Seminaire interne AeLoS
06/01/2011

Semantique de Kmelia (interaction multi-partie) exprimée en π-calcul

Christian ATTIOGBE

N. Yoshida
Lecture I
Intro of Pi-calculus

About the π-calculus

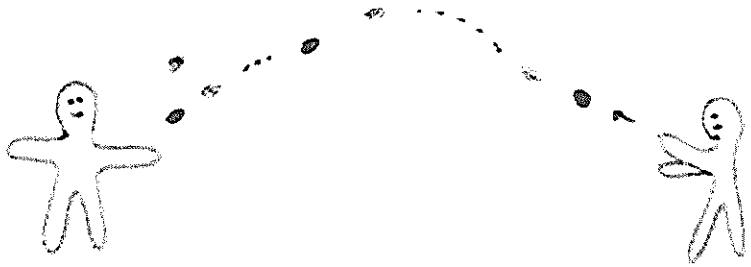
- There is no "canonical" π-calculus. For each application domain, there are alternative notations and lots of specialised variants:
  - we fix one notation and stick to it (once you know one notation, it's easy to understand the others)
  - we start with the simplest variant (the asynchronous π-calculus) and explore various extensions
- An interaction happens by *message passing* rather than *synchronisation*.
- In the asynchronous π-calculus, the communication is *asynchronous* rather than *synchronous*.
- The π-calculus evolved from (value-passing) CCS, but it is more expressive:
  - channel mobility: send and receive *channel names* as messages
  - restriction is interpreted as new (private, secret) channel generation

extrait d'un cours de Yoshida (Models of Concurrent Computation)
http://www.doc.ic.ac.uk/~yoshida/

Synchrony and Asynchrony

- Synchrony

- Asynchrony



cf. E-mail.
News.

6

The asynchronous π-calculus

The asynchronous π-calculus is a subset of the π-calculus presented independently by Honda and Tokoro (1991), and by Boudol (1992).

Communication is asynchronous: the output process $\overline{a}\langle b\rangle$ represents a message which is in the communication layer waiting to be picked up by a receiver (it does not have a continuation P like the synchronous process $\overline{a}\langle b\rangle.P$). Several messages can be in the communication layer at the same time, and their order is not preserved. Asynchronous communication is common in distributed systems, and can be used to simulate synchronous communication (handshake) when needed.

The asynchronous π-calculus is easier and more efficient to implement than the full π-calculus, also thanks to the absence of summation (the "+" of CCS). It is widely used as the basis for building more complicated calculi, by adding primitives for distributed or object-oriented programming.

Syntax

$u, v ::=$		identifiers
	$a, b, c, ..$	name
	$x, y, z, ..$	variable
$P, Q ::=$		processes
	$\mathbf{0}$	nil process
	$P \mid Q$	parallel composition of P and Q
	$(\nu\, a)P$	generation of a with scope P (also called *restriction*)
	$!P$	replication of P, i.e. infinite parallel composition $P \mid P \mid P \mid \ldots$
	$\overline{u}\langle v \rangle$	output of v on channel u
	$u(x).P$	input of *distinct* variables x on u, with continuation P

Notation:
$$\begin{cases} \widetilde{u} \stackrel{\text{df}}{=} u_1, \ldots, u_n & \text{when we don't care about each } u_i \\ (\nu\, a_1, \ldots, a_n)P \stackrel{\text{df}}{=} (\nu\, a_1) \ldots (\nu\, a_n)P \end{cases}$$

Later on, we will consider other CCS-like operators such as summation, output continuation, recursive definitions.

Infinite behaviour and Sorts

We use the macros for process definition $\mathbf{A}(\widetilde{x})$ and usage $\mathbf{A}\langle\widetilde{v}\rangle$ *only informally*. To represent infinite behaviour, we use the *replication* operator $!P$, stating that there are as many copies of P as needed, all running in parallel. For example:

$$a(x).P \mid a(x).Q \mid !\overline{a}\langle b\rangle \longrightarrow P\{^{b}/_{x}\} \mid a(x).Q \mid !\overline{a}\langle b\rangle \longrightarrow P\{^{b}/_{x}\} \mid Q\{^{b}/_{x}\} \mid !\overline{a}\langle b\rangle$$

Replication is very simple, yet combined with channel generation it can encode recursive definitions like the ones used in CCS (later on we will see how).

We separate channel names, which are like constants in a programming language, from variables, which are used to instantiate messages received in input. Consequently, we have two sorts:

$a, b, c \in \mathcal{N}$	Channel Names
$x, y, z \in \mathcal{V}$	Variables

This distinction is not mandatory to build the theory, but is convenient when the calculus is used to model actual systems.

Channel mobility example

Internet connection: a **Client** and a **Server** talk on dedicated communication ports, set up using the channel a:

$$\mathbf{Client}(a, c) \stackrel{\text{df}}{=} (\overline{a}\langle c\rangle \mid c(x).\mathbf{Client}_1\langle c, x\rangle)$$

$$\mathbf{Server}(a, s) \stackrel{\text{df}}{=} a(y).(\overline{y}\langle s\rangle \mid \mathbf{Server}_1\langle y, s\rangle)$$

Difference with CCS: the variable y is used as the name of a channel.

$$\begin{aligned} & \mathbf{Client}\langle a, c\rangle \mid \mathbf{Server}\langle a, s\rangle \\ \longrightarrow \quad & c(x).\mathbf{Client}_1\langle c, x\rangle \mid \overline{c}\langle s\rangle \mid \mathbf{Server}_1\langle c, s\rangle \\ \longrightarrow \quad & \mathbf{Client}_1\langle c, s\rangle \mid \mathbf{Server}_1\langle c, s\rangle \end{aligned}$$

After the two communication steps, client and server know each other's port.

Channel generation example

We can make the Internet connection example more flexible. To avoid external interferences, the client and server exchange newly generated port names:

$$\mathbf{Client}(a) \stackrel{\text{df}}{=} (\nu\, c)(\overline{a}\langle c\rangle \mid c(x).\mathbf{Client}_1\langle c, x\rangle)$$

$$\mathbf{Server}(a) \stackrel{\text{df}}{=} a(y).(\nu\, s)(\overline{y}\langle s\rangle \mid \mathbf{Server}_1\langle y, s\rangle)$$

With CCS notation of restriction, **Client**(a) is written as:

$$(\overline{a}\langle c\rangle \mid c(x).\mathbf{Client}_1\langle c, x\rangle) \setminus \{c\}$$

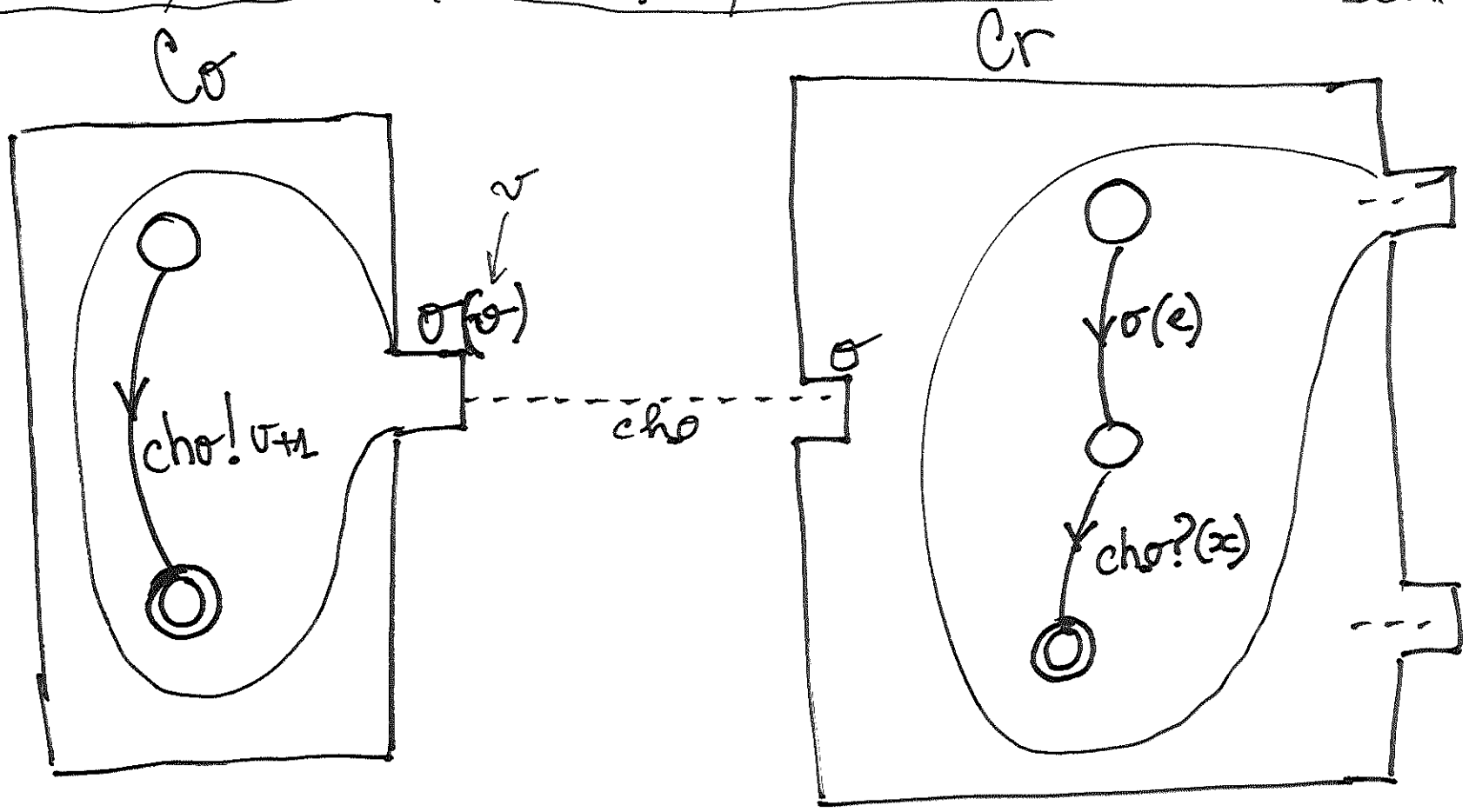
No other process knows about c and s because they are inside the name restriction: you can imagine that they will be created at run-time by the ν operator (pronounced "new").

$$\begin{aligned} & \mathbf{Client}\langle a\rangle \mid \mathbf{Server}\langle a\rangle \\ \longrightarrow\ & (\nu\, c)(c(x).\mathbf{Client}_1\langle c, x\rangle \mid (\nu\, s)(\overline{c}\langle s\rangle \mid \mathbf{Server}_1\langle c, s\rangle)) \\ \longrightarrow\ & (\nu\, c, s)(\mathbf{Client}_1\langle c, s\rangle \mid \mathbf{Server}_1\langle c, s\rangle) \end{aligned}$$

Note that $\mathbf{Client}_1$ and $\mathbf{Server}_1$ are now within the scope of the ν operator. This phenomenon, called *scope extrusion*, is a distinguishing feature of the π-calculus.

Sémantique de Kmel (interaction) exprimée avec π-calcul

Sem. Kélos 06/11/19
CA



Kmelia

avec quelques entorses à la syntaxe

Intro rapide à π-calculus (Milner)

Syntaxe

$u, v ::=$ identifiers
- $a, b, c, \ldots$ noms de canaux
- $x, y, z, \ldots$ noms de variables

$P, Q ::=$ processus
- 0 processus nil (Terminaison)
- $P \mid Q$ composition parallèle de P et Q
- $(\nu a) P$ generation d'un nom de canal a dans le processus P (restriction)
- $!P$ replication du processus P autant de fois que nécessaire = composition parallèle infinie
- $\bar{u}\langle v \rangle$ sortie/émission de v sur le canal u
- $u(x).P$ entrée/attente de x sur le canal u, puis on fait P

Encodage en π-calcul

Sem. Atlas 06/01/14
CA

Côté de c_0 : le service o

$o(ch_0, v)$ ... : attente d'appel avec un canal ch_0 et une valeur v.

et donc

$o(ch_0, v).\overline{ch_0}\langle v+1\rangle.0$: sur l'appel, on poursuit avec $v+1$ sur le canal reçu.

Pour les répétitions possible

réplication $!?$: autant de $!$ que nécessaire

$!o(ch_0, v).\overline{ch_0}\langle v+1\rangle.0$

Intéressant si pas de partage

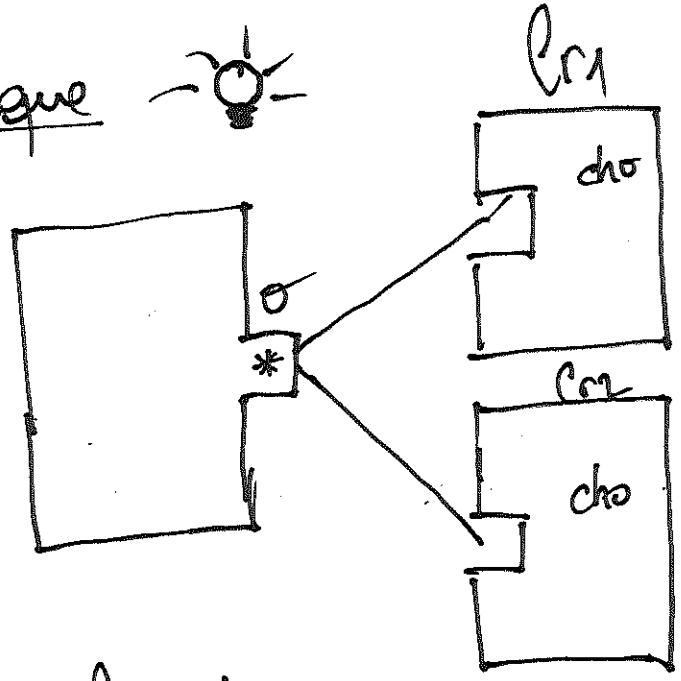
Encodage en π-calcul

Sém. Arch. 06/04/11
CA

Côté de Cr : le service appelant

$$\underbrace{\nu(cho)}_{\text{nouveau cho}} . \underbrace{\bar{o}\langle cho, e\rangle}_{\text{appel de o en passant cho, et e}} . \underbrace{cho(x)}_{\text{attente}} . 0$$

Remarque



$\nu(cho)$: Dans Cr1 nouveau canal cho pour communiquer avec o

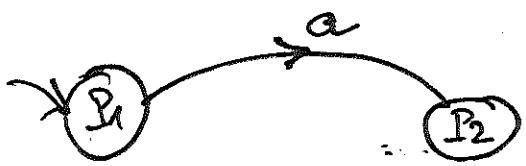
$\nu(cho)$: nouveau canal cho pour communiquer avec o

Chaque Cri est sur son canal ✓ intéressant si partagé

Encodage ds services

Sem. Atlas 06/01/14 CA

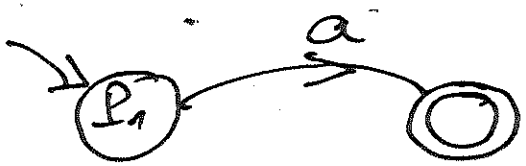
S :



$s(ch_s).a.[\![P_2]\!](ch_s)$

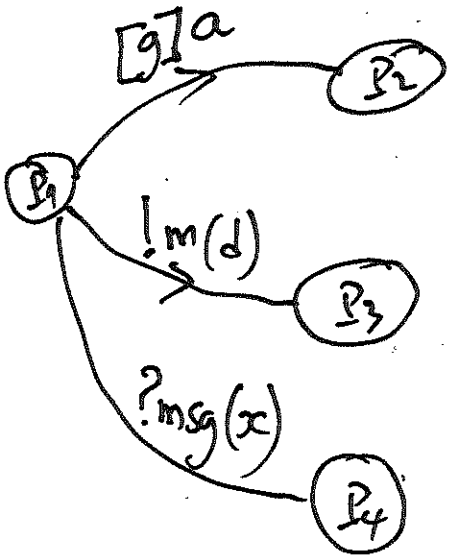
suite

Appel avec au moins le canal ch_s



$s(ch_0).a.0$

S :



$s(ch_s).P_1(ch_s)$

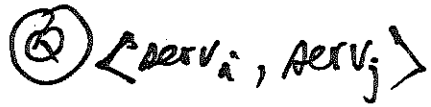
$P_1(ch_s) = [g]a.P_2(ch_s)$

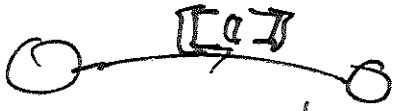
$+ \overline{ch_s}\langle m, d\rangle.P_3(ch_s)$

$+ ch_s(msg, x).P_4(ch_s, x)$

↑ à régler : exactement msg reçu

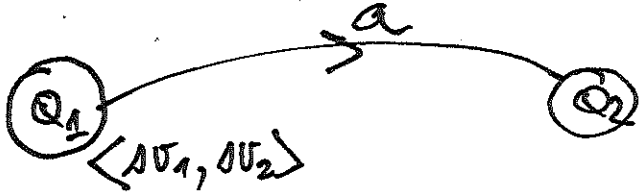
eLTS





transition obligatoire

Jean Achat 06/11/14
LIA



$$Q_1(ch_s) = a \,.\, Q_2(ch_s)$$
$$+\ ch_s(sv_1) \,.\, \overline{sv_1}\langle ds \rangle \,.\, Q_1(ch_s)$$
$$+\ ch_s(sv_2) \,.\, \overline{sv_2}\langle \text{---} \rangle \,.\, Q_1(ch_s)$$

} réglage de paramètres de sv_1 avec ds ou vide --- ou $ch_s \in ++$

Note

Si sv_1 et sv_2 avaient des paramètres, on les mettrait avec le canal ch_s : $\overline{sv_2}\langle ch_s, \text{---} \rangle$

work in progress