Pi Calculus

The $\pi$ calculus (Milner et al., 1992) is the main representative of a family of process algebra-based models, which has its roots in earlier models of concurrency such as:

- Communicating Sequential Processes (CSP) (Hoare, 1985)
- Petri Nets (Petri, 1962)
Pi Calculus

Its key feature is *mobility* of processes (Milner’s Turing Award lecture 1993).

- Mobility is represented as dynamic reconfiguration of the topology of communicating processes.
- Inspired by the actor model (Hewitt 1973) to more closely model practical distributed and mobile systems.
- Process algebras have been used as theoretical frameworks to reason about concurrency
  - Introduced techniques such as *bisimulation*, in order to study equivalence of concurrent programs.
# Theory and Practice

“In theory, theory and practice are highly related. In practice,...”

<table>
<thead>
<tr>
<th>Model</th>
<th>Paradigm(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$ calculus</td>
<td>Functional Programming</td>
</tr>
<tr>
<td>First order logic</td>
<td>Logic Programming</td>
</tr>
<tr>
<td>$\pi$ calculus / Actors</td>
<td>Concurrent Programming</td>
</tr>
</tbody>
</table>

See also, “A Theory of Objects” (Abadi and Cardelli, 1996).
A simple interaction between two processes over a channel $b$ can be modeled as follows in the $\pi$ calculus:

$$\overline{ba}.S \mid b(c)\overline{cd}.P$$
Π Calculus Example

After the interaction, the printer process has “moved” from the server to the client:

\[
\bar{b}a.S \mid b(c).\bar{c}d.P \\
\xrightarrow{\tau} S \mid \bar{c}d.P\{a/c\} \\
\equiv S \mid \bar{a}d.P
\]
Π Calculus Notation

Channels / Names

- $a, b, c, \ldots$

Processes / Agents

- $P, Q, R, \ldots$

e.g.:

\[
\overline{ba}.S | b(c).\overline{cd}.P \\
\xrightarrow{\tau} S | \overline{cd}.P\{a/c\} \\
\equiv S | \overline{ad}.P
\]
### \( \Pi \) Calculus Syntax

\begin{align*}
\alpha & ::= \overline{c}x & \text{Write } x \text{ on channel } c \\
& ::= c(x) & \text{Read } x \text{ on channel } c \\
& ::= \tau & \text{No interaction} \\
P, Q & ::= 0 & \text{Empty process} \\
& ::= \alpha.P & \text{Prefixed process} \\
& ::= P | Q & \text{Concurrent composition of } P \text{ and } Q \\
& ::= P + Q & \text{Nondeterministic choice of } P \text{ or } Q \\
& ::= (\nu c)P & \text{New channel } c, \text{ scope restricted to } P \\
\text{if } x = y \text{ then } P & & \text{Conditional execution (match)} \\
\text{if } x \neq y \text{ then } P & & \text{Conditional execution (mismatch)} \\
A(y_1, \ldots, y_n) & & \text{Process invocation} \\
\Delta & ::= A(x_1, \ldots, x_n) \triangleq P & \text{Process declaration}
\end{align*}
Free and Bound Variable Occurrences

It is important to understand the notion of scope of variables, since the same channel name, e.g., \( x \), may refer to different channels if it appears in different scopes.

\[
\begin{align*}
\text{bind } x \text{ in } P. \\
(\nu x)P \\
\{ \text{does not bind } x \text{ in } P. \\
\end{align*}
\]
Free and Bound Variable Occurrences

For example, in the expression:

\[ c(x) \cdot d x . P \]

the variable \( x \) refers to the same channel, that whose name is read over channel \( c \) and subsequently written over channel \( d \).

On the other hand, in the expression:

\[ \bar{c} x . d(x) . P \]

the two occurrences of variable \( x \) are in two different scopes:

- the first \( x \) refers to a channel whose name is to be written over channel \( c \),
- and the second \( x \) is creating a new scope to represent a channel whose name is read over channel \( d \) and potentially subsequently used in \( P \).
Exercise 3.6.1

Define the sets of free variable names (fn) and bound variable names (bn) of the following expressions in terms of $a$, $x$, fn($P$) and bn($P$).

1. $\text{fn}(a(x).P) =$

2. $\text{fn}((\nu x)P) =$

3. $\text{fn}({\bar a}x.P) =$

4. $\text{bn}(a(x).P) =$

5. $\text{bn}((\nu x)P) =$

6. $\text{bn}({\bar a}x.P) =$
Structural Congruence

$P$ and $Q$ are structurally congruent, $P \equiv Q$, in any of the following cases:

1. $P$ and $Q$ are variants of $\alpha$-conversion.

2. $P$ and $Q$ are related by the Abelian monoid laws for $|$ and $+$:

   \[
   P|Q \equiv Q|P \quad P+Q \equiv Q+P \\
   (P|Q)|R \equiv P|(Q|R) \quad (P+Q)+R \equiv P+(Q+R) \\
   P|0 \equiv P \quad P+P \equiv P
   \]

3. $P$ and $Q$ are related by the unfolding law:

   \[A(\overline{y}) \equiv P\{\overline{y}/\overline{x}\} \quad \text{if} \quad A(\overline{x}) \triangleq P\]
Structural Congruence

$P$ and $Q$ are structurally congruent, $P \equiv Q$, in any of the following cases:

4. $P$ and $Q$ are related by the scope extension laws:

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

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

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

\[
(\nu x)\text{if } u = v \text{ then } P \equiv \text{if } u = v \text{ then } (\nu x)P \quad \text{if } x \notin \{u, v\}.
\]

\[
(\nu x)\text{if } u \neq v \text{ then } P \equiv \text{if } u \neq v \text{ then } (\nu x)P \quad \text{if } x \notin \{u, v\}.
\]

\[
(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P
\]
Example of Second Scope Extension Law

The second scope extension law is important (when applied from right to left) because it allows to extend the scope of a variable $x$ from a single process $Q$ to the concurrent composition $P \parallel q$, provided it does not capture free occurrences of $x$ in $P$.

For example, consider the expression:

$$a(y).P \parallel (\nu b)\tilde{a}b.Q$$

(1)

Assuming that $b \notin \text{fn}(P) \cup \{a\}$, we can apply scope extension to make it structurally congruent to:

$$(\nu b)(a(y).P \parallel \tilde{a}b.Q)$$

The latter expression can evolve by communication of the concurrent embedded processes over channel $a$ into:

$$(\nu b)(P\{b/y\} \parallel Q)$$
Exercise

Reduce the following $\pi$-calculus expressions:

1. $a(x).\overline{cx} | (\nu b)\overline{ab}$
2. $a(b).\overline{cb} | (\nu b)\overline{ab}$
3. $a(x).\overline{bx} | (\nu b)\overline{ab}$
4. $(\nu a)\overline{ab}.P | \overline{ac}.Q | a(x).R$

- Notice that we use $\overline{xz}$ to denote the process $\overline{xz}.0$.
- We will also use $a(x)$ to denote the process $a(x).0$. 
Executor Example

To illustrate process declarations and invocations, consider the following executor example:

\[ \text{Exec}(x) \triangleq x(y).\bar{y} \]  

(2)

\[ \Lambda(x) \triangleq (\nu z)(\bar{x}z \mid z.P) \]  

(3)

Notice that we use \( a \) and \( \bar{a} \) to denote reading and writing on channel \( a \), where the actual value being written and read is unimportant (i.e., it does not appear in the scope of the reading process.) What matters is that there is a communication over the channel, signaling synchronization or coordination between processes. That is, \( a.P \mid \bar{a}.Q \xrightarrow{\tau} P \mid Q. \)

The process \( \Lambda(x) \) will behave as \( P \) when composed with \( \text{Exec}(x) \).
Replication

Another useful process declaration enables recursive and non-terminating programs to be modeled:

\[ !P \triangleq P \mid !P \]

For example, a process, \( P \), that can indefinitely receive values on channel \( a \) can be modelled in the \( \pi \) calculus as:

\[ P = !a(x) \]

If \( P \) interacts with another process, \( Q = \tilde{a}y \), which writes a value over channel \( a \), \( P \mid Q \) can be unfolded as follows:

\[
\begin{align*}
P \mid Q &= !a(x) \mid \tilde{a}y \equiv (a(x) \mid !a(x)) \mid \tilde{a}y \equiv (a(x) \mid \tilde{a}y) \mid !a(x) \\
&\frac{\tau}{0} \mid !a(x) \equiv !a(x) = P
\end{align*}
\]
Reference Cell in the \( \pi \) calculus

A reference cell can be defined in the \( \pi \) calculus as follows:

\[
\begin{align*}
\text{Ref}(g, s, i) & \triangleq (\nu l)(\bar{l}i \mid \text{GetServer}(l, g) \mid \text{SetServer}(l, s)) \\
\text{GetServer}(l, g) & \triangleq !g(c).l(v).(\bar{c}v \mid \bar{l}v) \\
\text{SetServer}(l, s) & \triangleq !s(c, v').l(v).(\bar{c} \mid \bar{l}v')
\end{align*}
\]

The following is an example process expression representing a client of the reference cell:

\[
(\nu c)\bar{s}(c, v).c.(\nu d)\bar{g}d.d(e).P
\]

In this example, process \( P \) will eventually receive the value of the reference cell over channel variable \( e \). If no other processes are interacting with the cell, it will receive the value \( v \).