set of final states,
- T is the transition relation ($T \subseteq Q \times Act \times Q$).

Automata Behaviour: language equivalence

(where L(M) is the set of all the sequences of input characters that bring the automaton M from its starting state to a final one)

M1 and M2 are *language equivalent* if and only if L(M1)=L(M2)

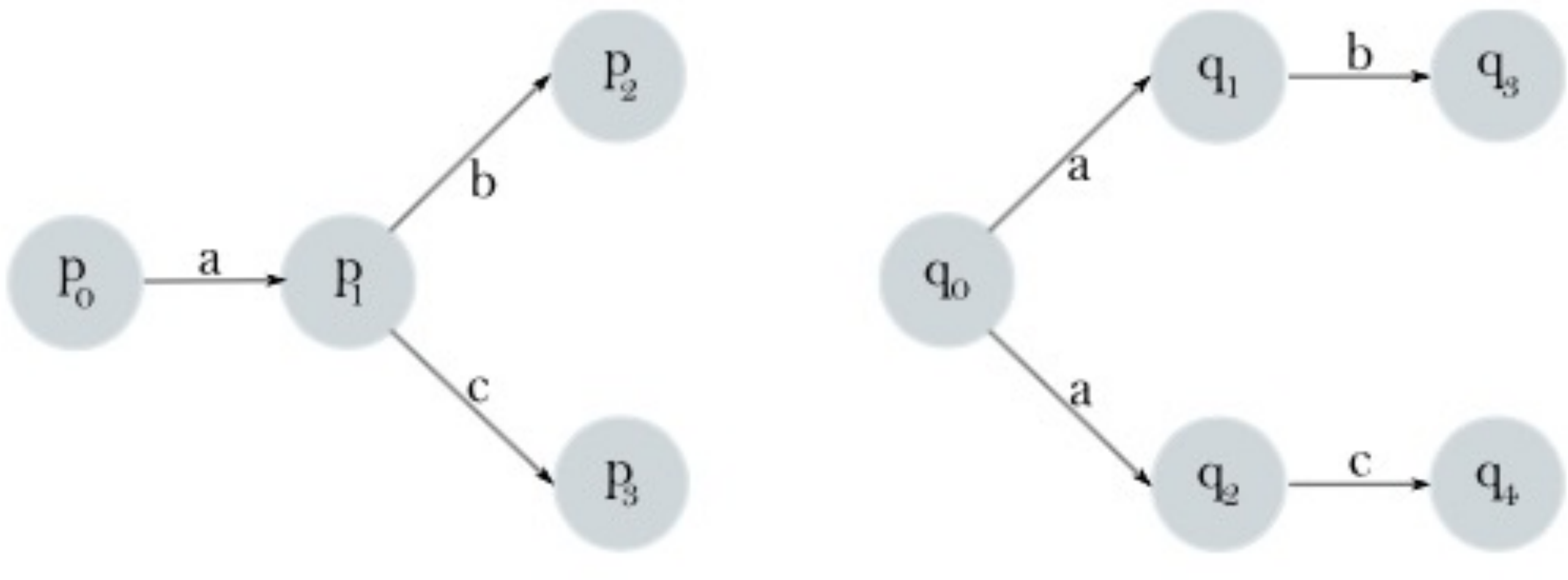




By considering the starting states as also final, they both generate the same language, i.e.:

$$(20.(tea + 20.coffee))^* = (20.tea + 20.20.coffee)^*$$

But, do they bahave the same from the point of view of an external observer??



The essence of the difference is WHEN the decision to branch is taken

→ language equivalence gets rid of branching points

→ *it is too coarse for our purposes!*

LTSs

In concurrency theory, we don't use finite automata but *Labeled Transition System (LTS)*. The main differences between the two formalisms are:

- automata usually rely on a finite number of states, whereas states of an LTS can be infinite;
- automata fix one starting state, whereas in an LTS every state can be considered as initial (this corresponds to different possible behaviors of the process);
- automata rely on final states for describing the language accepted, whereas in LTS this notion is not very informative.

Fix a set of action names (or, simply, actions), written N.

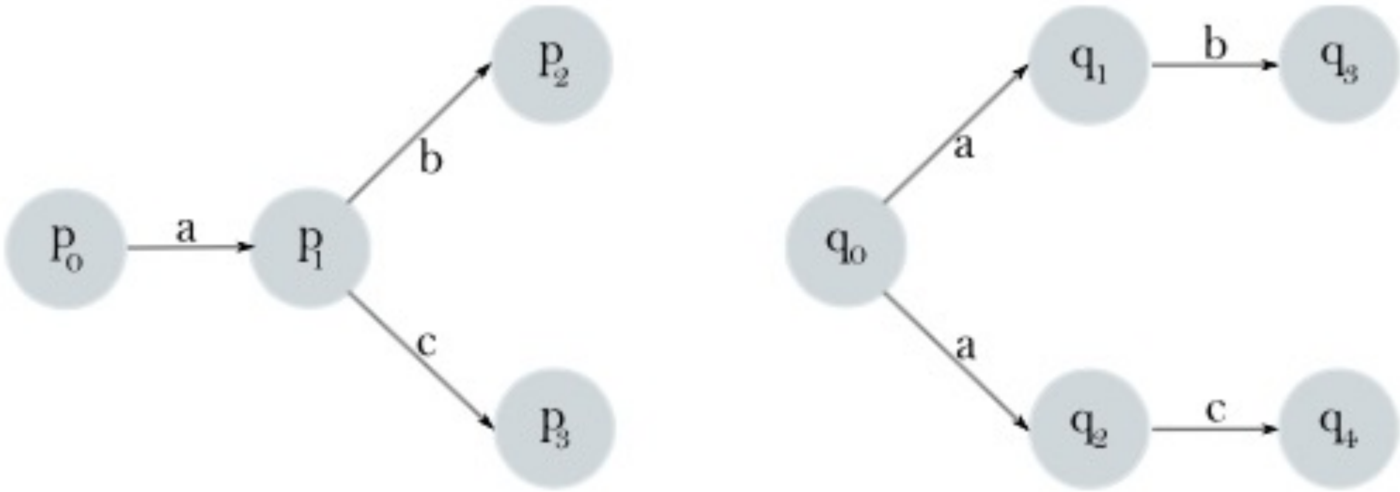
> A Labeled Transition System (LTS) is a pair (Q, T), where Q is the set of states and T is the transition relation ($T \subseteq Q \times N \times Q$).

We shall usually write s –a–> s′ instead of $\langle s,a,s' \rangle \in T$.

Bisimulation

Intuitively, two states are equivalent if they can perform the same actions that lead them in states where this property still holds

Ex.



P0 and Q0 are different because, after an a, the former can decide to do b or c, whereas the latter must decide this before performing a

Let (Q,T) be an LTS.

A binary relation $S \subseteq Q \times Q$ is a *simulation* if and only if

$$\forall (p,q) \in S \, \forall p \xrightarrow{a} p' \, \exists q \xrightarrow{a} q' \text{ s.t. } (p',q') \in S$$

We say that p is simulated by q if there exists a simulation S such that

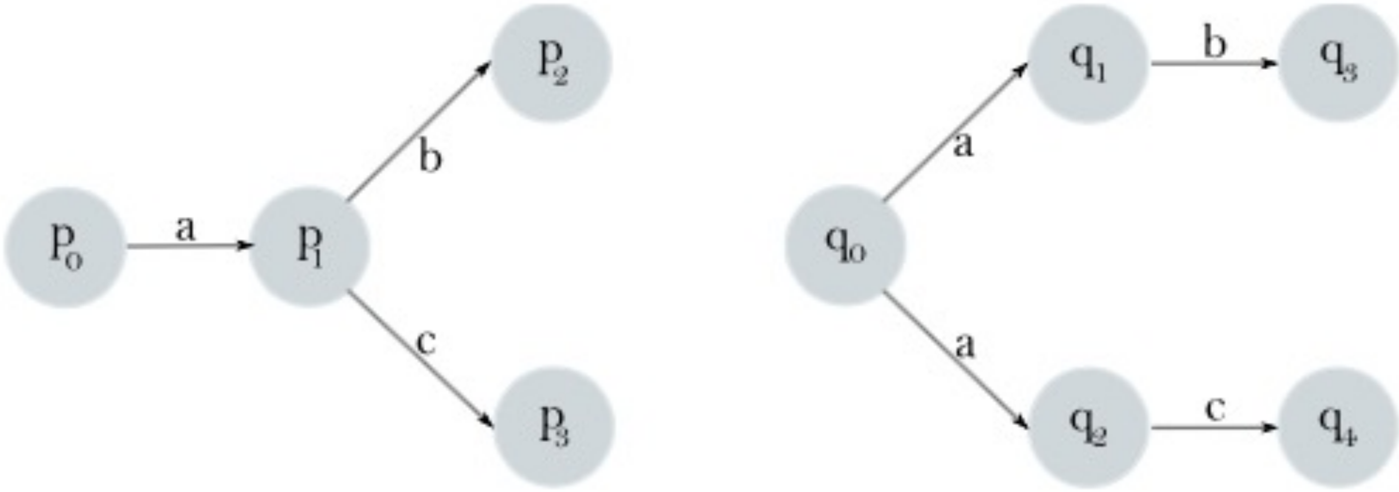
$$(p,q) \in S.$$

We say that S is a *bisimulation* if both S and S^{-1} are simulations (where

$$S^{-1} = \{(p,q) : (q,p) \in S\}).$$

Two states q and p are bisimulation equivalent (or, simply, bisimilar) if there exists a bisimulation S such that $(p, q) \in S$; we shall then write $p \sim q$.

Remark: (bi)simulation has been defined as a relations on the states of a single LTS. This is not a limitation since, given two LTSs, we can take their disjoint union and work on a relation that relates the state of the resulting (unique) LTS.



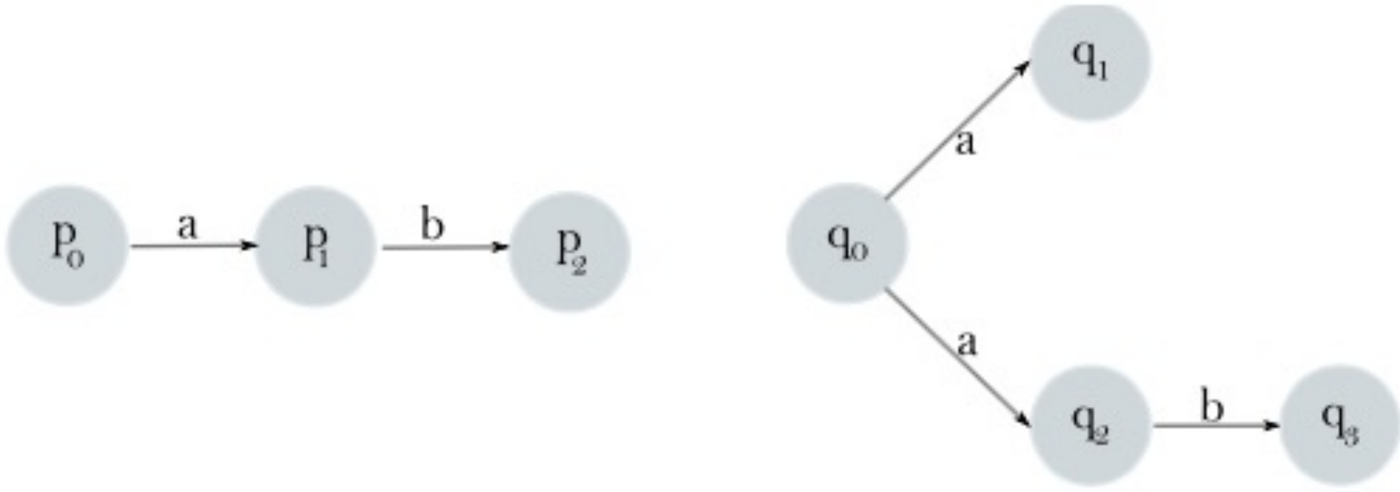
q0 is simulated by p0; this is shown by the following simulation relation:

S = {(q0,p0), (q1,p1), (q2,p1), (q3,p2), (q4,p3)}

To let p0 be simulated by q0, we should have that p1 is simulated by q1 or q2. If S contained one among (p1,q1) or (p1,q2), then it would not be a simulation: indeed, p1 can perform both a c (whereas q1 cannot) and a b (whereas q2 cannot)

Remark: for proving equivalence, it is NOT enough to find a simulation of p by q and a simulation of q by p

EX.:



p0 is simulated by q0:

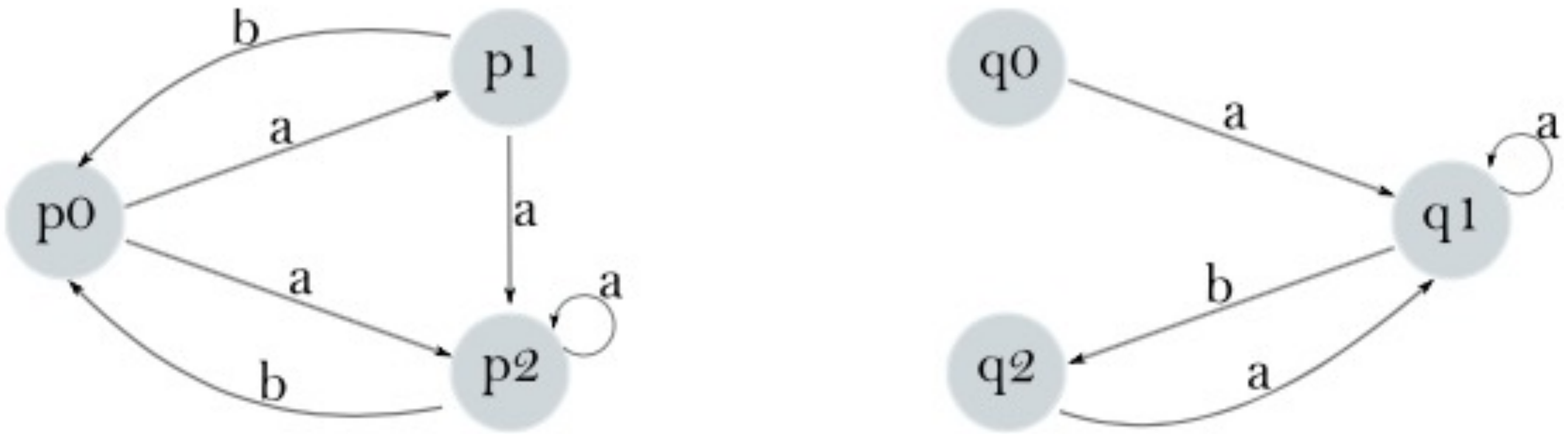
S = {(p0, q0), (p1, q2), (p2, q3)}

q0 is simulated by p0:

S′ ={(q0,p0),(q1,p1),(q2,p1),(q3,p2)}

BUT p0 and q0 are not bisimilar: the transition q0 –a–> q1 is not bisimulable by any transition from p0 (indeed, p0 –a–> p1 does not suffice, because p1 can perform a b and so cannot be bisimilar to q1)

An example of bisimilarity



To prove that p0 $\sim$ q0, it sufficies to check that the following relations are simulations:

S = {(p0,q0), (p1,q1), (p2,q1), (p0,q2)}

S^{-1} = {(q0,p0), (q1,p1), (q1,p2), (q2,p0)}

Thm: Bisimilarity is an equivalence relation.

Proof:

Reflexivity: we have to show that q ~ q, for every q. Consider the following relation

$$S = \{(p,p) : p \in Q\}$$

and observe that it is a bisimulation (it is a simulation, as well as its inverse – i.e., S itself).

Symmetry: we have to show that p ~ q implies q ~ p, for every p,q. By hypothesis, there exists a bisimulation S that contains the pair (p, q). By definition of bisimulation, S^{-1} is a simulation; hence, $(q,p) \in S^{-1}$ and, consequently, q ~ p.

Transitivity: we have to show that p ~ q and q ~ r imply p ~ r, for all p,q,r. Let us consider the following relation:

$$S = \{(x,z) : \exists y \text{ s.t. } (x,y) \in S1 \wedge (y,z) \in S2\}$$

where S1 and S2 are bisimulations; let us show that S is a bisimulation.

- Let (x, z) ∈ S and x –a–> x′.
- If (x, z) belongs to S, then, by definition, there exists y such that (x,y) ∈ S1 and (y,z) ∈ S2.
- Since S1 is a bisimulation, there exists y –a–> y′ such that (x′, y′) ∈ S1.
- Since S2 is a bisimulation, there exists z –a–> z′ such that (y′, z′) ∈ S2.
- Hence, from x –a–> x′, we found z –a–> z′ such that (x′, z′) ∈ S, because there exists a y′ such that (x′, y′) ∈ S1 and (y′, z′) ∈ S2.

QED

Thm.: $\sim$ is a bisimulation.
Proof:

The proof is done by showing that $\sim$ is a simulation.

By definition of similarity, we have to show that

$$\forall (p,q) \in \sim \ \forall p \xrightarrow{a} p' \ \exists q \xrightarrow{a} q' \text{ s.t.} (p',q') \in \sim$$

Let us fix a pair $(p,q) \in \sim$

Bisimilarity of p and q implies the existence of a bisimulation S such that $(p,q) \in S$.

Hence, for every transition $p \xrightarrow{a} p'$, there exists a transition $q \xrightarrow{a} q'$ such that $(p', q') \in S$.

So, $(p',q') \in \sim$

QED

Thm.: For every bisimulation S, it holds that $S \subseteq \sim$.
Proof:

Let $(p,q) \in S$. Then, there exists a bisimulation that contains the pair (p, q);

thus, $(p, q) \in \sim$.

QED

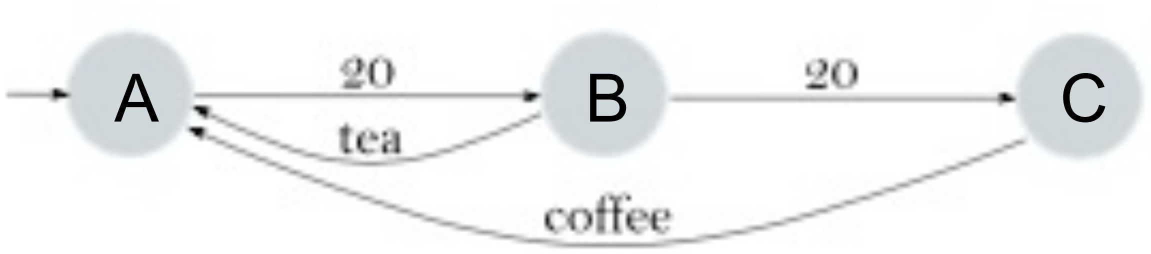
A syntax for non-deterministic processes

LTSs are a very natural formalism for pictorially describe the behavior of 'small' processes.

However, when the number of states and/or transitions grows up (potentially, it could also become infinite), this notation is not natural anymore.

The same happens for regular languages, where we use regular grammars/expressions instead of finite automata, when the language becomes too complex.

In our case, we now provide a syntax for processes that will allow us to write down LTSs in a more compact and readable way.



If we call A,B and C its states, we can describe its behavior with the following system of equations:

$$A = 20.B$$
$$B = tea.A + 20.C$$
$$C = coffee.A$$

where '.' denotes sequential composition and '+' non-deterministic choice.

By replacing the third equation in the second one and then the result in the first one, we obtain the following recursive definition of the machine behavior:

$$A = 20.(tea.A + 20.coffee.A)$$

The only ingredients we used to write down an LTS are:

- sequential compsition (of an action and a process),
- non-deterministic choice (between a finite set of prefixed processes), and
- recursion

To simplify process writing, we shall assume to have a finite set *Id* of processes identifiers and that the definitions we shall give will be parametric

For every identifier (denoted with capital letters A,B,..), we shall assume a unique definition of the form

$$A(x1,x2,..,xn) := P$$

where names x1,x2,..,xn are all distinct and all included in the names of P.

Let us denote with P{b1/x1 . . . bn/xn} the process obtained from P by replacing name xi with name bi, for every i = 1,..., n.

Examples

A = 20.(tea.A + 20.coffee.A)
is the coffee machine seen before
(process definition without parameters)

A(x,y) = 20.(x.A(x,y) + 20.y.A(x,y))
is the previous machine, parametric in the products delivered
(e.g., A(tea,coffee) is the original machine, A(bread,croissant) is a food delivery machine)

A(x,y,z) = z.(x.A(x,y,z) + z.y.A(x,y,z))
is the same machine, where also the value of the coin is a parameter
(A(tea,coffee,20) returns the original machine)

The set of non-deterministic processes is given by the following grammar:

$$P \ ::= \ \sum_{i \in I} \alpha_i.P_i \mid A(a_1 \ldots a_n)$$

where I is a finite set of indices and $\alpha_i \in \mathcal{A}$, for every $i \in I$.

Remark: we now fuse together in a unique operator sequential composition and nondeterministic choice.

If the index set I is empty, then $\Sigma_{i \in I} \alpha_i.P_i$ is the terminated process, that cannot perform any action; this process will be represented with the symbol **0**

We shall usually omit tail occurrences of '.**0**' and, for example, simply write a.b instead of a.b.**0**

From the syntax to the LTS

We have shown how it is possible, starting from an LTS, to generate a corresponding process

It is also possible the inverse translation and then the two formalisms do coincide; the rules that have to be used in this translation are:

$$\sum_{i\in I} \alpha_i.\mathcal{P}_i \xrightarrow{\alpha_j} \mathcal{P}_j \quad \text{for all } j \in I$$

$$\frac{P\{a_1/x_1 \ldots a_n/x_n\} \xrightarrow{\alpha} P'}{A(a_1 \ldots a_n) \xrightarrow{\alpha} P'} \; A(x_1 \ldots x_n) \triangleq P$$

Examples

$A(x,y) = 20.(x.A(x,y) + 20.y.A(x,y))$

Infer the transitions from a state associated to A(tea,coffee)

$$\frac{20.(tea.A(tea,coffee)+20.coffee.A(tea,coffee)) \xrightarrow{20} tea.A(tea,coffee)+20.coffee.A(tea,coffee)}{A(tea,coffee) \xrightarrow{20} tea.A(tea,coffee)+20.coffee.A(tea,coffee)}\ A(x,y)=20.(x.A(x,y)+20.y.A(x,y))$$

$B(x,y) = x.A(x,y) + 20.y.A(x,y)$

Infer the transitions from a state associated to B(tea,coffee)

$$\frac{tea.A(tea,coffee)+20.coffee.A(tea,coffee) \xrightarrow{tea} A(tea,coffee)}{B(tea,coffee) \xrightarrow{tea} A(tea,coffee)}\ B(x,y)=x.A(x,y)+20.y.A(x,y)$$

$$\frac{tea.A(tea,coffee)+20.coffee.A(tea,coffee) \xrightarrow{20} coffee.A(tea,coffee)}{B(tea,coffee) \xrightarrow{20} coffee.A(tea,coffee)}\ B(x,y)=x.A(x,y)+20.y.A(x,y)$$

EXAMPE: *counter for natural numbers*

there is a process C_0 that simulates the zero (it can have successors but not predecessors)

for every i > 0, there is a process C_i that can be incremented and decremented.

Assuming actions *inc* and *dec*, this can be modeled by having:

$$C_0 = inc.C_1$$
$$C_i = inc.C_{i+1} + dec.C_{i-1} \qquad \text{for every } i > 0$$

By using the inference rules, the resulting LTS is

$$C_0 \underset{dec}{\overset{inc}{\rightleftarrows}} C_1 \underset{dec}{\overset{inc}{\rightleftarrows}} C_2 \underset{dec}{\overset{inc}{\rightleftarrows}} C_3 \ldots$$

Notice that this LTS has infinite states!

EXAMPLE: *queue of booleans*

Dimension = 2

hence, a generic state of the buffer (described by the sequence of values currently memorized in the buffer) belongs to the set {ε, 0, 1, 00, 01, 10, 11}, where ε denotes an empty sequence.

Let i and j be elements of {0, 1}; we shall use the following notation:

- B_ε is the empty buffer;
- B_i is the buffer containing only the bit i;
- B_{ij} is the buffer containing the bits i and j (the order reflects the insertion order).

If we denote with in_i/out_i the actions of insertion/extraction of the bit i in/from the buffer, we can define the process defining equations in the following way:

- $B_\epsilon = \sum_{i\in\{0,1\}} in_i.B_i$
- $B_i = \sum_{j\in\{0,1\}} in_j.B_{ij} + out_i.B_\epsilon$
- $B_{ij} = out_i.B_j$

Exercise: build the LTS from this set of equations, by using the inference rules.

The above set of recursive equations comprises 7 equations.

By using parametric process definitions, we can reduce this number to 3, where we only have three kinds of buffer: the empty one, the one containing a single bit and the one containing two bits:

- $B_\epsilon = \sum_{i\in\{0,1\}} in_i.B'(out_i)$
- $B'(x) = \sum_{j\in\{0,1\}} in_j.B''(x, out_j) + x.B_\epsilon$
- $B''(x, y) = x.B'(y)$

<u>Exercise</u>: show that the two different sets of process definitions (the one with 7 definitions and the parametric one, with only 3 definitions) generate isomorphic LTSs.

<u>Exercise</u>: Modify the defining equations given above for the queue to model a stack (i.e., a buffer with LIFO policy).