

Overview of the Lecture

Distributed Systems: the Pi-calculus *Model-based Design of Distributed and Mobile Systems Lecture 8: Spring 2005*

Mikkel Bundgaard

mikkelbu -at- itu.dk

Department of Theoretical Computer Science
IT University of Copenhagen

The structure of the lecture:

- The Pi-calculus
 - Intuition
 - Syntax and reaction rules
 - Connectivity
 - Monadic vs. Polyadic π -calculus
 - Recursion vs. Replication
 - Examples: Buffer and Reference Cell
- Web Services
- Higher-order π -calculus (probably postponed to another lecture)
- Exercises

Mobility

- **Mobility** is an important concept which is not modelled by CCS. It can mean many things:
 1. processes move in the physical space of computing sites;
 2. processes move in the virtual space of linked processes;
 3. links move in the virtual space of linked processes.
- The π -calculus evolved from CCS. It models the changing connectivity of interactive systems (point 3).
- The **ambient calculus** evolved from the pi calculus and models the movement of one location of activity inside another (point 1).
- higher-order π -calculus, CHOCS, Homer, and **Distributed π -calculus** (point 2).

The π -calculus

- The π -calculus
 - An algebra for modeling systems of autonomous agents, known as mobile systems.
 - A **mobile system** is a form of communication network in which the communication between agents can change dynamically
 - Model interactions in **concurrent computational systems** as diverse as cellular telephone networks, the Internet, and object-oriented software programs.

The π -calculus

Describes concurrent systems using *pure* names. We can dynamically create new mobile names. Names in these systems serve (at least) two roles:

- **Information** — who knows what;
- **Communication** — channels for messages.

In the π -calculus these are closely *interlocked*

Compared to CCS we can now dynamically *forge* new links between agents, so connectivity is not static.

To this end we augment the synchronisation of CCS to also *carry information* (names)

Reaction Semantics vs. LTS Semantics

- We normally define the behaviour of a given process calculi using either a *reaction semantics* or a *labelled transition system semantics*
- In the presentation of CCS we use a *LTS semantics* that describes how the processes can interact with the environment
- Whereas, in a *reaction semantics* we describe how the processes can evolve by themselves without interacting with the environment
- There should be a *correspondence* between the two semantics for a given process calculi.
 - Reactions should correspond to τ -transitions of the LTS
 $(\longrightarrow = \xrightarrow{\tau})$
- The reaction semantics are often easier to understand, but the LTS semantics are often easier to reason with

The π -Calculus

The π -calculus is a very large research area and there exists a lot of literature about the π -calculus:

- Several books:
 - Milner “*Communicating and Mobile Systems: the π -calculus*” [Mil99]
 - Sangiorgi and Walker “*The π -calculus: A Theory of Mobile Processes*” [SW01]
- and tons of articles, covering all kinds of areas
 - Distributed and mobile systems (and their implementations)
 - Cryptographic Protocols
 - Web Services
 - Biomolecular Systems
 - Modelling business processes (BPML and BPEL4WS)
 - ...

So it easily becomes *overwhelming*

The Pi-Calculus — Syntax

The *syntax* of the pi-calculus (resembles CCS)

Prefix	α	::=	$\bar{a}(b)$	output
			$a(x)$	input
			τ	internal
Agent	P	::=	0	inactive
			$\alpha.P$	prefix
			$P + P$	sum
			$P \mid P$	parallel
			$(\nu a)P$	restriction
			$A(a_1, \dots, a_n)$	identifier
Def			$A(x_1, \dots, x_n) = P$	declaration

The Pi-Calculus — Reduction Semantics

Reaction Rules

$$\text{TAU} \frac{}{\tau.P + P' \longrightarrow P}$$

$$\text{REACT} \frac{}{(\lambda(y).P + P') \mid (\bar{x}(z).Q + Q') \longrightarrow P\{z/y\} \mid Q}$$

$$\text{PAR} \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \quad \text{RES} \frac{P \longrightarrow P'}{(\nu x)P \longrightarrow (\nu x)P'}$$

$$\text{STRUCT} \frac{P \longrightarrow P'}{Q \longrightarrow Q'} \quad (P \equiv Q \text{ AND } P' \equiv Q')$$

- $P\{z/y\}$ denotes the process P where we have substituted all free occurrences of y with z (using capture-free substitution)

α -conversion

- We can rename bound names without changing the process (called α -conversion)

$$a(x).(\bar{x}(b) \mid x(c)) \quad a(y).(\bar{y}(b) \mid y(c))$$

- As long as we do not **capture** any free names. So the following is **not** possible

$$a(x).(\bar{x}(b) \mid x(c)) \quad a(b).(\bar{b}(b) \mid b(c))$$

- Since the free name b has become bound
- We use α -conversion to satisfy side-conditions in reaction rules and structural congruence and in capture free substitution
- In capture free substitution $P\{z/y\}$ we first α -convert any bound names that could bind z

$$(y(x).0 \mid a(y).\bar{y}(d) \mid (\nu z)\bar{y}(z).0)\{z/y\} = z(x).0 \mid a(y).\bar{y}(d) \mid (\nu z')\bar{z}(z')$$

and not $= z(x).0 \mid a(y).\bar{z}(d) \mid (\nu z)\bar{z}(z)$

Binding

- Names in processes can be **bound** or **free** (and sometimes both)
- The input $a(x).P$ and the restriction $(\nu x)P$ both bind all free occurrences of the name x in P . Hence, the free occurrences of x in P just refer to the binder

$$a(x).(\bar{x}(b) \mid x(c))$$

- But it is only the free occurrences in P that refer to the binder in $a(x).P \mid (\nu x)P$

$$a(x).(\bar{x}(b).P \mid (\nu x)\bar{x}(c).Q)$$

- Since the last x is not free in the expression $(\bar{x}(b).P \mid (\nu x)\bar{x}(c).Q)$, since it is bound by (νx)
- Note that a name can occur both be free and bound in the same expression

$$a(x).P \mid x(y).Q$$

Structural Congruence

We write $P \equiv Q$ for the smallest congruence (equivalence relation that is closed under all contexts) that satisfy the following rules

- α -conversion; change of bound names
- $P \mid Q \equiv Q \mid P, \quad P \mid 0 \equiv P, \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R),$ (same laws for sum)
- $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q,$ if $x \notin fn(P)$ (same law for sum)
- $(\nu x)0 \equiv 0$
- $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
- $A(\bar{y}) \equiv P\{\bar{y}/\bar{x}\},$ if $A(\bar{x}) = P$
- **Examples:**
- $a(x).P \equiv a(y).P\{y/x\}, \quad \bar{a}(n).(P_1 + P_2) \equiv \bar{a}(n).(P_2 + P_1),$
 $P_1 \mid (P_2 + (\nu n)P_3) \equiv P_1 \mid (\nu n)(P_2 + P_3) \equiv (\nu n)(P_1 \mid (P_2 + P_3)),$ if
 $n \notin fn(P_1, P_2), \quad (\nu x)(P_1 \mid P_2) \not\equiv (\nu x)P_1 \mid (\nu x)P_2$

Examples — \equiv

Show that the following agents are structural congruent (ie. that $P \equiv Q$).

$$P = (\nu z)((\bar{x}\langle y \rangle + z(w).\bar{w}\langle y \rangle) \mid x(u).\bar{u}\langle v \rangle \mid \bar{x}\langle z \rangle)$$

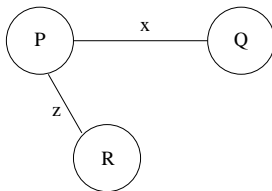
$$Q = x(u).\bar{u}\langle v \rangle \mid (\nu z)((\bar{x}\langle y \rangle + z(w).\bar{w}\langle y \rangle) \mid \bar{x}\langle z \rangle)$$

- Show that if x is not free in Q then $(\nu x)Q \equiv Q$

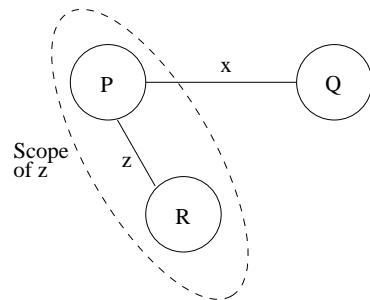
$$Q \equiv Q \mid \mathbf{0} \equiv Q \mid (\nu x)\mathbf{0} \equiv (\nu x)(Q \mid \mathbf{0}) \equiv (\nu x)Q \mid \mathbf{0} \equiv (\nu x)Q$$

Dynamic Connectivity

Consider the system $P \mid R \mid Q$



Now consider the following system $(\nu z)(P \mid R) \mid Q$



Examples — Reactions

So complementary actions can — if they both are unguarded and not in the same summation — constitute a *redex*, which can be fired to constitute a *reaction*

- $\bar{a}\langle n \rangle.P \mid a(x).x(b).Q \longrightarrow P \mid n(b).Q$
- $(\bar{a}\langle n \rangle.0 + \bar{b}\langle m \rangle.0) \mid a(x).Q \longrightarrow Q\{n/x\}$

$$P = (\nu z)((\bar{x}\langle y \rangle + z(w).\bar{w}\langle y \rangle) \mid x(u).\bar{u}\langle v \rangle \mid \bar{x}\langle z \rangle)$$

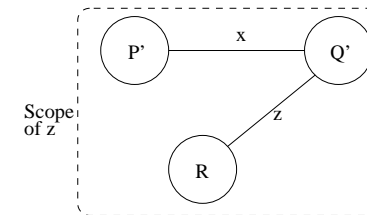
- P has several possible reactions. Which ?
- What are the resulting agents ?

Modelling a call-back function

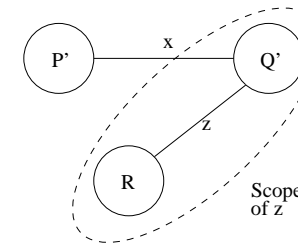
$$Q = (\nu x)(\bar{y}\langle x \rangle.x(b).P) \mid y(z).calculate_a.\bar{z}\langle a \rangle.Q$$

Scope Extrusion

Imagine that P sends the name z along x to Q



And if z is not free in P' .



Questions

- When Q received the local name z we tacitly assumed that z is not free in Q . But suppose it is e.g. $Q = x(y).\bar{z}(y)$.
- Which structural congruence rules must first be applied to $(\nu z)(P \mid R) \mid Q$ in order to make it possible for P and Q to synchronise?
- Scope extrusion** and **local names** are the main concepts behind the expressive power of the π -calculus

Monadic vs. Polyadic π -calculus

- We can express the main reaction rule (in the polyadic version) as

$$\text{REACT} \frac{}{(x(\vec{y}).P + P') \mid (\bar{x}(\vec{z}).Q + Q') \longrightarrow P\{\vec{z}/\vec{y}\} \mid Q}$$

where the vectors \vec{y} and \vec{z} must have the same length.

- If we only consider vectors of length 0 we basically have **CCS**
- And with vectors of length 1 we have π -calculus
- A **sorting discipline** (a kind of simple type system), can be applied to ensure that the requirement above is always satisfied.

Monadic vs. Polyadic π -calculus

- Remember **Mono** = 1 and **Poly** = many
- The π -calculus can be presented in the **monadic** or **polyadic** version (e.g. do we transmit single names or a vector of names)
- It is clear that we can encode the monadic version in the polyadic version. Why?

- The encoding in the other direction can be done using a private channel

$$\begin{aligned} [x(y_1 \cdots y_n).P] &= x(w).w(y_1).\cdots.w(y_n).P \\ [\bar{x}(z_1 \cdots z_n).Q] &= (\nu w)(\bar{x}(w).\bar{w}(z_1).\cdots.\bar{w}(z_n).Q) \end{aligned}$$

- We need the private channel to ensure that processes in the surrounding environment cannot **interfere**

Recursion vs. Replication

- Infinite behaviour can be expressed with either **replication** or **recursive definitions**

Replication

- $!P$ should be thought of as an infinite number of P s in parallel ($P \mid P \mid P \mid \dots$).
- Realised through the Structural congruence rule $!P \equiv !P \mid P$

Recursive Definitions

- $A(\vec{x}) = Q_A$, where Q_A is an process expression which may contain "calls" of A .
- We can express the same using either replication or recursive definitions (they have the same expressive power)

Recursion vs. Replication - pros and cons

- They can express the same, but they each have their own advantages and disadvantages

Replication

- Replication is often preferred in the **development of the theory**, since
 - we only have to add 1 structural congruence rule compared to introducing a new syntactic category and a new reaction rule

Recursive definitions

- Recursive definitions are often preferred in **pragmatic examples**, since
 - it gives a more clear syntax and allows for better division of expressions

- Often both notations are used at the same time



More Examples — Buffer

- Consider an **one-element buffer**

$$\begin{aligned}
 B(l,r) &= l(x).C(x,l,r) \\
 C(x,l,r) &= \bar{r}(x).B(l,r)
 \end{aligned}$$

- We define the binary **linking** operator as

$$P \frown Q = (\nu m)(P\{m/r\} \mid Q\{m/l\}), \text{ where } m \notin fn(P, Q)$$

- So we define a buffer of arbitrary length using linking

$$B^{(n)} = \overbrace{B \frown \dots \frown B}^{n \text{ times}}$$



Encode replication

- We can **encode** replication using recursive definitions

$$P = !(a(x).Q + \bar{a}(y).Q')$$

can be encoded as

$$P' = (a(x).Q + \bar{a}(y).Q') \mid P'$$

- Note that P (P') has an infinite number of immediate reactions
- Sometimes we use a variant of replication, called **guarded replication**, where we require that the process P in $!P$ must be prefixed.
 - Hence, $!a(x).P'$ is a (input) guarded replication, whereas $!(P' \mid Q')$ is not.
 - This version of replication can in some sense express the same, but it is easier to work with, since it takes a reaction to “activate” only instance of the replication



Two-element buffer in MWB

Declaring the buffer elements

$$\begin{aligned}
 \text{MWB} > \text{agent } B(l,r) &= l(x).C(x,l,r) \\
 \text{MWB} > \text{agent } C(x,l,r) &= \bar{r}(x).B(l,r)
 \end{aligned}$$

Linking the two elements and putting an element in parallel

$$\text{MWB} > \text{agent } DB(l,r,v) = (\hat{m}) (B(l,m) \mid B(m,r) \mid 'l(v).0)$$

- Note that we pass some of the state (the value received at l) between the processes.



Two-element buffer in MWB — cont.

Stepping the buffer

```
MWB>step DB<l,r,v>
* Valid responses are:
  a number N >= 0 to select the Nth commitment,
  <CR> to select commitment 0,
  q to quit.
0: |>t. (ˆv) (ˆv<v>.B<l,ˆv> | ˆv(x).C<ˆv,r,x>)
1: |>l. (\ˆv) (ˆv5) (ˆv5<ˆv>.B<l,ˆv5> | ˆv5(x).C<ˆv5,r,x> | 'l<v>.0)
2: |>'l. [v] (ˆv) (l(x).C<l,ˆv,x> | ˆv(x).C<ˆv,r,x>)
Step>0
0: |>t. (ˆv) (l(x).C<l,ˆv,x> | 'r<v>.B<ˆv,r>)
Step>0
0: |[l=r]>t. (ˆv) (ˆv<v>.B<l,ˆv> | ˆv(x).C<ˆv,r,x>)
1: |>l. (\ˆv) (ˆv5) (ˆv5<ˆv>.B<l,ˆv5> | 'r<v>.B<ˆv5,r>)
2: |>'r. [v] (ˆv) (l(x).C<l,ˆv,x> | ˆv(x).C<ˆv,r,x>)
Step>2
Concretion (^)[v]
```

Open distributed systems

- Many successful distributed systems are *loosely coupled* collections of programs and machines
- Linked by agreements on how widely *known names* should refer to certain local resources. For example:
 - well-known IP ports (finger, daytime, http, ftp);
 - library calls (Java API, libc).
 - Other ?
- We can also model these systems using the π -calculus.

More Examples — Reference Cell

- Instead of passing the state as arguments to the declarations we can *maintain* it at a local location l
- We write to it through public channel w , and read from it through r
- Hence, *encapsulating* the state of the cell and *accessing* it through methods

$$Ref(r, w, i) = (\nu l)(\bar{l}i \mid ReadS(l, r) \mid WriteS(l, w))$$

$$ReadS(l, r) = !r(c).l(v).(\bar{c}v \mid \bar{l}v)$$

$$WriteS(l, w) = !w(c, v').l(v).(\bar{c}v' \mid \bar{l}v')$$

Some clients that uses the cell (using local channels to avoid interference)

Write to the cell $(\nu c)\bar{w}\langle c, v \rangle.c().P$

Read from the cell $(\nu d)\bar{r}\langle d \rangle.d(e).Q$

Example π -calculus internet daemon

(Example taken from Ian Stark)

An *internet daemon* (e.g. xinetd) routes incoming service requests to appropriate handlers. Here is a π -calculus model of this.

Client:

$(\nu c)(\overline{server}\langle finger, c \rangle$

← ask server

$| c(x).\overline{print}\langle x \rangle)$

← print response

Server:

$server(service, reply).\overline{service}\langle reply \rangle$

← inet daemon

$| !finger(reply).\overline{reply}\langle users \rangle$

← finger daemon

$| !time(reply).\overline{reply}\langle now \rangle$

← time daemon

π -calculus and Web Services

- Recently, there has been much research in applying process calculi to **web service choreography** or *orchestration*
- Rehof *et al* — **Behave!**, state and check high-level communication protocols for concurrent, distributed applications
- Andrew Gordon and Martín Abadi — **Spi calculus**, Secure Global Computing with XML Web Services. Proving secrecy and authenticity properties by typing
- Andrew Gordon et al. — **Samoa**, Formal Tools for Securing Web Services. In particular, **TulaFale**, which is a security tool for web services
- π -calculus is the mathematical foundation of two of the main (Business) Process Markup Languages: **BPML** and **XLANG** (now **BPEL4WS**)

Copying processes

Since an agent variable can occur several times in an agent, we can now **copy agents**

$$a(X).(X \mid X) \mid \bar{a}(Q).R \longrightarrow Q \mid Q \mid R$$

So we can *encode recursion* and *replication* (note that it takes one reaction to unfold another copy).

- Encoding of replication

$$D = a(X).(X \mid \bar{a}(X)), \text{ where } a \notin fn(P)$$

$$R_P = (\nu a)(D \mid \bar{a}(P \mid D))$$

- Why do we have to send D with P

Higher-order π -calculus

The ideas of Bent Thomsen and Davide Sangiorgi

- We add higher-order input $a(X).Q$ and output $\bar{a}(P).Q$ prefixes, and add agent variables X
- Process-passing instead of name-passing
- New rule for reaction

$$\text{REACTHO} \frac{}{(a(X).P + P') \mid (\bar{a}(Q).R + R') \longrightarrow P\{Q/X\} \mid R}$$

Examples:

- $\bar{a}(R).P \mid a(X).X \longrightarrow P \mid R$
- $\bar{a}(R).P \mid a(X).0 \longrightarrow P$

Higher-order into First-order

- First-order π -calculus is **expressive enough** to encode higher-order π -calculus. The main theme of Sangiorgi's Thesis "*Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*"
- Principle of encoding (use **triggers**)
- Assume fresh name x for each agent variable X

$$[\bar{a}(P).Q] = (\nu p)\bar{a}(p).([\![Q]\!] \mid !p.[\![P]\!]), \text{ where } p \notin fn(P, Q)$$

$$[a(X).P] = a(x).[\![P]\!]$$

$$[X] = \bar{x}$$

Example

In HO π : $\bar{a}(R).P \mid a(X).(X \mid X) \longrightarrow P \mid R \mid R$

In π -calculus: $(\nu p)\bar{a}(p).([\![P]\!] \mid !p.[\![R]\!]) \mid a(x).(\bar{x} \mid \bar{x}) \longrightarrow$

$$(\nu p)([\![P]\!] \mid !p.[\![R]\!] \mid (\bar{p} \mid \bar{p})) \longrightarrow$$

$$(\nu p)([\![P]\!] \mid !p.[\![R]\!] \mid [\![R]\!] \mid \bar{p}) \longrightarrow \sim [\![P]\!] \mid [\![R]\!] \mid [\![R]\!]$$

Exercise — \equiv and \longrightarrow

(Some of the following exercises are taken from Philippa Gardner)
Using the **structural congruence** relation \equiv and the **reaction relation** \longrightarrow show that

$$\bar{x}(y).0 \mid (\nu y)x(z).Q\langle y, z \rangle$$

can react to become

$$(\nu y')Q\langle y', y \rangle$$

Exercise — Draw the Connectivity

Draw the connectivity of the following expression for each change of the configuration (assuming that P' and R' do not share any names)

$$P = \bar{a}(c).c().P'$$

$$Q = a(x).\bar{b}(x).0$$

$$R = b(y).\bar{y}\langle \rangle.R'$$

Exercise — Polyadic into Monadic

Apply the **encoding** of polyadic π -calculus into monadic π -calculus to the following expression

$$x(y_1, y_2).P \mid \bar{x}\langle z_1, z_2 \rangle.Q \mid \bar{x}\langle z'_1, z'_2 \rangle.Q'$$

And infer enough reductions to **convince** yourself that only the 'right' replacements occur (that either both z_1 and z_2 or both z'_1 and z'_2 are transmitted, such that no interference can occur).

If we use the following **simpler encoding**, where we just send one name at a time.

$$\begin{aligned} \llbracket x(y_1 \cdots y_n).P \rrbracket &= x(y_1).\cdots.x(y_n).P \\ \llbracket \bar{x}\langle z_1 \cdots z_n \rangle.Q \rrbracket &= \bar{x}\langle z_1 \rangle.\cdots.\bar{x}\langle z_n \rangle.Q \end{aligned}$$

What can go wrong? Give an example of this

Exercise — Boolean Values

We can encode **boolean values** as:

$$\begin{aligned} True_a &= a(t, f).\bar{t}\langle \rangle \\ False_a &= a(t, f).\bar{f}\langle \rangle \end{aligned}$$

And the **conditional** as:

$$Case_a(P, Q) = (\nu x)(\nu y)\bar{a}\langle x, y \rangle.(x().P + y().Q)$$

Try to evaluate the following two expressions

$$\begin{aligned} True_a \mid Case_a(P, Q) &\longrightarrow???? \\ False_a \mid Case_a(P, Q) &\longrightarrow???? \end{aligned}$$

How would you implement logical **and**, logical **or**, and logical **not**?

Exercise — Encoding of Recursion

Using the replication operator we can **encode recursion** ([Mil99]) using triggers.

Encode $A(\tilde{x}) = Q_A$, where Q_A depends on A and the scope of the definition is process P , as follows:

1. invent a **fresh** (unused) name, say a , to stand for A
2. for any agent R , denote by \widehat{R} the result of replacing every instantiation $A(\tilde{w})$ in R by $\bar{a}\langle\tilde{w}\rangle$
3. replace P , and the accompanying definition of A , by $\widehat{\widehat{P}} = (\nu a)(\widehat{P} \mid !a(\tilde{x}).\widehat{Q}_A)$.

All Done

If you have done all the exercises, come ask me for more ;-)

Exercise — Encoding of Recursion cont.

Consider the previous defined buffer:

$$\begin{aligned} B(l, r) &= l(x).C(x, l, r) \\ C(x, l, r) &= \bar{r}\langle x \rangle.B(l, r) \end{aligned}$$

- Denote the two right-hand sides Q_B and Q_C . Write down $\widehat{Q_B}$ and $\widehat{Q_C}$.

Assume that the scope of the definition is

$$P = \bar{l}\langle x_1 \rangle.\bar{l}\langle x_2 \rangle \mid (\nu m)(B(l, m) \mid B(m, r)) \mid r(y_1).r(y_2)$$

Write down $\widehat{\widehat{P}}$ and check that it behaves as expected (but with more reactions).

References

References

- [Mil99] Robin Milner. *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, 1999.
- [SW01] Davide Sangiorgi and David Walker. *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.