Roberto Bruni\textsuperscript{1} \quad Rocco De Nicola\textsuperscript{2}

\textsuperscript{1}Dipartimento di Informatica  
Università di Pisa

\textsuperscript{2}Dipartimento di Sistemi ed Informatica  
Università di Firenze

Process Algebras and Concurrent Systems  
May–June 2006

A solution to the 50 prisoners problem
Puzzle Time: 50 Prisoners Problem

Solve the problem and write down the solution as a process

50 prisoners kept in separate cells got a chance to be released: From time to time one of them will be carried in a special room (in no particular order, possibly multiple times consecutively, but with a fair schedule to avoid infinite wait) and then back to the cell.

The room is completely empty except for a switch that can turn the light either on or off (the light is not visible from outside and cannot be broken).

At any time, if any of them says that all the prisoners have already entered the room at least once and this is true, then all prisoners will be released (but if it is false, then the chance ends and they will never be released).

Before the challenge starts, the prisoners have the possibility to discuss together some “protocol” to follow.

Can you find a winning strategy for the prisoners?

Note that the initial state of the light in the room is not known.

First Attempt

The Light (initially ON)

\[
\text{Light} \triangleq \text{turnOff}.\text{turnOn}.\text{Light}
\]

One Counting-Down Prisoner

\[
\begin{align*}
C_{i+1} & \triangleq \text{turnOff}.C_i + \text{turnOn}.\text{turnOff}.C_{i+1} \\
C_0 & \triangleq \text{freeAll}
\end{align*}
\]

49 Generic Prisoners

\[
\begin{align*}
\text{P} & \triangleq \text{turnOn}.\text{IP} + \text{turnOff}.\text{turnOn}.\text{P} \\
\text{IP} & \triangleq \tau.\text{IP} \quad \text{(Idle Prisoner)}
\end{align*}
\]

Prison

\[
(\text{Light} \mid C_{50} \mid \text{P} \mid \ldots \mid \text{P}) \setminus \{\text{turnOn, turnOff}\}
\]
The light must be accessed in mutual exclusion

**The Room**

\[ \text{Room} \triangleq \text{roomIn} . \text{roomOut} . \text{Room} \]

**One Counting-Down Prisoner**

\[ \begin{align*}
C_{i+1} & \triangleq \text{roomIn} . (\text{turnOff} . \text{roomOut} . C_i + \text{turnOn} . \text{turnOff} . \text{roomOut} . C_{i+1}) \\
C_0 & \triangleq \text{freeAll}
\end{align*} \]

**49 Generic Prisoners**

\[ \begin{align*}
P & \triangleq \text{roomIn} . (\text{turnOn} . \text{roomOut} . \text{IP} + \text{turnOff} . \text{turnOn} . \text{roomOut} . P) \\
\text{IP} & \triangleq \text{roomIn} . \tau . \text{roomOut} . \text{IP}
\end{align*} \]

Initial state is not known

**The Light**

\[ \begin{align*}
\text{Light} & \triangleq \tau . \text{LightOn} + \tau . \text{LightOff} \\
\text{LightOn} & \triangleq \text{turnOff} . \text{LightOff} \\
\text{LightOff} & \triangleq \text{turnOn} . \text{LightOn}
\end{align*} \]

If the light is initially ON and the counting prisoner enters the room first then it must count 50 switching, otherwise only 49... but he cannot know!

**Prison**

\[ \langle \text{Light} | C_{50 \text{ or } 49} \mid P \mid \ldots \mid P \rangle \backslash \{ \text{turnOn}, \text{turnOff}, \text{roomIn}, \text{roomOut} \} \]
Solution

What about counting twice?
50 + 49 = 99
49 + 49 = 98

If he can count 98 switchings then all other prisoners must have entered the room at least once (well... most of them twice). If not:
49 + 48 = 97
48 + 48 = 96

**Generic Prisoners & Re-playing Prisoners & Idle Prisoners**

<table>
<thead>
<tr>
<th>P</th>
<th>=</th>
<th>roomIn.*(turnOn.roomOut.RP + turnOff.turnOn.roomOut.P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RP</td>
<td>=</td>
<td>roomIn.*(turnOn.roomOut.IP + turnOff.turnOn.roomOut.P)</td>
</tr>
<tr>
<td>IP</td>
<td>=</td>
<td>roomIn.μ.roomOut.IP</td>
</tr>
</tbody>
</table>

**Prison**

(Light | C₉₈ | P | ... | P) \{turnOn, turnOff, roomIn, roomOut\)

**Pi Calculus**

(Thanks to D. Gorla for providing part of the material)
A Bit Of History - I

- 1980: “A calculus of communicating systems” (by Robin Milner). The source of inspiration for a large body of the theory of concurrency developed in the last three decades.
- 1986: “A calculus of communicating systems with label passing” (by Mogens Nielses, Uffe Engberg). ECCS showed how label-passing could be added to CCS.
- 1989: “A calculus of mobile processes” (by Robin Milner, Joachim Parrow, David Walker). First paper on $\pi$-calculus, accounting for the work done intermittently over the previous two years.
- 1991/92: “An object calculus for asynchronous communication” (by Kohei Honda, Mario Tokoro) and “Asynchrony and the pi-calculus” (by Gerard Boudol). Independently consider the asynchronous fragment of $\pi$-calculus.

A Bit Of History - II

- 1992: “Functions as processes” (by Robin Milner) and “Barbed bisimulation” (by Robin Milner, Davide Sangiorgi). Introduce the use of structural congruence, reduction semantics and barbed congruence in place of LTS semantics.
- 1992: “Expressing mobility in process algebras: First-order and higher-order paradigms” (by Davide Sangiorgi). Introduces higher-order $\pi$-calculus ($\mathrm{HO}\pi$).
- 1996: “A theory of bisimulation for the $\pi$-calculus” (by Davide Sangiorgi). Introduces open semantics.
A Bit Of History - III

- 1999: “Communicating and mobile systems: the pi-calculus” (by Robin Milner). First book on \( \pi \)-calculus (emphasis on examples and modeling of systems).
- 2001: “The \( \pi \)-calculus: A theory of mobile processes” (by Davide Sangiorgi, David Walker). Covers the basic theory of \( \pi \)-calculus in great depth.

Our Focus

- From CCS to \( \pi \)-calculus
- LTS Semantics vs Reduction Semantics
- Early Semantics vs Barbed Semantics
- Polyadic vs Monadic Communication
- Asynchronous vs Synchronous
- Higher-Order vs First-Order
- Late and Open Semantics
**From CCS to $\pi$-calculus - 1**

Consider a scenario of somebody willing to buy a pizza.

In CCS, we can model this situation by composing in parallel the client $C$, and the “pizzaiolo” $P$.

$$C \triangleq \text{askPizza}.\text{pay}.\text{pizza} \quad P \triangleq \text{askPizza}.\text{pay}.\text{pizza}$$

The client $C$ asks for a pizza, pays and takes it away. The “pizzaiolo” $P$ receives the request for the pizza, gets the money and delivers the pizza.

**From CCS to $\pi$-calculus - 2**

If we use values, i.e. CCS with value passing, we can add further details to our system.

$$C \triangleq \text{askPizza}\langle\text{margherita}\rangle.\text{pay}(5 \text{ Euro}).\text{pizza}$$

$$P \triangleq \text{askPizza}(x).\text{pay}(y).\text{if } y = \text{price}(x) \text{ then } \text{pizza} \text{ else } \text{if } y > \text{price}(x) \text{ then } \text{pizza}.\text{output}(y - \text{price}(x)) \text{ else } \text{askMoney}$$

The client asks for a Margherita, pays the due amount and eats the pizza. The “pizzaiolo” receives the request for the pizza, gets the money then checks the received amount and gives back the requested pizza and possibly the change.
From CCS to $\pi$-calculus - 3

With $\pi$-calculus we can do more: home delivery of pizza!

$$C \triangleq \overline{\text{askPizza}(\text{myHome})}.\overline{\text{pay}.\text{myHome}(x).\overline{\text{eat}(x)}}$$

$$P \triangleq \text{askPizza}(y).\text{pay}.(\nu \text{pizza})\overline{\text{y(pizza)}}.P$$

The client can communicate the address where he wants the pizza be delivered.

$$\overline{\text{askPizza}(\text{myHome})}.\overline{\text{pay}.\text{myHome}(x).\overline{\text{eat}(x)}} \mid$$

$$\overline{\text{askPizza}(y).\text{pay}.(\nu \text{pizza})\overline{\text{y(pizza)}}.P}$$

$$\overset{\rightarrow}{\overline{\text{pay}.\text{myHome}(x).\overline{\text{eat}(x)}|\text{pay}.(\nu \text{pizza})\overline{\text{myHome}(pizza)}}.P}$$

$$\overset{\rightarrow}{\overline{\text{myHome}(x).\overline{\text{eat}(x)}|(\nu \text{pizza})\overline{\text{myHome}(pizza)}}.P}$$

$$\overset{\rightarrow}{(\nu \text{pizza})(\overline{\text{eat}(pizza)}|P)}$$

A Mobile Calculus

$\pi$-calculus is both a general model of (interaction-based) computation and a theoretical setting for studying mobile systems.

It is a conceptual framework for understanding mobility and it provides mathematical tools for reasoning about the behaviour of mobile systems.

But what is mobility?

- what are the entities that move around?
- in what space?
- moved by whom?

Two approaches

- processes that move in an abstract space of linked processes (e.g., mobile code, Jini technology, procedures passed as arguments)
- links that move in an abstract space of linked processes (the $\pi$-calculus view!)
About Links

The π-calculus has two basic entities

- processes (interacting through links)
- names of links

What is a link?

π-calculus is not prescriptive on this point.

1. Hypertext links can be created, passed around, disappear.
2. Connections between cellular telephones and network bases.
3. Memory can be allocated and de-allocated, with references passed as parameters in method invocations.

Roughly, a link is determined by the sharing of names.
Action prefixes can be executed to change system connectivity over time.

About names

Names can be:

- channels
- identifiers
- values (data)
- objects
- pointers
- references
- locations
- encryption keys
- ...

Names can:

1. be created and destroyed
2. sent them around to share information
3. acquired to communicate with previously unknown processes
4. used for evaluation or communication
5. be tested to take decisions based on their values
6. used as private means of communication, e.g. to share secret
- ...
Syntax of $\pi$-calculus

We assume a countably infinite set of names $\mathcal{N}$ is defined.

\[
(\text{Processes}) \ P ::= \begin{align*}
S & \quad \text{sum} \\
| P_1 | P_2 & \quad \text{parallel composition} \\
| (v x) P & \quad \text{name restriction} \\
| ! P & \quad \text{replication}
\end{align*}
\]

\[
(\text{Sums}) \ S ::= \begin{align*}
0 & \quad \text{inactive process (nil)} \\
| \pi. P & \quad \text{prefix} \\
| S_1 + S_2 & \quad \text{choice}
\end{align*}
\]

\[
(\text{Prefixes}) \ \pi ::= \begin{align*}
\pi\langle y \rangle & \quad \text{sends } y \text{ on } x \\
| x(z) & \quad \text{substitutes for } z \text{ the name received on } x \\
| \tau & \quad \text{internal action} \\
| [x = y] \pi & \quad \text{matching: tests equality of } x \text{ and } y
\end{align*}
\]

Notation, Comments and Remarks

- $(v z) P$ is alike CCS restriction $P \setminus z$.
- $! P$ models replication and denotes the parallel composition of an arbitrary number of copies of $P$.
- $[x = y] \pi. P$ is known as name matching: it is equivalent to $\textbf{if } x = y \textbf{ then } \pi. P$.
- Occurrences of $0$ will sometimes be omitted, thus, e.g., $\pi\langle y \rangle. 0$ will be written $\pi\langle y \rangle$.
- $x(z)$ indicates input while $\pi\langle y \rangle$ indicates output.
- In $x(z). P \ e (v z) P$, the name $z$ is bound in $P$ (i.e., $P$ is the scope of such name). A name that is not bound is called free.
- $\text{fn}(P) \ e \ \text{bn}(P)$ are the sets of all free, resp. bound, names of $P$.
- We take processes up to alpha-conversion, denoted by $=_{\alpha}$, which permits renaming of a bound name with a fresh name that is not already used.
Name Binding

Names Bound by Input Prefix

The binding of a name $z$ in a $\pi$-calculus process $x(z).P$ resembles the binding of a variable $z$ in a $\lambda$-calculus term $\lambda z.M$:

- the free occurrences of $z$ in $M$ indicate the places where the argument will be substituted upon application (e.g. $(\lambda z.M)N$).
- the free occurrences of $z$ in $P$ indicate the places where the name received on channel $x$ will be substituted upon communication.

Names Bound by Restriction

Restriction $(\nu z)P$ is more than CCS restriction:

- for example, the process $(\nu z)x(z).Q$ can extrude the scope of the restricted name $z$ by passing it on channel $x$ to other processes
- the process $(\nu z)(x(z)|z(y).Q)$ can extrude the scope of the restricted name $z$ and then wait to receive some data on it.

Name Substitution

Definition (Substitution)

A substitution $\sigma : \mathcal{N} \to \mathcal{N}$ is a function on names that is the identity except on a finite set of names.

Notation

We write $x\sigma = \sigma(x)$ for $\sigma$ applied to $x$ and $X\sigma = \{x\sigma \mid x \in X\}$.

We write $[y_1, \ldots, y_n/x_1, \ldots, x_n]$ for the substitution $\sigma$ such that $x_i\sigma = y_i$ for $i = 1, \ldots, n$ and $x\sigma = x$ otherwise.

Capture Avoiding

When applying $\sigma$ to a process $P$ we want to rename only the free occurrences of names $x$ in $P$ with $x\sigma$, not the bound ones.

Moreover, unintended capture of names $x\sigma$ by binders of $P$ must be avoided.
**Alpha-Conversion**

How to define \( P[z/x] \) if \( P = y(z).\overline{z}(x) \)? \( y(z).\overline{z}(z) \) is a wrong answer!

**Definition (\( \alpha \)-Conversion)**

- If the name \( y \) does not occur at all in \( P \), then \( P[y/z] \) is the process obtained by replacing each free occurrence of \( z \) in \( P \) by \( y \).
- A change of input bound names in a process \( P \) is the replacement of a subterm \( x(y).Q \) of \( P \) by \( x(w).Q[w/y] \) for \( w \) not occurring in \( Q \).
- A change of restriction bound names in a process \( P \) is the replacement of a subterm \( (\nu y)Q \) of \( P \) by \( (\nu w)Q[w/y] \) for \( w \) not occurring in \( Q \).
- Two processes \( P \) and \( Q \) are \( \alpha \)-convertible, written \( P =_\alpha Q \) if \( Q \) can be obtained from \( P \) by a finite number of changes of bound names.

\[ P =_\alpha y(w).\overline{w}(x). \]

---

**Application of Substitution**

Below, when a prefix/process and a substitution are considered, we assume that all the bound names are chosen to be different from their free names and from the names of the substitution. (If not, we use \( \alpha \)-conversion first).

**Application of \( \sigma \) to Prefixes**

\[
\begin{align*}
(\overline{x}(y))\sigma &= \overline{x\sigma}(y\sigma) \\
(x(z))\sigma &= x\sigma(z) \\
\tau\sigma &= \tau \\
([x = y]\pi)\sigma &= [x\sigma = y\sigma]\pi\sigma
\end{align*}
\]

**Application of \( \sigma \) to Processes**

\[
\begin{align*}
0\sigma &= 0 \\
(\pi.P)\sigma &= \pi\sigma.P\sigma \\
(S_1 + S_2)\sigma &= S_1\sigma + S_2\sigma \\
(P_1|P_2)\sigma &= P_1\sigma|P_2\sigma \\
((\nu z)P)\sigma &= (\nu z)P\sigma \\
(!P)\sigma &= !P\sigma
\end{align*}
\]

\[ y(z).\overline{z}(x)[z/x] =_\alpha y(w).\overline{w}(z) \]
An LTS for $\pi$-calculus - First Attempt

Like for value passing CCS, take labels $\alpha$ among $ab$ (input), $\bar{a}b$ (output) and $\tau$.

(INP) $a(x).P \xrightarrow{ab} P[b/x]$

(OUT) $\exists(b).P \xrightarrow{\bar{a}b} P$

(PARL) $P \xrightarrow{\alpha} P' \overset{\alpha}{\mid} Q \xrightarrow{\alpha} P' \mid Q$

(PARR) $Q \xrightarrow{\alpha} Q' \overset{\alpha}{\mid} P \xrightarrow{\alpha} P \mid Q'$

(COML) $P \xrightarrow{\bar{a}b} P' \overset{\alpha}{\mid} Q \xrightarrow{\bar{a}b} Q' \overset{\alpha}{\mid} P \xrightarrow{\alpha} P' \mid Q'$

(COMR) $P \xrightarrow{\bar{a}b} P' \overset{\alpha}{\mid} Q \xrightarrow{\bar{a}b} Q' \overset{\alpha}{\mid} P \xrightarrow{\alpha} P' \mid Q'$

An LTS for $\pi$-calculus - First Attempt

Now some problem with restriction may arise

(RES$\tau$) $P \overset{\tau}{\rightarrow} P' \overset{\tau}{\mid} (\nu z)P \overset{\tau}{\rightarrow} (\nu z)P'$

(RESIN) $P \xrightarrow{ab} P' \overset{\nu z}{\mid} P \overset{ab}{\rightarrow} (\nu z)P'$

(RESOUT) $P \xrightarrow{\bar{a}b} P' \overset{\nu z}{\mid} P \overset{\bar{a}b}{\rightarrow} (\nu z)P'$

OK $z \neq a, b$

NO $z = a$

NO $z = b \neq a$

OK

?Extrusion?
An LTS for \( \pi \)-calculus - Second Attempt

Let us introduce another kind of labels \( \bar{a}(z) \) for representing scope extrusion of a bound name \( z \).

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>kind</th>
<th>( fn(\alpha) )</th>
<th>( bn(\alpha) )</th>
<th>( n(\alpha) )</th>
<th>( \alpha\sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ab )</td>
<td>input</td>
<td>( {a, b} )</td>
<td>( \emptyset )</td>
<td>( {a, b} )</td>
<td>( a\sigma b\sigma )</td>
</tr>
<tr>
<td>( \bar{a}b )</td>
<td>free output</td>
<td>( {a, b} )</td>
<td>( \emptyset )</td>
<td>( {a, b} )</td>
<td>( \bar{a}\sigma b\sigma )</td>
</tr>
<tr>
<td>( \bar{a}(z) )</td>
<td>bound output</td>
<td>( {a} )</td>
<td>( {z} )</td>
<td>( {a, z} )</td>
<td>( \bar{a}\sigma(z) )</td>
</tr>
<tr>
<td>( \tau )</td>
<td>internal</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \tau )</td>
</tr>
</tbody>
</table>

\[
(RES) \quad \frac{P \xrightarrow{\alpha} P'}{(\nu z)P \xrightarrow{\alpha} (\nu z)P'} \quad z \notin n(\alpha)
\]

\[
(OPEN) \quad \frac{P \xrightarrow{\bar{a}b} P'}{(\nu b)P \xrightarrow{\bar{a}(b)} P'} \quad a \neq b
\]

We remark that labels are not \( \alpha \)-convertible.

An LTS for \( \pi \)-calculus - Second Attempt

Let us revisit the rules for parallel composition (we omit the symmetric version of each rule).

\[
(PAR_L) \quad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \quad \text{bn}(\alpha) \cap fn(Q) = \emptyset
\]

\[
(COM_L) \quad \frac{P \xrightarrow{\bar{a}b} P'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}
\]

\[
(CLO_L) \quad \frac{P \xrightarrow{\bar{a}b} P'}{P \parallel \tau \xrightarrow{(\nu b)(P' \parallel Q')}} \quad b \notin fn(P)
\]
An LTS for \( \pi \)-calculus - Second Attempt

Rules for \( \tau \)-prefixes, matching, sum, \( \alpha \)-conversion (usually tacitly assumed) and replication.

\[
\begin{align*}
(TAU) & \quad \frac{\tau.P \rightarrow P}{\tau P \rightarrow P} \\
(MAT) & \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{[\alpha = \alpha]P \stackrel{\alpha}{\rightarrow} P'} \\
(SUM_L) & \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P + Q \stackrel{\alpha}{\rightarrow} P'} \\
(SUM_R) & \quad \frac{Q \stackrel{\alpha}{\rightarrow} Q'}{P + Q \stackrel{\alpha}{\rightarrow} Q'} \\
(ALPHA) & \quad \frac{P \stackrel{\alpha}{\rightarrow} Q}{P \stackrel{\alpha}{\rightarrow} P' \\
(REP) & \quad \frac{Q \stackrel{\alpha}{\rightarrow} Q'}{P \parallel !P \stackrel{\alpha}{\rightarrow} P'}
\end{align*}
\]

The usual rule \((REP)\) has several drawbacks: not in the SOS spirit, forces to work up to a structural congruence, makes the transition system not image-finite.

An LTS for \( \pi \)-calculus - Third Attempt

In the particular case of \( \pi \)-calculus, the rule \((REP)\) can be replaced by suitable SOS rules:

\[
\begin{align*}
(REP_{ACT}) & \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P \stackrel{\alpha}{\rightarrow} P' \parallel !P}
\end{align*}
\]

Is it enough? OK in the typical case where replication is input guarded. Generally NO: what if \( P = Q + R \) with \( Q \) and \( R \) able to interact!

\[
\begin{align*}
(REP_{COM}) & \quad \frac{P \stackrel{a}{\rightarrow} P' \quad P \stackrel{a}{\rightarrow} P''}{!P \rightarrow (P' | P'') \parallel !P} \\
(REP_{CLO}) & \quad \frac{P \stackrel{z}{\rightarrow} P' \quad P \stackrel{z}{\rightarrow} P''}{!P \rightarrow ((\nu z)(P' | P'')) \parallel !P}
\end{align*}
\]

\( z \notin \text{fn}(P) \)
Structural Congruence for $\pi$-calculus - I

Why Structural Congruence?

The syntax of $\pi$-calculus processes is to some extent too concrete (even when taken up-to $\alpha$-conversion):

- The order in which we compose processes in parallel should not matter.
- The order in which we compose processes in sums should not matter.
- The order in which we restrict names should not matter.

In fact, we shall see that processes differing for the above aspects are always equivalent.

By taking processes up to a suitable structural congruence we can:

- Write processes in a canonical form.
- Represent all possible interactions with few rules.

---

Structural Congruence for $\pi$-calculus - II

\[
\begin{align*}
P \mid 0 & \equiv P & P_1 \mid P_2 & \equiv P_2 \mid P_1 & P_1 \mid (P_2 \mid P_3) & \equiv (P_1 \mid P_2) \mid P_3 \\
S + 0 & \equiv S & S_1 + S_2 & \equiv S_2 + S_1 & S_1 + (S_2 + S_3) & \equiv (S_1 + S_2) + S_3 \\
!P & \equiv !P & [a = a]_\pi.P & \equiv \pi.P & a \notin \text{fn}(P) & \frac{P \mid (\nu a)Q \equiv (\nu b)(\nu a)P}{P \mid (\nu a)Q \equiv (\nu b)(\nu a)P} \\
(\nu a)0 & \equiv 0 & (\nu a)(\nu b)P & \equiv (\nu b)(\nu a)P & \\
P & \equiv P & P & \equiv Q & P \equiv Q & Q \equiv R & (\text{equivalence}) & P \equiv R \\
& \frac{P \equiv Q}{Q \equiv P} & \frac{P \equiv Q}{Q \equiv R} & (\text{congruence}) & \\
& \frac{P =_\alpha P'}{P \equiv P'} & \frac{P \equiv P'}{\mathbb{C}[P] \equiv \mathbb{C}[P']} & & \\
& \frac{P \equiv P'}{P \equiv P'} & & & \\
\end{align*}
\]
Canonical Form

For each π-calculus process $P$ there exist:

- a finite number of names $x_1, \ldots, x_k$,
- a finite number of sums $S_1, \ldots, S_n$, and
- a finite number of processes $P_1, \ldots, P_m$ such that

$$P \equiv (\nu x_1) \ldots (\nu x_k) (S_1 | \ldots | S_n | !P_1 | \ldots | !P_m)$$

The structural congruence allows one to rearrange the syntactic terms describing π-calculus processes so that any two possible interacting subterms can be put side by side (in parallel composition).

Thus, all interactions can now be expressed by considering only a small number of possibilities.

Reduction Semantics for π-calculus

The so-called reduction semantics focuses on internal moves $P \rightarrow Q$ only.

$(RTAU)$ \[ (\tau . P + S) \rightarrow P \]

$(RCOM)$ \[ (a(x) . P_1 + S_1)(\exists (b) . P_2 + S_2) \rightarrow P_1[b/x] | P_2 \]

$(RPAR)$ \[ P \rightarrow P' \]
\[ P \mid Q \rightarrow P' \mid Q \]

$(RRES)$ \[ (\nu a)P \rightarrow (\nu a)P' \]

$(RSTRUCT)$ \[ P \equiv Q \]
\[ Q \rightarrow Q' \]
\[ Q' \equiv P' \]
\[ P \rightarrow P' \]
Harmony Lemma

The reduction semantics and the LTS semantics can be tightly reconciled.

Notation

Given two relations \( \mathcal{R} \) and \( \mathcal{S} \) on processes, we write \( P \mathcal{R} \mathcal{S} Q \) if there exists a process \( R \) such that \( P \mathcal{R} R \) and \( R \mathcal{S} Q \).

Theorem (Harmony Lemma)

For any \( \pi \)-calculus process \( P \) we have:

1. \( P \equiv \alpha \rightarrow P' \) implies \( P \mathcal{\rightarrow} \equiv P' \)
2. \( P \leftarrow\rightarrow P' \) if and only if \( P \mathcal{\leftarrow} \equiv P' \)

Some Exercises

1. Show that the set \( fn(P) \) is finite for any \( P \).
2. Is it the case that \( (\nu x)P \equiv (\nu x)P' \) implies \( P \equiv P' \)?
3. Argue why it is not the case that \( !(\nu x)P \equiv (\nu x)!P \).
4. A process is called image finite if for any label \( \alpha \) there are finitely many processes \( Q \) (up to \( \alpha \)-conversion) such that \( P \alpha \rightarrow Q \). Find a process \( P \) which is not image finite when considering rule \( (REP) \).
5. Argue why it is not the case that \( P \leftarrow\rightarrow P' \) if and only if \( P \mathcal{\leftarrow} P' \) when using rules \( (REP_{ACT}) \), \( (REP_{COM}) \) and \( (REP_{CLO}) \).
6. Show that in general \( P \alpha \rightarrow \equiv P' \) does not imply that \( P \equiv \alpha \rightarrow P' \).
Electoral Propaganda

We shall see how the use of restricted channels can prevent intrusions.

Assume we want to campaign for Antonio and have set up the following scenario:

**Naive Campaigning**

<table>
<thead>
<tr>
<th>Speaker</th>
<th>$\triangleq \overline{\text{air}(\text{vota antonio})}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microphone</td>
<td>$\triangleq \overline{\text{air}(x) \cdot \overline{\text{wire}(x)}}$</td>
</tr>
<tr>
<td>Loudspeaker</td>
<td>$\triangleq \overline{\text{wire}(y) \cdot \text{highvolume}(y)}$</td>
</tr>
<tr>
<td>Ad</td>
<td>$\triangleq \text{Speaker} \</td>
</tr>
</tbody>
</table>

This system will evolve as follows:

$$
\text{Ad} \quad \rightarrow \quad \overline{\text{wire}(\text{vota antonio})} \ \mid \ \text{Loudspeaker}
$$

$$
\rightarrow \quad \overline{\text{highvolume}(\text{vota antonio})}
$$

Electoral Propaganda and Intrusions

<table>
<thead>
<tr>
<th>Speaker</th>
<th>$\triangleq \overline{\text{air}(\text{vota antonio})}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microphone</td>
<td>$\triangleq \overline{\text{air}(x) \cdot \overline{\text{wire}(x)}}$</td>
</tr>
<tr>
<td>Loudspeaker</td>
<td>$\triangleq \overline{\text{wire}(y) \cdot \text{highvolume}(y)}$</td>
</tr>
<tr>
<td>Ad</td>
<td>$\triangleq \text{Speaker} \</td>
</tr>
</tbody>
</table>

Let

$$
\text{Rival} \quad \triangleq \overline{\text{wire}(z) \cdot \overline{\text{wire}(\text{vota ciccio})}}
$$

$$
\text{Ad} \mid \text{Rival} \quad \rightarrow \quad \overline{\text{wire}(\text{vota antonio})} \ \mid \ \text{Loudspeaker} \mid \text{Rival}
$$

$$
\rightarrow \quad \overline{\text{wire}(\text{vota ciccio})} \ \mid \ \text{Loudspeaker}
$$

$$
\rightarrow \quad \overline{\text{highvolume}(\text{vota ciccio})}
$$

Rival could use wire because it is a public channel. Instead, a secure propaganda would be:

$$
\text{SecureAd} \quad \triangleq \ (\nu \ \text{air, wire})(\text{Speaker} \ | \ \text{Microphone} \ | \ \text{Loudspeaker})
$$
Establishing Secure Communication Channels

Consider two processes Alice and Bob, that want to establish a secret channel using a Trusted Server with which they have a trustworthy (secret) communication link. We have that

- $c_{AS}$ is the communication channel between Alice and the server
- $c_{BS}$ is the communication channel between Bob and the server
- $c_{AB}$ is the new (secure) communication channel between Alice and Bob we want to establish.

We can code Alice, Bob and the Server as follows:

\[
A \triangleq (\nu c_{AB}) \overline{c_{AS}}(c_{AB}) \cdot \overline{c_{AB}} \langle \text{mess} \rangle \\
S \triangleq !c_{AS}(x) \cdot \overline{c_{BS}}(x) \mid !c_{BS}(y) \cdot \overline{c_{AS}}(y) \\
B \triangleq c_{BS}(z) . z(w) . \langle \text{use } z \rangle
\]

\[
(\nu c_{AS}, c_{BS})(A \mid S \mid B) \overset{\rightarrow}{\rightarrow} (\nu c_{AS}, c_{BS}, c_{AB})(S \mid \langle \text{use } \text{mess} \rangle)
\]

A Print Server

Printers and Spooler

\[
P_i \triangleq !\text{done}_i() . (\nu \text{job}) \overline{\text{free}_i}(\text{job}) . \text{job}(d) \cdot \overline{\text{done}_i}() \\
P_i \triangleq (\nu \text{done}_i)(\overline{\text{done}_i}() \mid P'_i) \\
S \triangleq !\text{pr}(c) . \sum_i \text{free}_i(j) \cdot \overline{\text{c}}(j) \\
PS \triangleq (\nu \text{free}_1) \ldots (\nu \text{free}_n)(P_1 \ldots | P_n | S)
\]

A Client: $C \triangleq (\nu \text{req}) \overline{\text{pr}(\text{req})} \cdot \text{req}(j) \cdot \overline{\text{j}}(\text{doc})$

Note that $fn(P_i) = \{\text{free}_i\}, fn(PS) = \{\text{pr}\}$ and $fn(C) = \{\text{pr}, \text{doc}\}$.

\[
PS | C \overset{\rightarrow}{\rightarrow} (\nu \text{req})(\nu \text{free}_1 \ldots \text{free}_n)(P_i | P_i | S \mid \sum_j \text{free}_i(j) \cdot \overline{\text{req}}(j) \mid \overline{\text{req}}(j) \cdot \overline{\text{j}}(\text{doc}) \\
\overset{\rightarrow}{\rightarrow} (\nu \text{req}, \text{job}', \overline{\text{done}_k})(PS'_k \mid \text{job}'(d) \cdot \overline{\text{done}_k}() \mid \overline{\text{req}}(\text{job'}) \mid \overline{\text{req}}(j) \cdot \overline{\text{j}}(\text{doc}) \\
\overset{\rightarrow}{\rightarrow} (\nu \text{req}, \text{job}', \overline{\text{done}_k})(PS'_k \mid \text{job}'(d) \cdot \overline{\text{done}_k}()) \\
\equiv (\nu \text{req}, \text{job}')(PS) \equiv PS
\]
Early Bisimilarity

Strong bisimilarity on $\pi$-calculus processes is analogous to strong bisimilarity on CCS processes... but there is one important caveat that regards the freshness of bound names in the labels $\bar{a}(z)$:

\[
P \triangleq (\nu z)\bar{a}(z).0 \\
Q \triangleq P \mid (\nu b)\bar{b}(y).0
\]

There is no apparent reason why one should distinguish $P$ from $Q$, but unfortunately $P$ can perform the action $\bar{a}(y)$ (taking $y$ as a fresh name), while $Q$ cannot!

Definition (Strong Bisimilarity)

**Strong bisimilarity** is the largest symmetric relation $\sim$ s.t. if $P \sim Q$:

- if $P \xrightarrow{\alpha} P'$ with $bn(\alpha) \cap fn(Q) = \emptyset$, then $Q \xrightarrow{\alpha} \sim P'$.

Some Facts

**Lemma (Structural congruence is a strong bisimulation)**

$P \equiv Q$ implies $P \sim Q$

**Theorem (Strong bisimilarity is a non-input congruence)**

- $S \sim S'$ implies $S + T \sim S' + T$
- $P \sim P'$ implies $(\nu z)P \sim (\nu z)P'$, $P\mid Q \sim P'\mid Q$, $!P \sim !P'$
- $P \sim P'$ implies $\tau.P \sim \tau.P'$, $\bar{x}(y).P \sim \bar{x}(y).P'$
- $\pi.P \sim \pi'.P'$ implies $[x = y]\pi.P \sim [x = y]\pi'.P'$

Strong bisimilarity is not preserved by substitutions and input

\[
\bar{z}(\bar{a}(\bar{a}(\bar{a})) \sim \bar{z}(\bar{a}(\bar{a} + \bar{a} = \bar{a}).\bar{z}(\bar{a}))
\]

but

\[
(\bar{z}(\bar{a}(\bar{a}(\bar{a}))[a/z] \not\sim (\bar{z}(\bar{a}(\bar{a} + \bar{a} = \bar{a}).\bar{z}(\bar{a}))[a/z])
\]

\[
x(z).((\bar{z}(\bar{a}(\bar{a})) \not\sim x(z).((\bar{z}(\bar{a}(\bar{a} + \bar{a} = \bar{a}).\bar{z}(\bar{a}))
\]
Strong Full Bisimilarity

Definition (Strong Full Bisimilarity)

\( P \) and \( Q \) are **strong full bisimilar**, written \( P \sim^c Q \), if \( P\sigma \sim Q\sigma \) for every substitution \( \sigma \).

\[
\bar{z}(\_)|a(\_) \not\sim^c \bar{z}(\_).a(\_)+a(\_).\bar{z}(\_)
\]

\[
\bar{z}(\_)|a(\_) \sim^c \bar{z}(\_).a(\_)+a(\_).\bar{z}(\_)+[z=a]_T
\]

Theorem (Strong full bisimilarity is a congruence)

- \( P \sim^c Q \) implies \( \Box[P] \sim^c \Box[Q] \) for any context \( \Box[\_]. \).
- \( P \sim^c Q \) implies \( P \sim Q \), but not vice versa.
- \( \sim^c \) is the largest congruence included in \( \sim \).

Reduction Based Bisimilarity

Definition (Reduction Bisimilarity)

**Reduction bisimilarity** is the largest symmetric relation \( \equiv \) s.t. if \( P \equiv Q \):

- if \( P \leftrightarrow P' \), then \( Q \leftrightarrow \equiv P' \).

What is the discriminating power of reduction semantics?

The trivial notion of **reduction bisimilarity** is seriously defective: e.g., it relates any two processes with no internal transition, like \( \overline{a}(b).0 \) and \( 0 \).

A **reduction congruence** relating \( P \) and \( Q \) whenever \( \Box[P] \) and \( \Box[Q] \) are reduction bisimilar for every context \( \Box[\_]. \) is also not very convenient: \( \overline{a}(b).0 \) is distinguished by \( 0 \) and also by \( \overline{a}(c).0 \) (guess with \( \Box[\_]. \) to use), but it cannot distinguish, e.g., \( !\tau.0 \) from \( \overline{a}(b).0|!\tau.0 \).

We need to observe something more about processes...
**Strong Barbed Bisimilarity - I**

**Definition (Barbs)**

A *barb* is the capability to emit and receive on certain channels:

- We write $P \downarrow_x$ if $P$ can perform an input action on $x$.
- We write $P \uparrow_x$ if $P$ can perform an output action on $x$.

Barbs express just structural properties (more abstract than LTS labels).

**Definition (Strong Barbed Bisimilarity)**

*Strong barbed bisimilarity* is the largest symmetric relation $\sim$ such that whenever $P \sim Q$:

- if $P \downarrow_{\mu}$ then $Q \downarrow_{\mu}$;
- if $P \hookrightarrow P'$ then $Q \hookrightarrow \sim P'$.

**Strong Barbed Bisimilarity - II**

**Lemma**

- $P \equiv Q$ implies $P \sim Q$
- $\sim$ is preserved by prefixing, sum and restriction.

Slightly better, but still not very satisfactory:

$\overline{a}(b).P \sim (\nu z)\overline{a}(z).Q$ for any $P$ and $Q$. 

Strong Barbed Equivalence

If we close $\sim$ by parallel composition, then we recover strong bisimilarity.

**Definition (Strong Barbed Equivalence)**

Two processes $P$ and $Q$ are **strong barbed equivalent**, written $P \sim Q$, if $P|R \sim Q|R$ for every $R$.

**Theorem (Strong Characterization 1)**

$P \sim Q$ if and only if $P \sim Q$.

---

Strong Barbed Congruence

If we close $\sim$ by any context, then we recover strong full bisimilarity.

**Definition (Strong Barbed Congruence)**

Two processes $P$ and $Q$ are **strong barbed congruent**, written $P \sim^c Q$, if $\mathbb{C}[P] \sim \mathbb{C}[Q]$ for every context $\mathbb{C}[.]$.

**Theorem (Strong Characterization 2)**

$P \sim^c Q$ if and only if $P \sim^c Q$.

Similar results are achieved in the weak case, which in fact gives the standard notion of equivalence for $\pi$-calculus processes (more satisfactory, because internal actions are abstracted away).
Extensions and Encodings

Over the years, $\pi$-calculus has been presented in many different flavours. We shall see some of the most known extensions.

Obvious questions will arise

- When are two calculi really different?
- Do they have the same expressive power?

To answer these questions, encodings have been devised for many pairs of languages, together with criteria for assessing their quality.

In this presentation we make some reasonable assumptions (non obvious when abstract semantics and context closure are involved).

- Translating a sub-calculus into a super-calculus is effortless.
- For the other direction, typically we present a natural translation that is sound but not complete, and thus it is not fully abstract. (It may happen that equivalent processes are mapped to non equivalent ones).

Polyadic Communication

A straightforward extension is to allow multiple (possibly none) arguments in communications:

$$
\pi ::= \overline{y}_1, \ldots, y_n \quad \text{ sends } \overline{y} \text{ on } x
$$

$$
| \ x(z_1, \ldots, z_n) \quad \text{ substitutes for } \overline{z} \text{ the names received on } x
$$

$$
| \ \tau \quad \text{ internal action}
$$

$$
| \ [x = y]\pi \quad \text{ matching: tests equality of } x \text{ and } y
$$

The semantics for a polyadic calculus is only notationally more complex than for the monadic calculus.

How do we treat communicating processes such that the arity of the input is different from the arity of the output?

A large variety of type systems has been developed, usually based on the so-called sort contexts, that can be used to catch conflicts at various depth, to prevent the use of ill-terms.
Polyadic vs Monadic Communication

Monadic processes are just a special instance of polyadic processes.

Vice versa, it is possible to translate (well-formed) polyadic processes in monadic ones. However, the translation (like many others we are going to see) is sound but not complete.

In particular, it may happen that two equivalent polyadic processes are mapped to non equivalent monadic processes.

Definition (From Polyadic Processes To Monadic Processes)

The translation is defined by:

$$[\overline{x_1, \ldots, y_n}.P] \triangleq (\nu k)\overline{x(k).\overline{y_1} \ldots \overline{y_n}.\overline{[P]}}$$

$$[x(z_1, \ldots, z_n).P] \triangleq x(k).k(z_1)\ldots k(z_n).\overline{[P]}$$

where $k$ must be fresh and $\overline{[\cdot]}$ is a homomorphism for all the remaining operators.

Non Blocking Output

Three features of $\pi$-calculus communication:

- **blocking input**: A process $P$ prefixed by an input action is blocked until the input is not received. $P$ needs to be suspended because its future behaviour might depend on the received values.

- **Processes Interaction**: Communication is the result of complementary action and is channel based.

- **blocking output**: A process $Q$ prefixed by an output is suspended until there is a process willing to receive his message.

In wide area network it does not make sense to have synchronous communication. Most of the nodes would be blocked waiting for each other.

There are variants of the calculus that make a different choice relatively to output actions.
Asynchronous $\pi$-Calculus

First Alternative
Sum is not considered, but syntax is otherwise unchanged

$$P \ ::= \ 0 \mid \pi.P \mid P_1 | P_2 \mid (\nu a)P \mid !P$$

and the following two rules are used to replace the (RCOM) rule.

$$\exists \langle b \rangle . P \mapsto \exists \langle b \rangle \mid P \quad a(x).P \mid \exists \langle b \rangle \mapsto P[b/x]$$

Second Alternative
Syntax is further modified by dropping the suffix of output actions

$$P \ ::= \ 0 \mid a(x).P \mid \exists \langle b \rangle \mid P_1 | P_2 \mid (\nu a)P \mid [a = b]P \mid !P$$

and the (RCOM) rule is replaced by

$$a(x).P \mid \exists \langle b \rangle \mapsto P[b/x]$$

Synchronous vs Asynchronous Communication - I

First Attempt

$$\llbracket \exists \langle y \rangle . P \rrbracket \triangleq \exists \langle y \rangle \mid [P]$$

NOT SATISFACTORY, because $P$ must remain blocked until the communication happens!

Second Attempt
In the translation of the output we prefix $P$ by the input of an ack, similarly in the translation of the input we must send the ack.

$$\llbracket \exists \langle y \rangle . P \rrbracket \triangleq \exists \langle y \rangle \mid a(-)[P]$$
$$\llbracket x(z).Q \rrbracket \triangleq x(z).\langle \exists \langle - \rangle \mid [Q] \rangle$$

NOT SATISFACTORY: $a$ is not fresh (other processes can interfere)!
Synchronous vs Asynchronous Communication - II

Third Attempt

We enforce the sharing of a private name \( a \) for sending the value and receiving the ack

\[
\begin{align*}
\llbracket \overline{\sigma}(y).P \rrbracket & \triangleq (\nu a)(\overline{\sigma}(a) \mid \overline{\sigma}(y) \mid a(-).\llbracket P \rrbracket) \\
\llbracket x(z).Q \rrbracket & \triangleq x(a).a(z).(\overline{\sigma}(-) \mid \llbracket Q \rrbracket)
\end{align*}
\]

NOT SATISFACTORY, because the sending of the value on \( a \) can be incorrectly fetched as the ack!

The solution

We can use two different private channels for sending the data and receiving the ack.

Synchronous vs Asynchronous Communication - III

Definition (From Synchronous Processes To Asynchronous Processes)

The translation is defined by:

\[
\begin{align*}
\llbracket \overline{\sigma}(y).P \rrbracket & \triangleq (\nu k)(\overline{\sigma}(k) \mid k(a).(\overline{\sigma}(y) \mid \llbracket P \rrbracket)) \\
\llbracket x(z).Q \rrbracket & \triangleq x(k).(\nu a)(\overline{k}(a) \mid a(z).\llbracket Q \rrbracket)
\end{align*}
\]

where \( a, k \) must be fresh and \( \llbracket - \rrbracket \) is a homomorphism for all the remaining operators (mixed sum cannot be encoded, while the encoding is possible, but more complex, for input-guarded and output-guarded sum).

1. The sender sends on \( x \) a fresh name \( k \) and waits to receive a new private channel \( a \) on this name that will use to send the data \( y \).
2. The receiver waits to receive on channel \( x \) the name \( k \) where to send the fresh name \( a \) (an ack) that will be used to receive the data.
Process Mobility

We know that process mobility is a possible alternative to link mobility. $HO\pi$ can model situation like:

- messages with executable attachments
- applets downloadable via HTTP
- remote code evaluation (a client sends a program to a server that executes it and then returns the result to the client).

Definition (Higher Order $\pi$-calculus)

$$P ::= \mathbf{0} \mid \eta P \mid P_1|P_2 \mid (\nu z)P \mid X$$

$$\eta ::= \tau \mid x(z) \mid x(Z) \mid \overline{x}(y) \mid \overline{x}(P)$$

where $X$ ranges over a set $\mathcal{V}$ of process variables with $\mathcal{V} \cap \mathcal{N} = \emptyset$.

Replication $!P$ can be encoded in $HO\pi$ as $(\nu r)( Dup \mid \overline{r}(P \mid Dup) )$, where $r$ must be fresh and $Dup \triangleq r(X) \cdot (X \mid \overline{r}(X) )$ is called duplicator.

Higher-Order vs First-Order

Definition (From HO Processes To FO Processes)

$$\overline{x}(P).Q \triangleq (\nu p)\overline{x}(p) \cdot (\overline{x}(Q) \mid !p(\_).\overline{x}(P) )$$

$$x(Z).R \triangleq x(n_Z).\overline{x}(R)$$

$$[X] \triangleq \overline{n_X}(\_ )$$

where $p$ is fresh and we assume that the names $n_X$ in bijection with process variables $X$ are not used elsewhere in the translated process. The encoding $\overline{\_}$ is a homomorphism for all the remaining operators.

- Instead of sending $P$, we place $P$ as a service on a fresh server $p$.
- Instead of receiving a process, we input such server name.
- Then, an invocation of $P$ is just an output on $p$.
- The replicated input guard $!p(\_ )$ blocks $P$ until it is invoked. It releases a running instance of $P$ upon invocation. Note the use of replication allows to serve an unbounded number of invocations.
Late Semantics - I

- Early and late transition relations differ only in input transitions.
- We introduced transitions $P \xrightarrow{xy} P'$ for every name $y$ that can be consumed from the input. They are called early transitions because the name $y$ is recorded in the label. The action $xy$ is called free input, with $\text{fn}(xy) = \{x, y\}$ and $\text{bn}(xy) = \emptyset$.
- Late transitions $P \xrightarrow{z} P'$ postpone the choice of the received name: $z$ is just a (fresh) placeholder for the name to be received, rather than the name itself. Bound input labels $x(z)$ are used instead of free input, with $\text{fn}(x(z)) = \{x\}$ and $\text{bn}(x(z)) = \{z\}$.
- Late bisimilarity is very close to early bisimilarity, but imposes a slightly stronger requirement when simulating input actions.
- Late bisimilarity is not a congruence, and late congruence is its closure under arbitrary substitutions.
- Late bisimilarity/congruence are strictly finer than their early versions.

Late Semantics - II

\begin{align*}
(INP'_L) \quad & a(z). P \xrightarrow{a(z)} P \\
(COM'_L) \quad & \frac{P \xrightarrow{a(z)} P' \quad Q \xrightarrow{a(b)} Q'}{P \mid Q \xrightarrow{\tau} P'[b/z] \mid Q'} \\
(CLO'_L) \quad & \frac{P \xrightarrow{a(z)} P' \quad Q \xrightarrow{a(z)} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')} \quad T
\end{align*}

$P \xrightarrow{ab} Q$ if and only if $\exists P', z$ such that $P \xrightarrow{a(z)} P'$ and $Q = P'[b/z]$.

Definition (Strong Late Bisimilarity)

Strong late bisimilarity is the largest symmetric relation $\sim_L$ s.t. if $P \sim_L Q$

- if $P \xrightarrow{\alpha} P'$ with $\alpha$ not an input, then $Q \xrightarrow{\alpha} P'$.
- if $P \xrightarrow{z} P'$, then $Q \xrightarrow{z} Q'$ and $P'[b/z] \sim_L Q'[b/z]$ for every $b$. 

Bruni, De Nicola (UNIPI, DSI-UNIFI) Process Algebras and Concurrent Systems IMT Lucca 59 / 145
Open Semantics

What we have learned:

- Bisimilarities have a co-inductive nature.
- Congruences are good.
- Bisimilarities are not congruences because free names are open to substitutions (e.g. via input prefixing).

*Open bisimilarity* can be defined co-inductively and it is a congruence... but maybe it distinguishes too much.

Some Exercises

- Show that $!P \neq !P$ but $!P \sim !P$.
- Show that $!!P \neq !P$ but $!!P \sim !P$.
- Let $P \triangleq \overline{a(b)} \cdot \overline{a(b)}$ and $Q \triangleq \overline{a(b)} | a(b)$. Show that $P \sim Q$, while $[P] \not\sim [Q]$, where $[.]$ denotes the translation to asynchronous $\pi$.
  (Hint: Proceed by contradiction, noticing that since $\sim$ is a congruence w.r.t. parallel composition, if $[P] \sim [Q]$ holds then also $[P]|a(x) \sim [Q]|a(x)$ should hold.)
- Give a (sound) encoding of Synchronous Polyadic $\pi$ in Asynchronous Polyadic $\pi$. 
From pi-calculus to spi-calculus - I

Aim
How to describe and analyze security protocols (for authentication, secrecy, electronic commerce)?

Wide Mouthed Frog Protocol
A simplified version of the protocol is usually expressed as:

Message 1 \( A \rightarrow S : \{ K_{AB} \}^{K_{AS}}_{c_{AS}} \)
Message 2 \( S \rightarrow B : \{ K_{AB} \}^{K_{SB}}_{c_{SB}} \)
Message 3 \( A \rightarrow B : \{ M \}^{K_{AB}}_{c_{AB}} \)

Ingredients
These protocols typically rely on
- cryptography
- communication channels
From pi-calculus to spi-calculus - II

Abstract level
We have already seen an example on how to use pi-calculus to describe (creation, passing and usage of) secret channels: the scoping of fresh names become the basis for security properties.

Cryptography?
However, the usual cryptographic operations of security protocols can hardly be modeled in pi-calculus. (Even if they could be expressed for a particular application domain, it is better to have them as first class citizen, because the use of cryptography is notoriously error-prone.)

Spi-calculus

Spi calculus
Spi-calculus is a mild extension of the pi-calculus designed for describing and analyzing cryptographic protocols. Moreover, w.r.t. other proposals, it is directly executable and it has a precise semantics.

Features
- environment as (any) arbitrary spi-calculus process (no need to model it explicitly)
- security guarantees (integrity and secrecy) as equivalences between spi-calculus processes
- two different spi-calculus processes for modeling the protocol and for specifying a magical, trivially correct, version of the protocol
## Syntax of (shared-key cryptography) spi-calculus

For convenience some simple data structures are included

\[
\begin{align*}
\text{(terms) } L, M, N &::= n &\text{name} \\
&\quad | (M, N) &\text{pair} \\
&\quad | 0 &\text{zero} \\
&\quad | \text{suc}(M) &\text{successor} \\
&\quad | \{ M \}_N &\text{shared-key encryption} \\
\text{(processes) } P, Q, R &::= 0 &\text{nil} \\
&\quad | \overline{M}(N).P &\text{output} \\
&\quad | M(x).P &\text{input} \\
&\quad | P|Q &\text{parallel} \\
&\quad | (\nu n)P &\text{restriction} \\
&\quad | !P &\text{replication} \\
&\quad | [M \text{ is } N]P &\text{match} \\
&\quad | \text{let } (x, y) = M \text{ in } P &\text{pair split} \\
&\quad | \text{case } M \text{ of } 0 : P \text{ suc}(x) : Q &\text{integer case} \\
&\quad | \text{case } M \text{ of } \{ x \}_N \text{ in } P &\text{shared-key decryption}
\end{align*}
\]
Notation, Comments and Remarks

- The syntax of the match operators is slightly different
- We have included terms for expressing natural numbers and pairs
- Correspondingly, in processes, we have two new forms:
  - let \((x, y) = M\) in \(P\) behaves as \(P[L/x, y]\) if the term \(M\) is a pair \((L, N)\), otherwise it can do nothing
  - case \(M\) of \(0 : P\) suc\((x) : Q\) behaves as \(P\) if the term \(M\) is 0, as \(Q[L/x]\) if \(M\) is suc\((N)\), otherwise it can do nothing
- We have introduced the term \(\{ M \}_N\): it represents the ciphertext obtained by encrypting the \(M\) under the key \(N\)
- Correspondingly, we have introduced the process case \(M\) of \(\{ x \}_N\) in \(P\). It attempts to decrypt the term \(M\) with the key \(N\): if \(M\) is a ciphertext of the form \(\{ L \}_N\), then the process behaves as \(P[L/x]\), otherwise it can do nothing
- The only way to decrypt an encrypted packet is to know its key
- An encrypted packet does not expose the key used to encrypt it
- The decryption algorithm can always detect whether a ciphertext was encrypted with the expected key or not

Wide Mouthed Frog Protocol

Informal Specification (again)

A simplified version of the protocol is usually expressed as:

Message 1  \(A \rightarrow S: \{ K_{AB}\}^{K_{AS}}_K\) on \(c_{AS}\)
Message 2  \(S \rightarrow B: \{ K_{AB}\}^{K_{SB}}_K\) on \(c_{SB}\)
Message 3  \(A \rightarrow B: \{ M\}^{K_{AB}}_K\) on \(c_{AB}\)

Spi-calculus formalization

\[
A(M) \triangleq (\nu K_{AB})c_{AS}(\{ K_{AB}\}^{K_{AS}}_K).c_{AB}(\{ M\}^{K_{AB}}_K)\\
S \triangleq c_{AS}(x).case\ x\ of\ \{ y\}^{K_{AS}}_K in\ c_{AB}(\{ y\}^{K_{SB}}_K)\\
B \triangleq c_{SB}(x).case\ x\ of\ \{ y\}^{K_{SB}}_K in\ c_{AB}(z).case\ z\ of\ \{ w\}^{Y}_{Y} in\ Use(w)\\
Sys(M) \triangleq (\nu K_{AS})(\nu K_{SB})(A(M) | S | B)
\]
Semantics of (shared-key) spi-calculus: Simplification

Simplification relation

\[ !P > P|!P \]

\[ [M \text{ is } M]P > P \]

let \((x, y) = (M, N)\) in \(P\) \(>\) \(P[M, N/x, y]\)

case 0 of 0 : \(P\) suc\((x)\) : \(Q\) \(>\) \(P\)

case suc\((M)\) of 0 : \(P\) suc\((x)\) : \(Q\) \(>\) \(Q[M/x]\)

case \(\{ M \}N\) of \(\{ x \}N\) in \(P\) \(>\) \(P[M/x]\)

Semantics of spi-calculus: Structural Congruence

Structural Congruence

\[ P \mid 0 \equiv P \quad P_1 \mid P_2 \equiv P_2 \mid P_1 \quad P_1 \mid (P_2\mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \]

\[ (\nu a)0 \equiv 0 \quad (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P \]

\[ a \notin \text{fn}(P) \quad \frac{P \equiv Q \quad Q \equiv P}{P \equiv R} \quad \text{(equivalence)} \]

\[ P \equiv P' \quad P \gg P' \quad P \equiv P' \]

\[ P \equiv P' \quad \frac{\mathbb{C}[P] \equiv \mathbb{C}[P']}{} \quad \text{(congruence)} \]
Semantics of (shared-key) spi-calculus: Reaction

Reaction Relation

(RCOM) \[ m(x).P \mid \overline{m(N)}.Q \leftrightarrow P[N/x] \ | \ Q \]

(RPAR) \[ P \leftrightarrow P' \]
\[ P \mid Q \leftrightarrow P' \mid Q \]

(RRES) \[ P \leftrightarrow P' \]
\[ (\nu a)P \leftrightarrow (\nu a)P' \]

(RSTRUCT) \[ P \equiv Q \quad Q \leftrightarrow Q' \quad Q' \equiv P' \]
\[ P \leftrightarrow P' \]

Indistinguishability

Consider the process

\[ P(M) \triangleq (\nu K)c\langle \{ M \} \rangle \]

it sends message \( M \) encrypted under a fresh key \( K \) on public channel \( c \).

Should \( P(M) \) and \( P(M') \) be distinguishable?

Intuitively not, because there is no observer that can discover \( K \) and tell whether \( M \) or \( M' \) is sent under \( K \).

But \( P(M) \) are not structural congruent and transmit different messages on \( c \): an equivalence like weak bisimilarity would distinguish them.

We need a coarse-grained equivalence that cannot distinguish between \( P(M) \) and \( P(M') \)!
Semantics of spi-calculus: Barbs

- We write $P \downarrow_x$ if $P$ can perform an input action on $x$.
- We write $P \downarrow_{\overline{x}}$ if $P$ can perform an output action on $x$.

\[
\begin{align*}
\frac{m(x).P \downarrow_m}{P \downarrow_\beta} & \quad \frac{\overline{m}(N).P \downarrow_{\overline{m}}}{P \equiv Q \quad Q \downarrow_\beta} \\
\frac{(P \mid Q) \downarrow_\beta}{(\nu m)P \downarrow_\beta} & \quad \beta \notin \{m, \overline{m}\}
\end{align*}
\]

- We write $P \downarrow_x$ if $P$ can input on $x$ after some reductions.
- We write $P \downarrow_{\overline{x}}$ if $P$ can output on $x$ after some reductions.

\[
\frac{P \downarrow_\beta}{P \downarrow_\beta} \quad \frac{P \leftrightarrow Q \quad Q \downarrow_\beta}{P \downarrow_\beta}
\]

Semantics of spi-calculus: Testing equivalence

The idea is to test processes against all possible environments.

**Definition (Testing Preorder)**

We call a test any pair $(R, \beta)$ comprising a process $R$ and a barb $\beta$. $P \sqsubseteq Q$ iff for any test $(R, \beta)$, if $(P \mid R) \downarrow_\beta$ then $(Q \mid R) \downarrow_\beta$

**Definition (Testing Equivalence)**

$P \simeq Q$ iff $P \sqsubseteq Q$ and $Q \sqsubseteq P$

**Theorem (Some facts about testing)**

- Structural congruence implies testing equivalence
- Testing equivalence is a congruence
- Barbed congruence implies testing equivalence, but not vice versa
Wide Mouthed Frog Protocol: Guarantees - I

Authenticity (or integrity)

We want to make sure that an attacker cannot cause B to Use other messages (i.e., that B always applies Use to the message M that A sends).

A magical, trivially correct specification is

\[ A(M) \triangleq (\nu K_{AB})_{c_{AS}}(\{ K_{AB} \}_{K_{AS}})_{c_{AB}}(\{ M \}_{K_{AB}}) \]
\[ S \triangleq c_{AS}(x).\text{case } x \text{ of } \{ y \}_{K_{AS}} \text{ in } c_{SB}(\{ y \}_{K_{SB}}) \]
\[ B_{\text{spec}}(M) \triangleq c_{SB}(x).\text{case } x \text{ of } \{ y \}_{K_{SB}} \text{ in } c_{AB}(z).\text{case } z \text{ of } \{ w \}_{y} \text{ in } \text{Use}(M) \]
\[ \text{Sys}_{\text{spec}}(M) \triangleq (\nu K_{AS})(\nu K_{SB})(A(M) \mid S \mid B_{\text{spec}}(M)) \]

Authenticity

Does \( \text{Sys}_{\text{spec}}(M) \simeq \text{Sys}(M) \) for all M?

Wide Mouthed Frog Protocol: Guarantees - II

Secrecy

We want to make sure that M cannot be read while in transit from A to B (i.e., the whole protocol does not reveal M if Use does not reveal M).

\[ A(M) \triangleq (\nu K_{AB})_{c_{AS}}(\{ K_{AB} \}_{K_{AS}})_{c_{AB}}(\{ M \}_{K_{AB}}) \]
\[ S \triangleq c_{AS}(x).\text{case } x \text{ of } \{ y \}_{K_{AS}} \text{ in } c_{SB}(\{ y \}_{K_{SB}}) \]
\[ B \triangleq c_{SB}(x).\text{case } x \text{ of } \{ y \}_{K_{SB}} \text{ in } c_{AB}(z).\text{case } z \text{ of } \{ w \}_{y} \text{ in } \text{Use}(w) \]
\[ \text{Sys}(M) \triangleq (\nu K_{AS})(\nu K_{SB})(A(M) \mid S \mid B) \]
\[ \text{Sys}(M') \triangleq (\nu K_{AS})(\nu K_{SB})(A(M') \mid S \mid B) \]

Secrecy

Does Use(M) \( \simeq \) Use(M') imply Sys(M) \( \simeq \) Sys(M') for all M, M'?
 spi-calculus variants: Hashing

Likewise pi-calculus, there are several versions of the spi-calculus: they differ in particular in what cryptographic constructs they include.

Definition (Hashing)

A cryptographic hash function \( \mathcal{H} \) is such that (computationally speaking) it is very expensive to recover an input from its image or to find two inputs with the same image.

\[
\begin{align*}
(\text{terms}) \quad L, M, N &::= \ldots \mid \mathcal{H}(N) \quad \text{hash of } M
\end{align*}
\]

In spi-calculus we pretend that “very expensive” means impossible.

- No construct to recover \( M \) from \( \mathcal{H}(M) \), thus \( \mathcal{H} \) cannot be inverted.
- \( \mathcal{H} \) is free of collisions because \( \mathcal{H}(M) \neq \mathcal{H}(M') \) when \( M \neq M' \).

Spi-calculus variants: Public-key encryption - 1

Definition (Key pairs)

Traditional public-key encryption systems are based on key pairs:

- one of the keys is private to one principal
- the other key is public
- neither key can be recovered from the other (or at most the public key can be derived from the private one, but not vice versa)
- any principal can encrypt a message using the public key
- an encrypted packet does not reveal the public key that was used to encrypt it
- only a principal that has the corresponding private key can then decrypt the message
- the decryption algorithm can always detect whether a ciphertext was encrypted with the expected public key or not
Spi-calculus variants: Public-key encryption - II

**Definition (Public-key encryption in spi-calculus)**

Key pairs are formed by $M^+$ and $M^-$ for each $M$.

(terms) $L, M, N ::= \ldots$ as usual  
$| M^+$ public key  
$| M^-$ private key  
$| \{M\}_N$ public-key encryption

(processes) $P, Q, R ::= \ldots$ as usual  
$| \text{case } M \text{ of } \{x\}_N \text{ in } P$ decryption

**Simplification relation**

\[
\text{case } \{M\}_N^+ \text{ of } \{x\}_N^- \text{ in } P \quad > \quad P[M/x]
\]

Spi-calculus variants: Digital signatures

**Definition (Digital signatures)**

Key pairs appear also in digital signatures:
- private keys are used for signing  
- public keys are used for checking signatures

(terms) $L, M, N ::= \ldots$ as usual  
$| M^+$ public key  
$| M^-$ private key  
$| \{M\}_N$ private-key signature

(processes) $P, Q, R ::= \ldots$ as usual  
$| \text{case } M \text{ of } \{x\}_N \text{ in } P$ signature check

**Simplification relation**

\[
\text{case } \{M\}_N^- \text{ of } \{x\}_N^+ \text{ in } P \quad > \quad P[M/x]
\]
Join Calculus

Orc
Orc

Orc is an elegant language proposed by Cook and Misra as a basic programming model for structured orchestration of services, whose primitives meets simplicity with yet great generality:

- The basic computational entities orchestrated by an Orc expression are not just web services but, more generally, site names.
- Site names can be passed as arguments in site call, thus allowing a disciplined usage of name mobility.
- As a workflow language, it has been shown that Orc can straightforwardly encode all most common workflow patterns.
- A prototype implementation of Orc is also available.

Orc Sites

Orc relies on the basic notion of site, an abstraction amenable for:

- being invoked
- publishing values

Site calls

Site calls are the simplest Orc expressions:

- A site call can be a RMI, a call to a monitor procedure, to a function or to a web service.
- Each invocation to a site $s$ elicits at most one response value published by $s$.
- A site computation might itself start other orchestrations, store effects locally and make (or not) such effects visible to clients.
- Sites can be composed by means of few operators to form expressions.
Orc Expressions

Orc neatly separates orchestration from computation:

- Orc expressions can be considered like scripts to be invoked, e.g., within imperative programming languages
- the syntax for assigning the result of an expression \( e \) to a variable \( z \) is \( z \in e \)
- Orc expressions can involve wide-area computation over multiple servers.

Contrary to site calls, an expression can, in principle, publish any number of response value

The assignment symbol \( \in \) (due to Hoare) in \( z \in e \) makes explicit that \( e \) can return zero or more results, one of which is assigned to \( z \).

Orc composition principles

Orc has three composition principles for building expressions:

- ordinary parallel composition \( f \mid g \), called symmetric parallel (e.g., the parallel of two site calls can produce zero, one or two values)

- sequencing \( f > x > g \): a fresh copy \( g[v/x] \) of \( g \) is executed on any value \( v \) published by \( f \) (i.e., a pipeline is established from \( f \) to \( g \)).

- asymmetric parallel composition \( f \) where \( x \in g \): \( f \) and \( g \) start in parallel, but all sub-expressions of \( f \) that depend on the value of \( x \) must wait for \( g \) to publish a value. When \( g \) produces a value it is assigned to \( x \) and that side of the orchestration is cancelled (i.e., it allows lazy evaluation, selection and pruning).

Sequencing and asymmetric parallel composition, take inspiration from universal and existential quantification, respectively.
Orc Syntax

(Expressions) $e, f, g ::=$

- $0$ nil
- $M\langle p_1, \ldots, p_n \rangle$ site call
- $f > x > g$ sequencing
- $f|g$ symmetric parallel
- $g \mathbf{where} x \in f$ asymmetric parallel
- $E\langle p_1, \ldots, p_n \rangle$ expression call

(Definitions) $D ::= E(x_1, \ldots, x_n) \triangle f$ expression definition

(Parameters) $p, q, r ::= x$ variable
- $c$ constant
- $M$ site

- $x$ is bound (with scope $g$) in $f > x > g$ and $g \mathbf{where} x \in f$
- the free variables of an expression $e$ are denoted by $fv(e)$
- if $x \notin fv(g)$ we abbreviate $f > x > g$ by writing $f \gg g$

Orc Semantics: Actions

The operational semantics of Orc is given by a Labelled Transition Systems defined in the SOS style

Transition Labels

- $M(\bar{c}, k)$ denotes a site call
- $k?c$ denotes a site response
- $!c$ denotes a locally published value
- $\tau$ denotes an internal action

The abstract semantics considered in the literature are trace equivalence and strong bisimilarity
Orc Semantics: Site Call

Two special auxiliary sites are \( \text{let}(x_1, \ldots, x_n) \) and \( \text{Signal} \).

\[
\begin{align*}
k & \text{ globally fresh} \\
M \langle \epsilon \rangle & \xrightarrow{M(\epsilon,k)} \ ?k & (\text{SITECALL}) \\
\quad ?k & \xrightarrow{k?\epsilon} \ \text{let}(c) & (\text{SITERET}) \\
\text{let}(c) & \xrightarrow{!c} 0 & (\text{LET}) \\
\text{Signal} & \xrightarrow{!\{} 0 & (\text{SIGNAL})
\end{align*}
\]

Getting the latest news of date \( d \) from CNN

\[
\text{CNN}(3\text{June2006}) \xrightarrow{\text{CNN}(3\text{June2006},k)} \ ?k \xrightarrow{k?\text{GiantAfricanLizardsInvadeFlorida}} \!\text{GiantAfricanLizardsInvadeFlorida} 0 \\
z \in \text{CNN}(d) \xrightarrow{\implies} z = \text{GiantAfricanLizardsInvadeFlorida}
\]

Orc Semantics: Parallel Composition

\[
\begin{align*}
g & \xrightarrow{l} g' & (\text{SYMLEFT}) \\
g | f & \xrightarrow{l} g' | f \\
f & \xrightarrow{l} f' & (\text{SYMRIGHT}) \\
g | f & \xrightarrow{l} g | f'
\end{align*}
\]

Getting news from CNN and BBC

\[
\begin{align*}
\text{CNN}(3\text{June2006}) & \mid \text{BBC}(3\text{June2006}) \xrightarrow{\text{CNN}(3\text{June2006},k_{\text{CNN}})} \\
?k_{\text{CNN}} & \mid \text{BBC}(3\text{June2006}) \xrightarrow{?k_{\text{BBC}}?\text{GiantUsaTouristsInvadeMadagascar}} \\
?k_{\text{CNN}} & \mid ?k_{\text{BBC}} \xrightarrow{k_{\text{BBC}}?\text{GiantUsaTouristsInvadeMadagascar}} \\
?k_{\text{CNN}} & \mid \text{let}(\text{GiantUsaTouristsInvadeMadagascar}) \xrightarrow{k_{\text{CNN}}?\text{GiantAfricanLizardsInvadeFlorida}} \\
\text{let}(\text{GiantAfrican...}) & \mid \text{let}(\text{GiantUsa...}) \xrightarrow{!\text{GiantAfricanLizardsInvadeFlorida}} \\
z \in \text{CNN}(d) & \mid \text{BBC}(d) \xrightarrow{\implies} z = \text{GiantAfricanLizardsInvadeFlorida}
\end{align*}
\]
Orc Semantics: Sequential Composition

\[ f \xrightarrow{l} f' \quad l \neq !c \]
\[ f > x > g \xrightarrow{l} f' > x > g \quad \text{(SEQ)} \]
\[ f \xrightarrow{!c} f' \]
\[ f > x > g \xrightarrow{\tau} (f' > x > g) | g[c/x] \quad \text{(SEQPIPE)} \]

Getting all news from CNN and BBC by email

\[ (\text{CNN}(d) \mid \text{BBC}(d)) > n \xrightarrow{\text{Email}(\text{rb@gmail.it}, n)} \]
\[ \text{CNN}(d, k_{\text{CNN}}) \xrightarrow{\text{BBC}(d, k_{\text{BBC}})} \]
\[ (\text{?}k_{\text{CNN}} \mid ?k_{\text{BBC}}) > n \xrightarrow{\text{Email}(\text{rb@gmail.it}, n)} \text{?GantUsaTouristsInvadeMadagascar} \]
\[ (\text{?}k_{\text{CNN}} \mid \text{let}(\text{GiantUsa...})) > n \xrightarrow{\text{Email}(\text{rb@gmail.it}, n)} \text{?GantUsaTouristsInvadeMadagascar} \]
\[ (\text{?}k_{\text{CNN}} \mid 0) > n \xrightarrow{\text{Email}(\text{rb@gmail.it}, n)} \text{Email}(\text{rb@gmail.it}, \text{GiantUsa...}) \]
\[ \xrightarrow{\text{Email}(\text{rb@gmail.it}, \text{GiantMexican...})} \]

Orc Semantics: Asymmetric Parallel Composition

\[ g \xrightarrow{l} g' \]
\[ g \text{ where } x : \in f \xrightarrow{l} g' \text{ where } x : \in f \quad \text{(A.L.)} \]
\[ f \xrightarrow{l} f' \quad l \neq !c \]
\[ g \text{ where } x : \in f \xrightarrow{l} g \text{ where } x : \in f' \quad \text{(A.R.)} \]
\[ f \xrightarrow{!c} f' \]
\[ g \text{ where } x : \in f \xrightarrow{\tau} g[c/x] \quad \text{(A.P.)} \]

Getting one news from CNN and BBC by email

\[ \text{Email}(\text{rb@gmail.it}, n) \text{ where } n : \in (\text{CNN}(d) \mid \text{BBC}(d)) \xrightarrow{\text{CNN}(d, k_{\text{CNN}})} \text{BBC}(d, k_{\text{BBC}}) \]
\[ \text{Email}(\text{rb@gmail.it}, n) \text{ where } n : \in (?k_{\text{CNN}} \mid ?k_{\text{BBC}}) \xrightarrow{\text{GiantUsa...}} \]
\[ \text{Email}(\text{rb@gmail.it}, n) \text{ where } n : \in (?k_{\text{CNN}} \mid \text{let}(\text{GiantUsa...})) \xrightarrow{\tau} \]
\[ \text{Email}(\text{rb@gmail.it}, \text{GiantUsaTouristsInvadeMadagascar}) \]
Orc Semantics (in one slide)

\[
\begin{align*}
& \text{globally fresh } k \\
& \frac{M(c) \xrightarrow{k?c} c}{?k \xrightarrow{k?c} \text{let}(c)} \quad \text{(SITECALL)} \\
& \frac{?k \xrightarrow{k?c} \text{let}(c)}{
}\end{align*}
\]

\[
\begin{align*}
& f \xrightarrow{l} f' \quad l \neq 1c \\
& \frac{f > x > g \xrightarrow{l} f' > x > g}{f \xrightarrow{1c} f'} \quad \text{(SEQ)} \\
& \frac{f > x > g \xrightarrow{\tau} (f' > x > g) \mid g[c/x]}{g \xrightarrow{l} f'} \quad \text{(SEQPIPE)} \\
& \frac{g \text{ where } x :\in f \xrightarrow{l} g' \text{ where } x :\in f}{f \xrightarrow{l} f'} \quad \text{(ASYMLEFT)} \\
& \frac{g \text{ where } x :\in f \xrightarrow{l} g \text{ where } x :\in f'}{f \xrightarrow{1c} f'} \quad \text{(ASYMRIGHT)} \\
& \frac{g \text{ where } x :\in f \xrightarrow{\tau} g[c/x]}{g \xrightarrow{l} f'} \quad \text{(ASYPRETUNE)} \\
\end{align*}
\]

\[
\begin{align*}
& \frac{E(\bar{x}) \Delta f}{E(\bar{p}) \xrightarrow{\tau} f[\bar{p}/\bar{x}]} \quad \text{(DEF)} \\
& \frac{\text{let}(c) \xrightarrow{1c} 0}{\text{LET}} \\
\end{align*}
\]

\[
\begin{align*}
\text{Signal} \xrightarrow{l} 0 \quad \text{(SIGNAL)}
\end{align*}
\]

Some Strongly Bisimilar Expressions

\[
\begin{align*}
& f \mid 0 \sim f \\
& f \mid g \sim g \mid f \\
& (f \mid g) \mid h \sim f \mid (g \mid h) \\
& 0 > x > f \sim f \\
& f > x > (g > y > h) \sim (f > x > g) > y > h \quad \text{if } x \notin \text{fv}(h) \\
& f >> (g > y > h) \sim (f >> g) > y > h \quad \text{(corollary)} \\
& (f \mid g) > x > h \sim (f > x > h) \mid (g > x > h) \\
& 0 \text{ where } x :\in M(c) \sim M(c) >> 0 \\
& (f \mid g) \text{ where } x :\in h \sim (f \text{ where } x :\in h) \mid g \quad \text{if } x \notin \text{fv}(g) \\
& (f > y > g) \text{ where } x :\in h \sim (f \text{ where } x :\in h) > y > g \quad \text{if } x \notin \text{fv}(g) \\
& (f \text{ where } y :\in g) \text{ where } x :\in h \sim (f \text{ where } x :\in h) \text{ where } y :\in g \\
& \quad \text{if } x \notin \text{fv}(g) \text{ and } y \notin \text{fv}(h)
\end{align*}
\]
Some Examples - I

Let \( Rtimer(t) \) a site that responds with a signal after \( t \) units of time.

Time Out

\[
Epay(ccnumber) \triangleq \text{let}(x) \quad \text{where} \quad x \in \left( \begin{array}{l}
\text{CreditCheck}(ccnumber) \\
Rtimer(10) \gg \text{let}(false)
\end{array} \right)
\]

Priority

\[
\text{News}(d) \triangleq \text{let}(x) \quad \text{where} \quad x \in \left( \begin{array}{l}
\text{ANSA}(d) \\
(Rtimer(5) \gg \text{let}(y)) \quad \text{where} \quad y \in \text{CNN}(d)
\end{array} \right)
\]

Recursive Definitions

\[
\text{Metronome} \triangleq \text{Signal} \mid (Rtimer(1) \gg \text{Metronome})
\]

Some Examples - II

Fork-Join Parallelism and Synchronization

\[
\begin{align*}
\text{CityDate} & \triangleq \left( \text{let}(x, y) \quad \text{where} \quad x \in \text{GoogleLocate} \right) \quad \text{where} \quad y \in \text{GoogleDate} \\
\text{WForecast} & \triangleq \text{CityDate} > x > \text{CnnWeather}(x) \\
& \quad \text{where} \quad z \in \text{WForecast} \rightarrow z = 11^\circ\text{C}/22^\circ\text{C} - \text{PartiallyCloudy}
\end{align*}
\]

\[
\text{Sync}(\tilde{M}) \triangleq \text{let}(x_1) \gg \ldots \gg \text{let}(x_n) \gg \text{Signal} \\
\quad \text{where} \quad x_1 \in M_1 \\
& \quad \ldots \\
& \quad \text{where} \quad x_n \in M_n
\]

\((M_1, \ldots, M_n)\) are executed in parallel, but the signal is emitted only after having the response from every \( M_i \).

\[
\begin{align*}
\text{SyncList}(F, [\]) & \triangleq \text{Signal} \\
\text{SyncList}(F, a :: as) & \triangleq \text{Sync}(F(a), \text{SyncList}(F, as))
\end{align*}
\]
Some Examples - III

Let $f(b)$ a site that responds with a signal if $b$ is \texttt{true} and it remains silent if $b$ is \texttt{false}.

Choices

\[
\begin{align*}
\text{Cond}(b, S, T) & \overset{\Delta}{=} (f(b) \gg S) \lor (f(\neg b) \gg T) \\
A.P + B.Q & \overset{\Delta}{=} \text{Cond}(b, P, Q) \text{ where } b \in \left( \begin{array}{c} A \gg \text{let(true)} \\ B \gg \text{let(false)} \end{array} \right)
\end{align*}
\]

Some Examples - VI

Best Fare Search

Suppose you want to book a trip by contacting your two favourite companies (they might not respond).

- Any offer less than a threshold $c$ is fine with you.
- If both offers are greater than $c$, then the less expensive is also fine.

Let $LT(a, b)$ a site that responds with $a$ if it is below $b$, otherwise it remains silent.
Let $Min(a, b)$ a site that responds with the minimum between $a$ and $b$.

\[
\begin{align*}
\text{Trip}(c, \text{AirA}, \text{AirB}) & \overset{\Delta}{=} \text{let}(z) \text{ where } z \in LT(x, c) \lor LT(y, c) \lor Min(x, y) \\
&\text{where } x \in \text{AirA} \\
&\text{where } y \in \text{AirB}
\end{align*}
\]
Service-oriented computing

**Features**

*Service-oriented computing* is an emerging paradigm where services are understood as:

- autonomous
- platform-independent

computational entities that can be:

- described
- published
- categorised
- discovered
- dynamically assembled for developing massively distributed, interoperable, evolvable systems.

**Widespread success**

Many large companies invested a lot of efforts and resources to promote service delivery on a variety of computing platforms, mostly through the Internet in the form of Web services.

**Expectations**

Tomorrow, there will be a plethora of new services as required for e-government, e-business, and e-science, and other areas within the rapidly evolving Information Society.
 SENSORIA

Software Engineering for Service-Oriented Overlay Computers

SENSORIA (http://sensoria.fast.de) is an IST-FET project funded by the European Union as an Integrated Project in the 6th framework programme as part of the Global Computing Initiative.

Aim

Developing a novel, comprehensive approach to the engineering of software systems for service-oriented overlay computers.

Industrial consortia are developing orchestration languages, targeting the standardization of Web services and XML-centric technologies, but they lack clear semantic foundations!

Strategy

Integration of foundational theories, techniques, methods and tools in a pragmatic software engineering approach.

A General Theory of Services

The role of process calculi

A crucial role in the project will be played by formalisms for service description that lay the mathematical basis for analysing and experimenting with components interactions, and for combining services.

Core calculus

We seek for a small set of primitives that might serve as a basis for formalizing and programming service oriented applications over global computers.

Features

As an outcome of an initial study pursued during the first months of SENSORIA, we propose a process calculus that features explicit notions of service definition, service invocation and session handling.
Sources of inspiration

- $\pi$-calculus (naming primitives)
- web$\pi$, cjoin, Sagas: primitives for transactions and compensations
- Orc, (sequencing and asymmetric parallel composition)

Summarizing, SCC combines the service oriented flavour of Orc with the name passing communication mechanism of the $\pi$-calculus.

What’s new?

The main novelty is the session handling mechanisms for the definition of

- session naming and scoping
- structured interaction protocols (bi-directional and more complex than the simple one-way and request-response provided by Orc)
- service interruption, cancelation and update (dynamic environment)

Service Centered Calculus - II

Why not $\pi$?

A small set of well-disciplined, higher-level primitives can favour and make more scalable the development of typing systems and proof techniques centered around the notion of service and sessions, for ensuring, e.g., compatibility of client and service behaviour, or the absence of deadlock in service composition.

Roadmap

The formal presentation of SCC involves some key notational and technical solutions. For this reason, we will give a gentle, step-by-step presentation of the various ingredients:

- first a persistent fragment (called PSC), then full SCC
- syntax and (reduction-based) operational semantics
- a number of programming samples that demonstrate flexibility of the chosen set of primitives
- a few encodings that relates our proposal with existing ones
Service Definition

Service definition

\[ s \Rightarrow (x)P \]

- \( s \) is the service name
- \( x \) is the formal parameter
- \( P \) is the actual implementation of the service.

Example: Successor

\[ \text{succ} \Rightarrow (x)x + 1 \]

Received an integer returns its successor.

Service Invocation

Service Invocation

\[ s\{(x)P\} \leftarrow Q \]

- each new value \( v \) produced by the client \( Q \) will trigger a new invocation of service \( s \) (like Orc sequencing)
- for each invocation, a suitable instance \( P\{v/x\} \) of the process \( P \), with \( x \) bound to the actual invocation value \( v \), implements the client-side protocol for interacting with the new instance of the service \( s \)

Example: A client for the successor

\[ \text{succ}\{(x)(y)\text{return } y\} \leftarrow 5 \]

(in this case, the client side makes no use of the formal parameter)
Service Activation

A service invocation causes activation of a new session:
- dual fresh identifiers, \( r \) and \( \overline{r} \), name the two sides of the session
- client and service protocols run each at the proper side of the session

Example: Successor

The invocation of service successor triggers the session

\[
(\nu r) \left( \ldots \overline{r} \triangleright 5 + 1 \ldots \right) \left\{ \ldots r \triangleright (y) \nu \text{return } y \ldots \right. 
\]

The client waits for a value from the server (6) to be substituted for \( y \)

Session Communication

Within sessions, communication is bi-directional, in the sense that the interacting protocols can exchange data in both directions.

After the value 6 has been communicated:

\[
(\nu r) \left( \ldots \overline{r} \triangleright 0 \ldots \right) \left\{ \ldots r \triangleright \text{return } 6 \ldots \right. 
\]

Session Returning Values

Values can be returned outside the session to the enclosing environment and used for invoking other services.

\[
(\nu r) \left( \ldots \overline{r} \triangleright 0 \ldots \right) \left\{ \ldots 6 \mid r \triangleright 0 \ldots \right. 
\]
Session Termination

Printing values
A client invokes the service \texttt{succ} and then prints the result:

\[
\text{print}\{(z)0\} \leftarrow (\text{succ}\{(x)(y)(\text{return}\ y) \leftarrow 5)\}.
\]

(in this case, the service \texttt{print} is invoked with vacuous protocol \texttt{(z)0}).

Handling Interruption
A protocol (on both sides of a session) can be interrupted (e.g. due to the occurrence of an unexpected event), and interruption can be notified to a suitable handler at the partner site.

Printing values with faulty printers
Below, a suitable service \texttt{fault} handles printer failures:

\[
\text{print}\{(z)0\} \leftarrow_{\text{fault}} (\text{succ}\{(x)(y)(\text{return}\ y) \leftarrow 5)\}.
\]

Persistent Session Calculus

We start by presenting the close-free fragment of SCC
We call it PSC for persistent session calculus:

- sessions can be established
- a session can be garbage collected when the protocol has run entirely,
- but sessions can neither be aborted nor closed by one of the parties

A note on well-formedness
A process is well-formed if (assuming by $\alpha$-conversion that all its bound names are different from each other and from the free names):

- each session name $r$ occurs only once ($r \triangleright 0$ is immaterial)
- but it is allowed to have both sessions $r \triangleright Q$ and $\overline{r} \triangleright Q'$.

The use of dual names is not strictly necessary, but we prefer to keep this distinction to make evident that once the protocol is started there might still be some reasons for distinguishing the two side ends (e.g., types).
PSC Syntax

We presuppose a countable set \( \mathcal{N} \) of names \( a, b, c, \ldots, r, s, \ldots, x, y, \ldots \). A bijection \( \tau \) on \( \mathcal{N} \) is presupposed s.t. \( \overline{a} = a \) for each name \( a \).

\( a \) and \( \overline{a} \) denote just dual names for communicating in both directions.

\[
P, Q ::= \begin{align*}
0 & \quad \text{Nil} \\
| a.P & \quad \text{Concretion (pass} \ a \text{ to session partner)} \\
| (x)P & \quad \text{Abstraction (take from session partner)} \\
| \text{return } a.P & \quad \text{Return Value (out of current session)} \\
| s \Rightarrow (x)P & \quad \text{Service Definition} \\
| s\{x\}P & \Leftarrow Q \quad \text{Service Invocation} \\
| r \triangleright P & \quad \text{Session Side} \\
| P \mid Q & \quad \text{Parallel Composition} \\
| (\nu a)P & \quad \text{New Name}
\end{align*}
\]

(operators are listed in decreasing order of precedence)

Conventions

- Free occurrences of \( x \) in \( P \) (including \( \overline{x} \)) are bound in \((\nu x)P\) and \((x)P\).
- Capture-avoiding substitution of the free occurrences of \( x \) with \( \nu \) (and of \( \overline{x} \) with \( \overline{\nu} \)) in \( P \) is denoted \( P\{\nu/x\} \).
- We identify processes up to alpha-equivalence.
- We omit trailing \( 0 \), writing e.g. \( \nu \) instead of \( \nu.0 \).
- Restriction and parallel composition have the usual meaning.
- Service definitions are persistent, i.e., each new invocation is served by a fresh instance of the protocol (process calculists may think of an implicit replication prefixing each service definition.)
- A service invocation \( s\{(x)P'\} \Leftarrow Q \) invokes \( s \) for any concretion (value) \( u \) produced by the execution of \( Q \). The process \((x)P'\) is the client-side protocol for interacting with (the instance of) \( s \).
- Sessions, service definitions and service invocations can be nested at arbitrary depth; in an interaction it is however the innermost service or session name that counts.
PSC Structural Congruence

Axioms

\[(P \mid Q) \mid R \equiv P \mid (Q \mid R)\]
\[P \mid Q \equiv Q \mid P\]
\[P \mid 0 \equiv P\]

\[(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P\]
\[(\nu x)0 \equiv 0\]

\[P \mid (\nu x)Q \equiv (\nu x)(P \mid Q)\] if \(x \notin \text{fn}(P)\)

\[r \triangleright (\nu x)P \equiv (\nu x)(r \triangleright P)\] if \(x \notin \{r, \overline{r}\}\)

\[s\{x\}P \Leftarrow (\nu y)Q \equiv (\nu y)(s\{x\}P \Leftarrow Q)\] if \(y \notin \text{fn}(s\{x\}P) \cup \{r, \overline{r}\}\)

\[r \triangleright 0 \equiv 0\]

PSC Operational Semantics

\[\mathbb{C}[s \Rightarrow (x)P] \mid \mathbb{D}[s\{y\}P' \Leftarrow (Q|u.R)] \rightarrow (\nu r)\left(\mathbb{C}[r \triangleright P\{u/x\} \mid s \Rightarrow (x)P] \mid \mathbb{D}[\overline{r} \triangleright P'\{u/x\} \mid s\{y\}P' \Leftarrow (Q|R)]\right)\]
if \(r\) is fresh and \(u, s\) not bound by \(\mathbb{C}, \mathbb{D}\)

\[\mathbb{C}[r \triangleright (P|u.Q)] \mid \mathbb{D}[\overline{r} \triangleright (R|z)S] \rightarrow \mathbb{C}[r \triangleright (P|Q)] \mid \mathbb{D}[\overline{r} \triangleright (R|S\{u/z\})]\]
if \(u\) not bound by \(\mathbb{C}, \mathbb{D}\)

\[a \triangleright (P|\text{return } b.Q) \rightarrow b \triangleright a \triangleright (P|Q)\]

\[\mathbb{C}[P] \rightarrow \mathbb{C}[P']\] if \(P \equiv Q, Q \rightarrow Q', Q' \equiv P'\)

Active Contexts

The reduction rules exploit active contexts \(\mathbb{C}, \mathbb{D}\) to specify where the interacting processes are located. An active context is a process with a hole \([\_]\) in an active position.

\[\mathbb{C}, \mathbb{D} ::= \_ | \mathbb{C}|P | a\{x\}P \Leftarrow \mathbb{C} | a \triangleright \mathbb{C} | (\nu a)\mathbb{C}\]

With \(\mathbb{C}[P]\) we denote the process obtained by filling the hole in \(\mathbb{C}\) with \(P\).
PSC Examples: Functional flavour

A common pattern of service invocation is:

\[ s\{(x)(y)\text{return }y\} \leftarrow P \]

Shorthand Notation

\[ s \leftarrow P \]

We write

\[ g \leftarrow (f \leftarrow v) \]

(or simply \( g \leftarrow f \leftarrow v \), stipulating that \( \leftarrow \) is right-associative) instead of something like

\[ g\{(z)(w)\text{return }w\} \leftarrow (f\{(x)(y)\text{return }y\} \leftarrow v). \]

PSC Examples: Pairing service

For simplicity, we shall use polyadic notation, like \( \langle v_1, \ldots, v_n \rangle.P \) and \( (a_1, \ldots, a_n)P \).

Pairing service

\[ pair \Rightarrow (z)(x)(y)\langle x, y \rangle \]

Shorthand

Binding occurrences of names that are not subsequently used (like \( z \) above) are written in the following with \(-\). Also, we presuppose a distinct name \( \langle \rangle \) to be used as a unit value.

\( P \) and \( Q \) give results to be paired. The pair produced by the service is bound to \( p \) and returned as the result.

\[ pair\{(-)(P|Q|(p)\text{return }p)\} \leftarrow \bullet \]
PSC Examples: Blind invocation

**Vacuous protocol**
If no reply is expected from a service, the client can employ a vacuous protocol

\[ a\{(-)0\} \leftarrow P \]

**Shorthand**

\[ a\{} \leftarrow P \]

**Getting news by email**
Assume that there are the following available services:

- service `emailMe` that expects a value `msg` and then sends the message `msg` to your email address;
- services `ANSA`, `BBC` and `CNN` that return the latest news.

\[ emailMe\{} \leftarrow \text{pair}\{(-)(ANSA \leftarrow \bullet | BBC \leftarrow \bullet | CNN \leftarrow \bullet | (p) \text{return } p) \} \leftarrow \bullet \]

will send you the first two news collected from `ANSA`, `BBC` and `CNN`.

---

**PSC Examples: Conditionals**

**If Then Else**
Let us assume that the services

- `true \Rightarrow (x, y)x`
- `false \Rightarrow (x, y)y`

are available.

\[ if \Rightarrow (b, \text{then, else})(b\{(-)(c)(c \leftarrow \bullet)\} \leftarrow (\text{then, else}) \]

invokes either the service `then` or the service `else` depending on the value (that should be either `true` or `false`) of the first argument `b`. 
Clock (service side)

Service invocations can be nested recursively inside a service definition:

\[
\text{clock} \Rightarrow (-) \left( \begin{array}{l}
\text{return } \bullet \\
\text{clock} \{\} \leftarrow \bullet
\end{array} \right)
\]

defines a service that, when invoked with \( \text{clock} \{\} \leftarrow \bullet \), produces an infinite number of \( \bullet \) values on the service-side.

PSC Examples: Recursion - II

Clock (client side)

To produce the \( \bullet \) values on a specific location different from the service-side, the service to be invoked can be written as

\[
\text{clock}' \Rightarrow (s) \left( \begin{array}{l}
s \{\} \leftarrow \bullet \\
\text{clock}' \{\} \leftarrow s
\end{array} \right)
\]

and a local publishing service

\[
\text{pub} \Rightarrow (s) \text{return } s
\]

must be located where the \( \bullet \) is to be produced.
The name \( \text{pub} \) should be passed to \( \text{clock}' \) as argument:

\[
\text{clock}' \{\} \leftarrow \text{pub}
\]
PSC Examples: Recursion - III

Programming Pattern

The service \textit{pub} (or alike) can be useful in many applications, because it allows to publish values in the location where it is placed.

In fact, in PSC \textit{sessions cannot be closed} and therefore recursive invocations on the client-side are nested at increasing depth (while the return instruction can move values only one level up).

News Streaming (client side)

A recursive process that repeatedly invokes service \textit{s} on value \textit{x} with publishing service \textit{p} is shown below:

\[
rec' \Rightarrow (s, x, p)(s\{(-)(y)p\} \Leftarrow y | rec'\{\} \Leftarrow (s, x, p) \Leftarrow x).
\]

Sample of invocation of the service \textit{rec'}:

\[
rec'\{\} \Leftarrow (\text{ansa}, \bullet, id) \mid id \Rightarrow (x) \text{return } x
\]

that returns the stream of news obtained from the ANSA service.

PSC Examples: Pipeline and forwarder - I

Observation

The process seen at the end of the previous example produces an unbounded stream of values.

Question

Is it possible to deploy some sort of pipeline between two services \textit{p} and \textit{q} in such a way that \textit{q} is invoked for each value produced by \textit{p}?

A note

If \textit{P} is a process that produces a stream of values then the composition \textit{q \Leftarrow P} already achieves the aim.

How to pipeline?

Thus to compose \textit{p} and \textit{q} in a pipeline it suffices to design a client-side protocol for collecting all the values returned by \textit{p}...
PSC Examples: Pipeline and forwarder - II

**Trivial Recursion does not work!**

One might think to exploit recursion to deploy local receivers of the form (x) return x, but the implicit nesting of sessions would cause all such receivers to collect values only from different sessions than the original one.

**No Replicator!**

Extending the syntax with π-calculus like replicator !P:

\[ \text{pipe} = (\neg)!x \text{return } x \]

**No Code Passing!**

Extending the syntax with return \( P.Q \), whose semantics is:

\[ r \triangleright (R|\text{return } P.Q) \rightarrow P | r \triangleright (R|Q) \]

Replication can then be coded as follows:

\[ !P = (\nu \text{rec})( \text{rec} \Rightarrow (\neg)(\text{return } P | \text{rec}{} \Leftarrow \bullet) | \text{rec}{} \Leftarrow \bullet) \]

PSC Examples: Pipeline and forwarder - III

**We need a publisher!**

Without extending the syntax of the calculus, a solution is to exploit a publishing service like \( pub \) above, which must be passed to \( p \) (and properly used therein).

**Conference Announcements**

For instance, if \( EATCS \) and \( EAPLS \) return streams of conference announcements on the received service name, then

\[
\text{emailMe}{} \Leftarrow \begin{cases} 
\text{pub} \Rightarrow (s)\text{return } s \\
\text{EATCS}{} \Leftarrow \text{pub} \\
\text{EAPLS}{} \Leftarrow \text{pub}
\end{cases}
\]

will send you all the announcements collected from \( EATCS \) and \( EAPLS \). More concisely, this can be equivalently written as

\[
\text{EATCS}{} \Leftarrow \text{emailMe} | \text{EAPLS}{} \Leftarrow \text{emailMe}.
\]
PSC Examples: Encoding of the lazy \( \lambda \)-calculus

The translation is in the spirit of Milner’s \( \pi \)-calculus encoding:

\[
\begin{align*}
[x]_p &= x\{ \} \leftarrow p \\
[\lambda x. M]_p &= p \Rightarrow (x)([M]_q) \\
[M N]_p &= (\nu m)(\nu n) \left( \begin{array}{c} [M]_m \\
\text{if } n \Rightarrow (s)[N]_n \\
\text{then } m\{(-)p \} \leftarrow n \end{array} \right)
\end{align*}
\]

The more important differences

- Each service invocation opens a new session where the computation can progress (remind that sessions cannot be closed in PSC)
- All service definitions will remain available even when no further invocation will be possible.

If on one hand, the encoding witnesses the expressive power of PSC, on the other hand, it also motivates the introduction of some mechanism for closing sessions.

Encoding of PSC into \( \pi \)-calculus

The encoding below shows that PSC can be seen as a disciplined fragment of the \( \pi \)-calculus.

\[
\begin{align*}
[a \{ (x) P \} \leftarrow Q]_{in, out, ret} &= (\nu z) (Q)_{in, z, ret} \mid !z(x). (\nu r, \tilde{r}) a[r, \tilde{r}, x]. [P]_{r, \tilde{r}, out} \\
[a \Rightarrow (x) P]_{in, out, ret} &= !a(r, \tilde{r}, x). ([P]_{\tilde{r}, out} \\
[a \triangleright P]_{in, out, ret} &= [P]_{a, \tilde{a}, out} \\
[a . P]_{in, out, ret} &= \text{out } a . [P]_{in, out, ret} \\
[(x) P]_{in, out, ret} &= \text{in}(x) . [P]_{in, out, ret} \\
[\text{return } a . P]_{in, out, ret} &= \text{return } a . [P]_{in, out, ret} \\
[P | Q]_{in, out, ret} &= [P]_{in, out, ret} \mid [Q]_{in, out, ret} \\
[\nu x P]_{in, out, ret} &= (\nu x) [P]_{in, out, ret} \\
[0]_{in, out, ret} &= 0
\end{align*}
\]

The encoding can hardly be extended to full SCC calculus due to the session interruption mechanism that has no direct counterpart in the \( \pi \)-calculus.
From PSC to SCC - I

Even though PSC is expressive enough to model service definitions and invocations, it does not provide operators for explicit closing of sessions.

Once the two protocols $r \triangleright P_1$ at client-side and $\bar{r} \triangleright P_2$ at service-side are activated, the session $r$ (resp. $\bar{r}$) is garbage collected by the structural congruence only if the protocol $P_1$ (resp. $P_2$) reduces to $0$.

Many sessions can never reduce to $0$, e.g., those containing service definitions!

Also, one may want to explicit program session termination, for instance in order to implement cancellation workflow patterns or Orc’s asymmetric parallel or to manage abnormal events.

From PSC to SCC - II

The full SCC calculus comprises a mechanism for closing sessions that can be roughly described as follows:

- a service name, identifying the so-called termination handler service, is associated to each session
- the first time the protocol running inside the session invokes such a service, the session is closed

**termination handler service**

The name $k$ of the termination handler service is indicated as a subscript:

$$r \triangleright_k P$$

In case $P$ contains an invocation to $k$, like $k\{(x)P\}' \leftrightarrow (Q|v.R)$, the overall session $r$ may be closed.
From PSC to SCC - III

The termination handler service is associated to sessions on their instantiation. The intuition that we follow is that the termination of the session on one side, should be communicated to the opposite side.

A slight asymmetry

- The syntax of clients becomes: \( a\{ (x)P \} \leftarrow_k Q \)
  (we added the name \( k \) of the termination handler service to be associated to the session instantiated on the service-side)
- Services are now specified with the process \( a \Rightarrow (x)P : (y)T \)
  (an additional protocol \( (y)T \) is specified which represents the body of a fresh termination handler service that will be associated to the corresponding session on the client-side).

Shorthand

In service definition we write \( a \Rightarrow (x)P \) for \( a \Rightarrow (x)P : (y)0 \). We also omit \( k \) in \( a\{ (x)P \} \leftarrow_k Q \) and \( a \leftarrow_k Q \) when it is not relevant.

An Abstract Example

To gain some familiarity with the extended service invocation mechanism, consider the following process composed (from left to right) by

- a termination handler service \( k \),
- a client willing to invoke service \( a \) with value \( v \),
- and the definition of the service \( a \)
  \[ k \Rightarrow (x)S \mid a\{ (x)P \} \leftarrow_k (v|Q) \mid a \Rightarrow (x)P' : (y)T \]

Now, a fresh service name \( k' \) is associated to the newly installed termination handler service specified on the service-side. Thus, the freshly activated processes will look like:

\[ (vr, k')(\bar{\tau} \triangleright_k P\{^v_{/x}\}\{^k'_{/\text{close}}\} \mid r \triangleright_k P'\{^v_{/x}\}\{^k_{/\text{close}}\} \mid k' \Rightarrow (y)T) \]

Note that the session on the client-side has associated the name \( k' \), while the session on the service-side has associated the name \( k \).
SCC Syntax

We presuppose a countable set \( \mathcal{N} \) of names \( a, b, c, \ldots, r, s, \ldots, x, y, \ldots \). A bijection \( \tau \) on \( \mathcal{N} \) is presupposed s.t. \( \overline{a} = a \) for each name \( a \).

A special name close is reserved for the specification of session protocols.

\[
P, Q, T, \ldots := \quad \begin{array}{ll}
0 & \text{Nil} \\
| a.P & \text{Concretion} \\
| (x)P & \text{Abstraction} \\
| \text{return } a.P & \text{Return Value} \\
| a \Rightarrow (x)P : (y)T & \text{Service Definition} \\
| a\{(x)P\} \leftarrow_k Q & \text{Service Invocation} \\
| a \triangleright_k P & \text{Session} \\
| P | Q & \text{Parallel Composition} \\
| (\nu a)P & \text{New Name}
\end{array}
\]

(operators are listed in decreasing order of precedence)

SCC Operational Semantics - I

Structural Congruence and Active Contexts

As before (but over the extended syntax).

Termination names

An auxiliary function \( tn \) is defined on active contexts that keeps track of the \( \text{termination names} \) associated to sessions that enclose the hole:

\[
\begin{align*}
\text{tn}([\_]) &= \emptyset \\
\text{tn}(\{ s \}) &= \text{tn}(C) \cup \{ s \} \\
\text{tn}(a \triangleright_s C) &= \text{tn}(C) \setminus \{ a \} \\
\text{tn}(C | P) &= \text{tn}(a\{(x)P\} \leftarrow_s C) = \text{tn}(C)
\end{align*}
\]

This function is used to check whether a service invocation should be interpreted as a closing signal for some of the enclosing sessions.
SCC Operational Semantics - II

\[
\begin{align*}
\mathbb{C}[ s \Rightarrow (x)P : (z)T ] & | \\
\mathbb{D}[ s \{ (y)P' \} \leftarrow_k (Q | u.R) ] & \Rightarrow (\nu r, k') \\
& \left( \\
\begin{array}{c}
\mathbb{C} \left[ s \Rightarrow (x)P : (z)T \right] \mid \\
k' \Rightarrow (z)T \mid \\
r \triangleright_k P\{u/x\}\{}^{k'}_{\text{close}} \mid \\
\mathbb{D} \left[ T \triangleright_{k'} P\{u/y\}\{}^{k'}_{\text{close}} \mid \\
\mathbb{D}[ s \{ (y)P' \} \leftarrow_k (Q | R) ]
\end{array}
\right)
\end{align*}
\]

if \( s \notin \text{tn}(\mathbb{D}) \), \( r, k' \) are fresh and \( u, s, k \) not bound by \( \mathbb{C}, \mathbb{D} \)

\[
r \triangleright_s \mathbb{D}[ s \{ (y)P \} \leftarrow_k (Q | u.R) ] \Rightarrow s\{\} \leftarrow u
\]

if \( s \notin \text{tn}(\mathbb{D}) \) and \( u \) not bound by \( \mathbb{D} \)

\[
\begin{align*}
\mathbb{C}[ r \triangleright_k (P | u.Q) ] & | \\
\mathbb{D}[ T \triangleright_{k'} (R \{ (z)S \} ] & \Rightarrow \mathbb{C}[ r \triangleright_k (P | Q) ] | \\
\mathbb{D}[ T \triangleright_{k'} (S\{u/z\} | R) ] & \\
& \text{if } u \text{ not bound by } \mathbb{C}, \mathbb{D}
\end{align*}
\]

\[
a \triangleright_k (P | \text{return } b.Q) \Rightarrow b \mid a \triangleright_k (P | Q)
\]

\[
\mathbb{C}[ P ] \Rightarrow \mathbb{C}[ P' ] \text{ if } P \equiv Q, \; Q \rightarrow Q', \; Q' \equiv P'
\]

SCC Examples: Service Update

Soccer World Cup

\textit{soccerWorldChampion} \Rightarrow (¬)brasil

returns the name of the last winner of the soccer world championship.

When a different team becomes the new world champion then the service must be updated!

In PSC there is no way to cancel a definition and replace it with a new one. By contrast, in SCC we can exploit session closing:

\[
r \triangleright_{\text{new}} \left( \begin{array}{c}
\text{soccerWorldChampion} \Rightarrow (¬)brasil | \text{update} \Rightarrow (y)(\text{new}\{} \leftarrow y \) \\
\text{new} \Rightarrow (z)( \text{soccerWorldChampion} \Rightarrow (¬)z | \text{update} \Rightarrow (y)(\text{new}\{} \leftarrow y \)
\end{array} \right)
\]

The service \textit{update}, when invoked with a new name \( z \), permits to cancel the currently available service \textit{soccerWorldChampion} and replace it with a new instance that returns the name \( z \).
SCC Examples: Closure Protocol

A typical usage of termination handler services is to program them to close the current session and then include their definition in the session protocol.

Consider the process

\[ \text{close } \{ \} \leftarrow (end \Rightarrow (x)\text{return } x) \]

If included in the client-side protocol, it will handle service-side termination by closing the client-side session:

\[ (\nu \text{ end}) s\{ (y) (P | \text{close } \{ \} \leftarrow (end \Rightarrow (x)\text{return } x)) \} \leftarrow \text{end } \nu \]

Note that the closing of the client-side session will in turn activate the termination handler on the service-side.

SCC Examples: A blog service - I

Blog

We consider a service that implements a blog, i.e. a web page used by a web client to log personal annotations.

Interaction with the Blog

A blog provides two services:

- \textit{get} to read the current contents of the blog
- \textit{set} to modify the contents.

The close-free fragment is not expressive enough to faithfully model such a service because it does not support service update, here needed to update the blog contents.
SCC Examples: A blog service - II

Blog Factory

\[
\text{newBlog} \Rightarrow (v, \text{get}, \text{set})(\text{blog} \{\} \leftarrow_{\text{newBlog}} (v, \text{get}, \text{set})) \mid \\
\text{blog} \Rightarrow (v, \text{get}, \text{set})( \text{get} \Rightarrow (-)v \mid \\
\text{close} \{\} \leftarrow (\text{set} \Rightarrow (v')\text{return} (v', \text{get}, \text{set})) )
\]

We use the service \text{newBlog} as a factory of blogs. It receives three names:

- the initial contents \(v\)
- the name for the new \text{get} service
- the name for the new \text{set} service

Upon invocation, the factory forwards the three received values to the \text{blog} service which is the responsible for the actual instantiation of the \text{get} and \text{set} services.

The update of the blog contents is achieved by invoking the service \text{close} which is bound to \text{newBlog}; this invocation cancels the currently available \text{get} and \text{set} services and delegates to \text{newBlog} the creation of their new instances passing also the updated contents \(v'\).

SCC Examples: A blog service - III

Blog update

The process below installs a wiki page with initial contents \(v\), then it adds some new contents \(v'\).

\[
\text{newBlog} \{\} \leftarrow (v, \text{get}, \text{set}) \mid \\
\text{set} \{\} \leftarrow (\text{update}\{ \text{get} \leftarrow \bullet | v' \leftarrow \bullet \})
\]

The service \text{update} simply computes the new contents appending \(v'\) to the contents \(v\) received after service invocation:

\[
\text{update} \Rightarrow (-)(x)(y).x \circ y
\]

Here \(\circ\) denotes justaposition of blog contents.
SCC Examples: Encoding Orc in SCC - I

SCC as a service orchestration language
To evaluate the expressiveness and usability of SCC as a language for service orchestration, one has to challenge its ability of encoding some frequently used service composition patterns.

Workflow patterns
A library of basic patterns, called the workflow patterns, has been identified by van der Aalst et al.

It has been shown that Orc can conveniently model most workflow patterns.

The Orc challenge
If we can show that SCC can encode Orc, then by transitivity we can implement van der Aalst’s workflow patterns.

SCC Examples: Encoding Orc in SCC - II

While a value is trivially encoded as itself, i.e., \([u] = u\), for variables (and thus for actual parameters) we need two different encodings, depending on whether they are passed by name or evaluated.

We distinguish the two encodings by different subscripts:

\([x]_n = x \quad [x]_v = x \leftarrow \langle \rangle\)

- The evaluation of a variable \(x\) is encoded as a request for the current value to the variable manager of \(x\).
- Variable managers are created by both sequential composition and asymmetric parallel composition.
SCC Examples: Encoding Orc in SCC - II

\[
\begin{align*}
[E(x) \triangleq P] & \quad = \quad E \Rightarrow (x)[P] \\
[a(p)] & \quad = \quad a \leftarrow [p],_v \\
[x(p)] & \quad = \quad (\nu \text{forw}, \text{pub})( \text{forw} \{ \} \leftarrow [x],_v | \\
& \quad \quad \quad \text{forw} \Rightarrow (a)\text{pub} \{ \} \leftarrow [a(p)],_v | \\
& \quad \quad \quad \text{pub} \Rightarrow (y)\text{return } y ) \\
[E(p)] & \quad = \quad E \leftarrow [p],_n \\
[P|Q] & \quad = \quad [P][Q] \\
[P > x > Q] & \quad = \quad (\nu z, \text{pub})( \ z \{ \} \leftarrow [P],_v | \\
& \quad \quad \quad z \Rightarrow (y)(\nu x)(x \Rightarrow (-)y | \ \text{pub} \{ \} \leftarrow [Q],_v | \\
& \quad \quad \quad \text{pub} \Rightarrow (y)\text{return } y ) \\
[Q \ \text{where } x : \in P] & \quad = \quad (\nu x, z, s)( [Q] | (z \Rightarrow (y)(x \Rightarrow (-)y)) | \\
& \quad \quad \quad (s \{ \} \leftarrow z \{ \} ) | s \Rightarrow (-)(\text{close} \{ \} \leftarrow [P],_v ) )
\end{align*}
\]

From Orc to SCC: An Example

Emailing news in Orc

Let us consider the Orc expression

\[
CNN(d)|BBC(d) > x > email(x)
\]

which invokes the news services of both CNN and BBC asking for news of day \(d\). For each reply it sends an email (to a default address) with the received news. Thus this expression can send from zero up to two emails.

The SCC encoding is as follows:

\[
(\nu z, \text{pub})(z \{ \} \leftarrow (CNN \leftarrow d|BBC \leftarrow d) | \\
\quad z \Rightarrow (y)(\nu x)(x \Rightarrow (-)y | \ \text{pub} \{ \} \leftarrow email \leftarrow x \leftarrow \{ \} ) | \\
\quad \text{pub} \Rightarrow (y)\text{return } y )
\]

We have supposed here to have CNN, BBC and email available as services.
Alternatives and Future Extensions

- Multi-party sessioning
- Replicator / recursion / return $P$
- Distribution
- Types
- Delegation
- Long-running transactions and compensations
- Synchronized termination
- XML querying
- SLA and QoS