π-Calculus

Reasoning about concurrency and communication (Part 1).

Theoretical Foundations of Concurrency

A formal study of concurrency enables:

- understanding the essential nature of concurrency
- reasoning about the behavior of concurrent systems
- developing tools to aid in producing correct systems

The π-calculus of Robin Milner:

- an algebra (operators, expressions, reaction rules)
- an interpretation for concurrent/communicating/mobile processes

A Quick Overview

Operation	Notation	Meaning
prefix	$\pi.P$	sequencing
action	$x(y)$ $\overline{x}y$	communication
summation	$a.P + b.Q$	choice
	$\Sigma\ \pi_i.P_i$	
recursion	$P = \{\ldots\}.P$	repetition
replication	$!P$	
composition	$P \mid Q$	concurrency
restriction	$(\nu\ x)P$	encapsulation

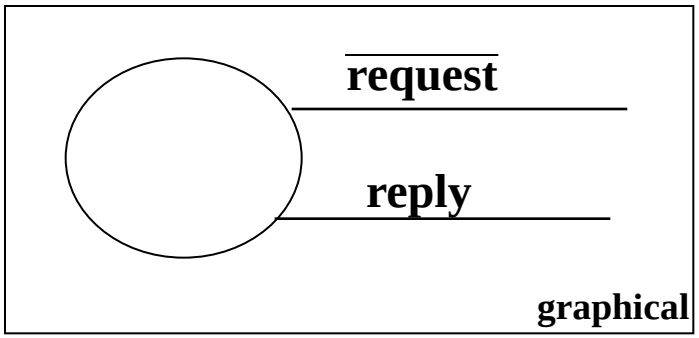
The Structure of a Process

A process is an autonomous entity possessing named ports through which it may communicate with other processes. The name of the process and its ports are introduced as:

ProcessName(*port-list*) = *behavior of the process*

In the description of the process' behavior, port names with overbars are interpreted as "output" ports while names without overbars are often interpreted as "input'" ports.

The process below models a simple client that has one output port, "request" and one input port, "reply".



Client(request, reply) = ….$\overline{\text{request}}$…reply...

algebraic

The special behavior "0" (zero) represents a terminated process (e.g., a process that takes no action).

A Sequential Process

The behavior of a process is expressed by algebraic equations. Suppose that we want to describe a process that behaves like a "client." This behavior can be expressed as:

$$\text{Client}(\text{open}, \text{close}, \text{request}, \text{reply}) = \overline{\text{open}}.\overline{\text{request}}_1.\text{reply}_1.\ \overline{\text{request}}_2.\text{reply}_2.\overline{\text{close}}.0$$

The dot (".") is a prefix operation expressing sequential behavior.

The above equation is read as follows: the Client process issues an "opens" message followed by two request-reply exchanges. It then "closes" the session and terminates.

A Repetitive Sequential Process

A Client that engages in repeated sessions can be expressed using a recursive definition as:

Client(open, close, request, reply) =

$\overline{open}.\overline{request}.reply.\ \overline{request}.reply.\overline{close}.Client(open, close, request, reply)$

The above equation is read as follows: the Client process issues an “open” message followed by two request-reply exchanges. It then “closes” the session and acts like the Client process again.

A Process with Alternative Behavior

A typical sequential server must be able to enforce a protocol of interaction with its clients. The behavior of a typical sequential server process can be modeled as follows.

IdleServer(open, request, reply, close) = open.BusyServer(open, request, reply,close)

BusyServer(open, request, reply, close)
= request.$\overline{\text{reply}}$.BusyServer(open, request, reply, close)
+ close.IdleServer(open, request, reply, close)

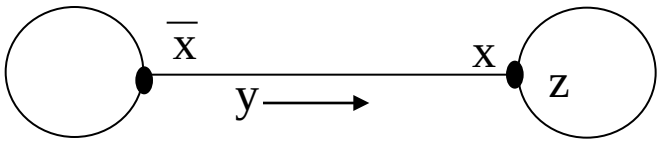
Notes:

- the "+" operator represents choice or alternative action
- the server will only engage in an "open" action interaction at the start
- the server can handle any number of request-reply sequences
- once the server engages in a "close" action, it returns to its original condition
- the server can iteratively handle

Communicating Processes

Processes can be composed, allowing them to communicate through ports with complementary names (i.e., one agent has an output port and the other has an input port with the same name).

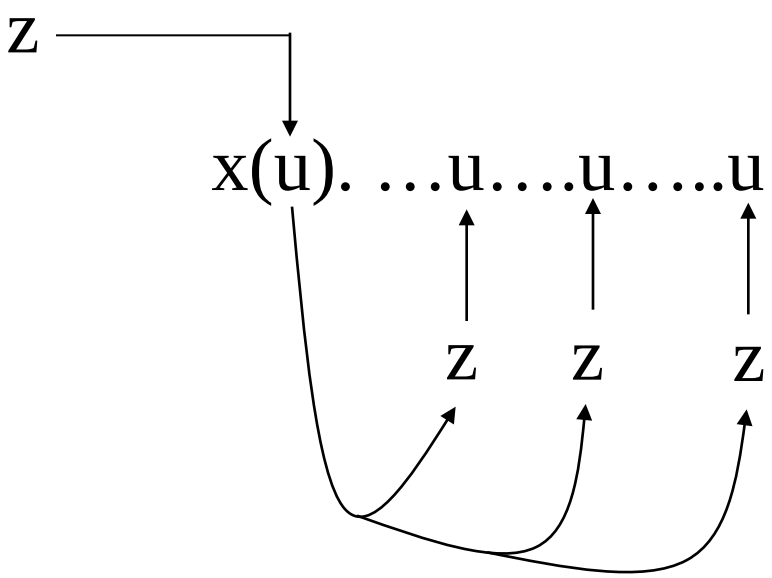
Concurrent communicating agents can synchronize their behaviors through their willingness or unwillingness to communicate. This reflects a rendezvous style of interaction.

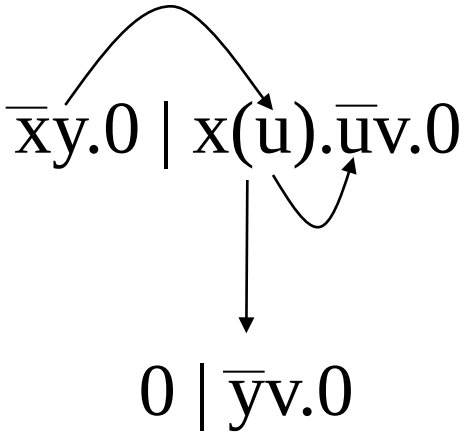


$$\overline{x}y.0 \mid x(z).0$$

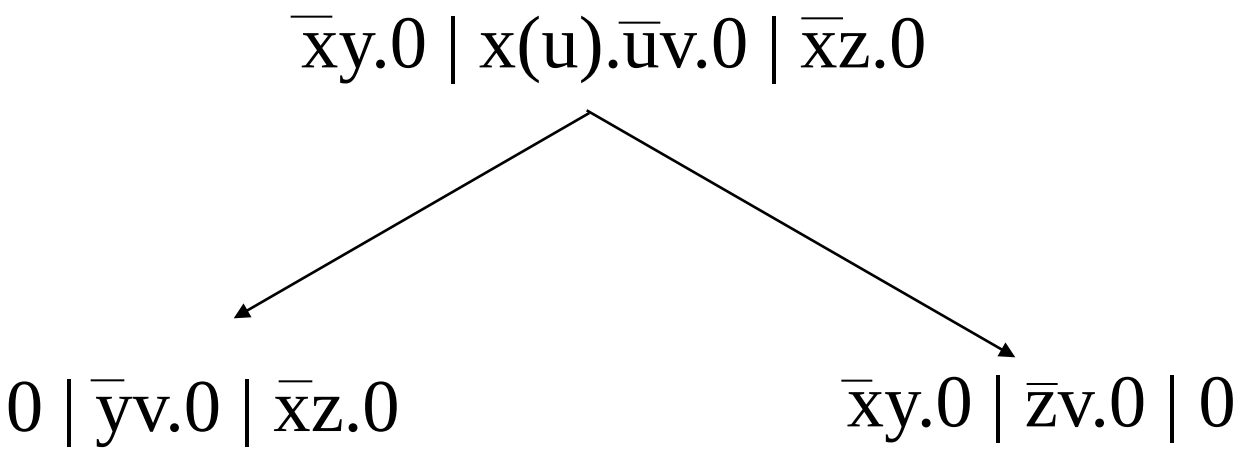
Binding Names on Input

When an input command is a prefix to a process description, the actual name received on an input port replaces in the body of the process description the formal name used in the input command.





Semantics of Concurrent Communication

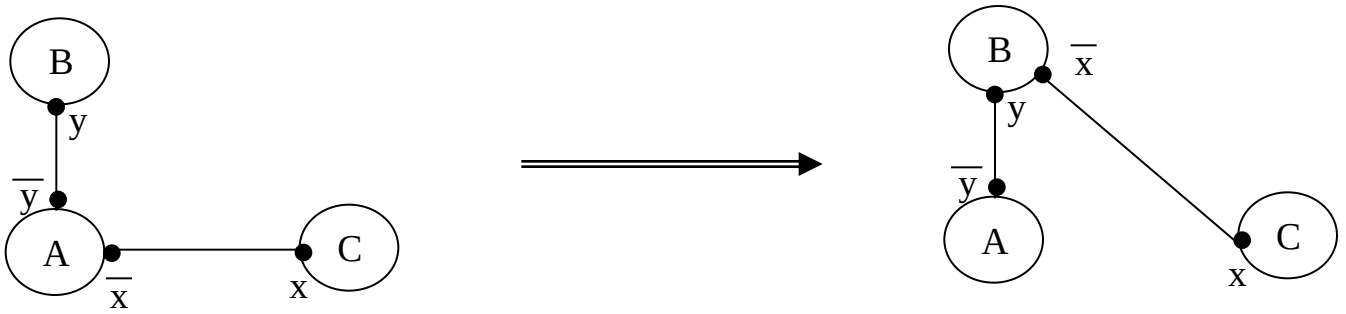


A system can evolve in different ways depending on the interactions among processes.

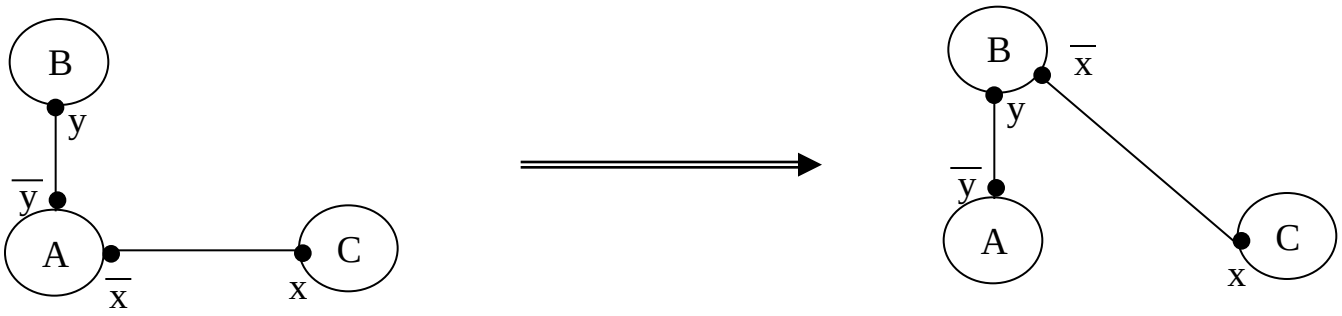
Mobility

Mobility in the π-calculus:

- refers to dynamic change in the communication topology among processes
- is accomplished by a process acquiring and losing ports through which it may communicate with other processes
- is realized by transmitting the name of a port as the value of some communication between two processes allowing the transmitted port to be known to the receiving process



Mobility



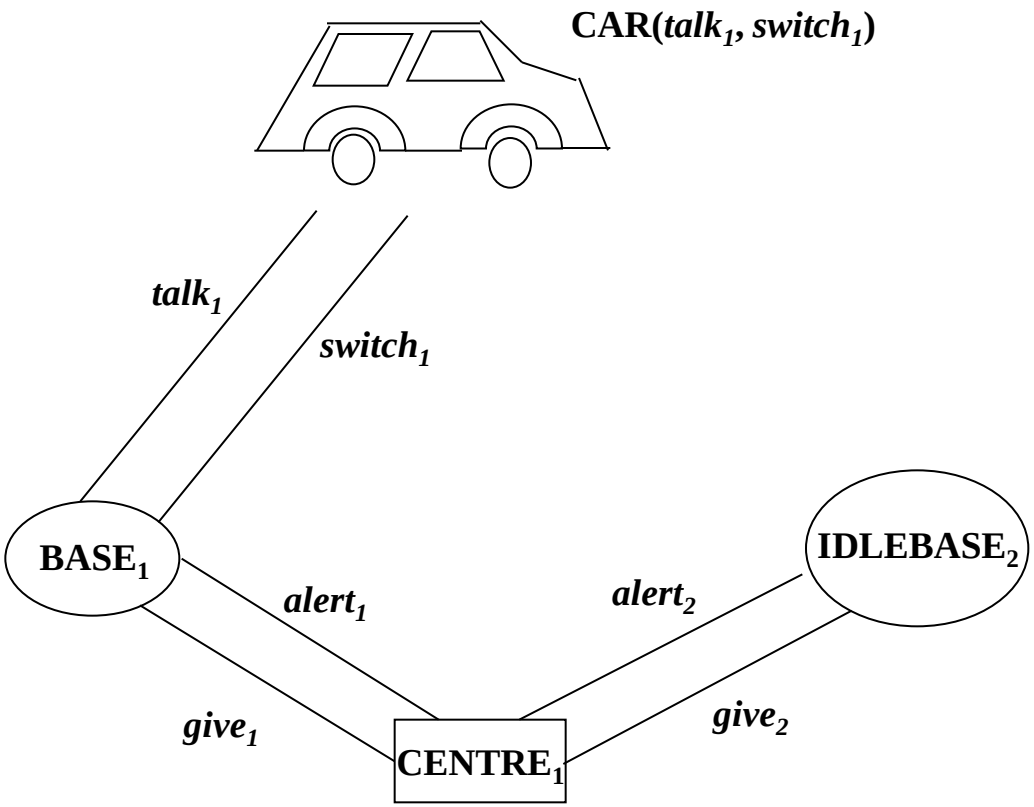
$A(x,y) = \overline{x}.A(x,y) + \overline{y}(x).A'(y)$ $\quad B(y) = y(z).\overline{z}.B'(y,z)$

$A'(y) = \ldots$ $\quad B'(y,z) = \ldots$

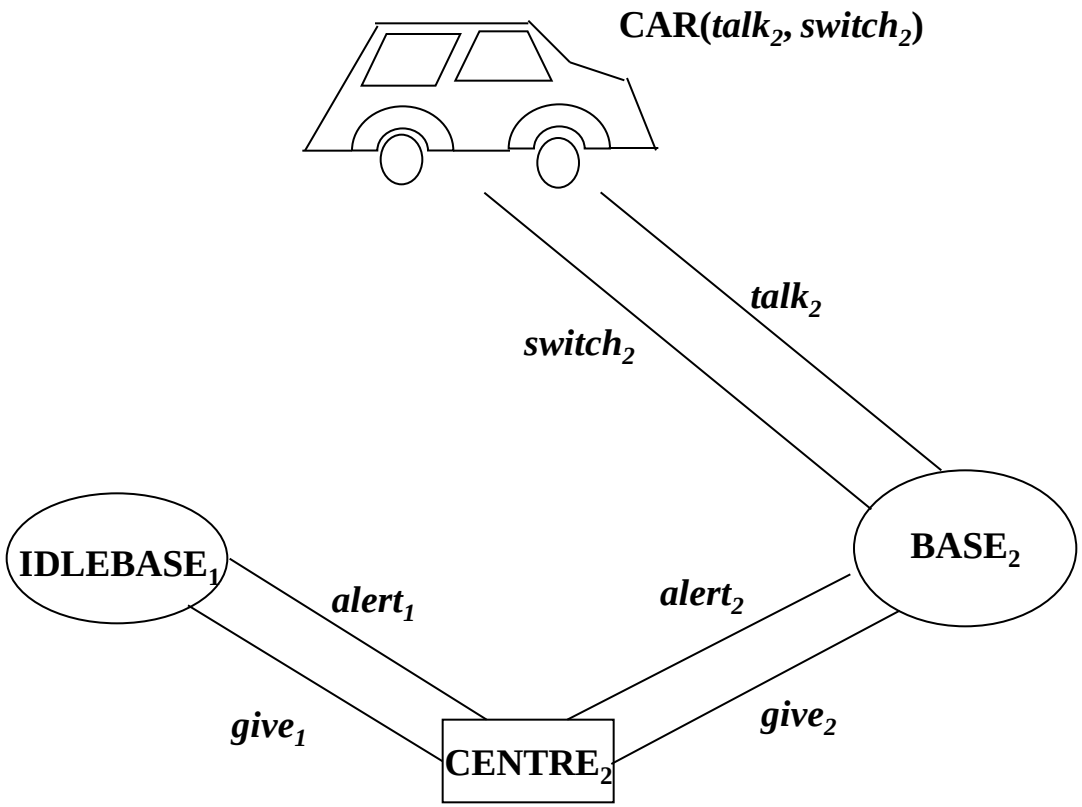
$(A \mid B \mid C) = (x.A(x,y) + \overline{y}(x).A'(y) \mid y(z).\overline{z}.B'(y,z) \mid C)$

$= (A'(y) \mid \overline{x}.B'(y,x) \mid C)$

Mobility


CAR(talk1, switch1)
talk1
switch1
BASE1
IDLEBASE2
alert1
alert2
give1
give2
CENTRE1

Mobility


CAR(talk2, switch2)
talk2
switch2
IDLEBASE1
BASE2
alert1
alert2
give1
give2
CENTRE2

Mobility

The bases can be modeled by:

BASE(talk, switch, give, alert) = talk.BASE(talk, switch, give, alert)
+ give(t, s).$\overline{\text{switch}}$ t s .IDLEBASE(talk, switch, give, alert)

IDLEBASE(talk, switch, give, alert) = alert.BASE(talk, switch, give, alert)

The car's behavior can be described as:

CAR(talk, switch) = $\overline{\text{talk}}$.CAR(talk, switch)
+ switch(talk' switch').CAR(talk', switch')

Mobility

A simple control system, that alternates between the two transmitters, is given by:

$$\text{CENTRE}_1 = \overline{\text{give}_1}\ \text{talk}_2\ \text{switch}_2\ .\overline{\text{alert}_2}.\text{CENTRE}_2$$
$$\text{CENTRE}_2 = \overline{\text{give}_2}\ \text{talk}_1\ \text{switch}_1\ .\overline{\text{alert}_1}.\text{CENTRE1}$$

The mobile transmission system is:

$$\text{SYSTEM}_1 = (\nu\ \text{talk}_1, \text{switch}_1, \text{give}_1, \text{alert}_1, \text{talk}_2, \text{switch}_2, \text{give}_2, \text{alert}_2)$$
$$(\text{Car}(\text{talk}_1, \text{switch}_1) \mid \text{BASE}_1 \mid \text{IDLEBASE}_2 \mid \text{CENTRE}_1)$$

where $\text{BASE}_i = \text{BASE}(\text{talk}_i, \text{switch}_i, \text{give}_i, \text{alert}_i)$ for i = 1,2

$\text{IDLEBASE}_i = \text{IDLEBASE}(\text{talk}_i, \text{switch}_i, \text{give}_i, \text{alert}_i)$ for i = 1,2

Replication

The terminating process definition:

$$\text{Client(open, close, request, reply)} = \overline{\text{open}}.\overline{\text{request}}_1.\text{reply}_1.\ \overline{\text{request}}_2.\text{reply}_2.\overline{\text{close}}.0$$

can be used to describe a longer-running system using replication as follows:

$$\text{SYSTEM} = (\ !\ \text{Client} \mid \text{Server}\ )$$

Which is equivalent to spawning/forking as many copies of the Client process as desired.

Restriction

A name that is private to a process or a group of collaborating processes can be defined by restriction similar to the effect of scoping or encapsulation.

Suppose that two different process are each attempting to communicate two values to a process using port x. The following will not work correctly:

$$\bar{x}a.\ \bar{x}b\ldots. \mid x(f).x(g)\ldots. \mid \bar{x}c.\bar{x}d...$$

Because the middle process could receive the a from the left process and the c from the right process. Encapsulation can be used to create a "private" link along which the values can be passed with interference, as in:

$$(\nu\ w)(\bar{x}w.\bar{w}a.\bar{w}b\ \ldots) \mid x(u).u(f).u(g)\ldots \mid (\nu\ t)(\bar{x}t.\bar{t}c.\bar{t}d\ldots)$$