Calculi for Service Oriented Computing

Rocco De Nicola

Dipartimento di Sistemi ed Informatica, Università di Firenze

GLOBAN School
Warsaw - September 2008

Part I: Basic Process Calculi

Sequential Programming vs Concurrent Programming

Classical Sequential Programming

1. Denotational semantics: the meaning of a program is a partial function from states to states
2. Nontermination is bad!
3. In case of termination, the result is unique.

Concurrent - Interactive - Reactive Programming

1. Denotational semantics is very complicate due to nondeterminism
2. Nontermination might be good!
3. In case of termination, the result might not be unique.

Programming Reactive System

The denotational approach is not sufficient for modelling systems such as:

- Operating systems
- Communication protocols
- Mobile phones
- Vending machines
- Web Services

The above systems compute by reacting to stimuli from their environment and are known as Reactive Systems. Their distinguishing features are:

- Interaction (many parallel communicating processes)
- Nondeterminism (results are not necessarily unique)
- There may be no visible result (exchange of messages is used to coordinate progress)
- Nontermination is good (services are expected to run continuously)

Analysis of Reactive Systems

Even short parallel programs may be hard to analyze, thus we need to face few questions:

1. How can we develop (design) a system that "works"?
2. How do we analyze (verify) such a system?

We need appropriate theories and formal methods and tools, otherwise we will experience again:

1. To study mathematical models for the formal description and analysis of concurrent programs.
2. To devise formal languages for the specification of the possible behaviour of parallel and reactive systems.
3. To develop verification tools and implementation techniques underlying them.

Our Approach

Process Algebras and Operational Semantics

- The chosen abstraction for reactive systems is the notion of processes.
- A process performs an action and becomes another process.
- Labelled Transition Systems (LTS) describe the behaviour of processes and their interaction.

These Lectures

- We shall (briefly) introduce the SOS approach to the specification of reactive systems and two important process calculi *CCS* and π-calculus
- We shall then discuss the generalisation of the process algebraic approach to Service oriented Computing and present a formalism that we have developed within the SENSORIA project.

Outline of the lectures

1. Labelled Transition Systems as Concurrency Models
2. Process Calculi and their semantics
3. Service Oriented Process Calculi
4. CASPIS
5. MarCaspis
6. ???

Operational Semantics for Concurrent Processes

Systems are described by associating to them a behaviour represented as transition graph: two main models (or variants thereof) have been used.

Kripke Structures

State Labelled Graphs: States are labelled with the properties that are considered relevant (e.g. the value of - the relation between - some variables)

Labelled Transition Systems

Transition Labelled Graph: Transition between states are labelled with the action that induces the transition from one state to another.

In this lectures, we shall mainly rely on **Labelled Transition Systems** and actions will play an important role.

Finite State Automata

Definition

A *finite state automaton* M is a 5-tuple
$M = (Q, A, \rightarrow, q_0, F)$ *where*

- Q is a finite set of states
- A is the alphabet
- $\rightarrow \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$ is the transition relation
- $q_0 \in Q$ is a special state called initial state,
- $F \subseteq Q$ is the set of (final states)

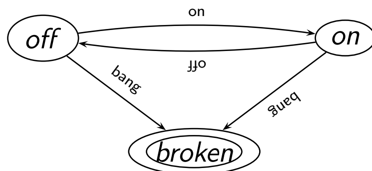


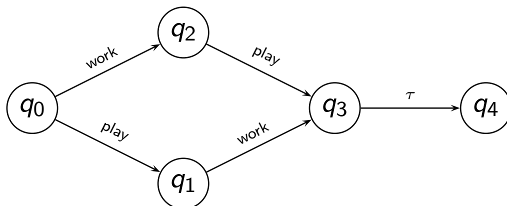
Figure: A finite state automaton

Labelled Transition Systems

Definition

A Labelled Transition System S is a 4-tuple $S = (Q, A, \rightarrow, q_0)$ *where* :

- Q is a set of states
- A is a finite set of actions
- $\rightarrow \subseteq Q \times A \times Q$ is a ternary relation called transition relation it is often written $q \xrightarrow{a} q'$ instead of $(q, a, q') \in \rightarrow$
- $q_0 \in Q$ is a special state called initial state.



If initial states are not relevant (or known) LTSs are triples $(Q, A, \rightarrow)$ ...

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

We have two main types of atomic actions:

- Visible Actions
- Internal Actions

Why operators for describing systems

How can we describe (possibly very large) automata or LTSs?

As a table?

Rows and columns are labelled by states, entries are either empty or marked with a set of actions.

As a listing of triples?

$\rightarrow = \{(q_0, a, q_1),\ (q_0, a, q_2),\ (q_1, b, q_3),\ (q_1, c, q_4),\ (q_2, \tau, q_3),\ (q_2, \tau, q_4)\}.$

As a more compact listing of triples?

$\rightarrow = \big\{(q_0, a, \{q_1, q_2\}),\ (q_1, b, q_3),\ (q_1, c, q_4),\ (q_2, \tau, \{q_3, q_4\})\big\}.$

As XML?

`<lts><ar><st>q0</st><lab>a</lab><st>q1</st></ar>...</lts>`.

Why operators for describing systems - ctd

Linguistic aspects are important!

The previous solutions are ok for machines ... not for humans.

Are prefix and sum operators sufficient?

They are ok to describe small finite systems:

- $p = a.b.(c + d)$
- $q = a.(b.c + b.d)$
- $r = a.b.c + a.c.d$

But additional operators are needed

- to design systems in a structured way (e.g. $p|q$)
- to model systems interaction
- to abstract from details
- to represent infinite systems

Operational Semantics of Operators

Inference Systems

An inference system is a set of inference rules of the form

$$\frac{p_1, \cdots, p_n}{q}$$

Inference Rules

To each process built using operators we associate an LTS by relying on structural induction to define the meaning of each operator (states of the LTS are named by processes terms). For a generic operator *op* we have one or more rules of the form:

$$\frac{E_{i_1} \xrightarrow{\alpha_1} E'_{i_1} \quad \cdots \quad E_{i_m} \xrightarrow{\alpha_m} E'_{i_m}}{op(E_1, \cdots, E_n) \xrightarrow{\alpha} op(E'_1, \cdots, E'_n)} \qquad \text{where } \{i_1, \cdots, i_m\} \subseteq \{1, \cdots, n\}.$$

The elegance of Operational Semantics

LTS as terms

Few SOS rules define all the automata that can ever be specified with the chosen operators. Given any term, the rules are used to derive the corresponding automaton. The set of rules is fixed once and for all. The LTS is the least one satisfying the inference rules.

Definition by Structural induction

The interaction of complex systems is defined in terms of the behavior of their components.

Proofs by Rule induction

A property is true for the whole LTS if whenever it holds for the premises of each rule, it holds also for the conclusion.

Presentations of Labelled Transition Systems

Process Algebra as denotations of LTS

- LTS are represented by terms of process algebras.
- Terms are interpreted via operational semantics as LTS.

Process Algebra Basic Principles

1. Define a few elementary (atomic) processes modelling the simplest process behaviour;
2. Define appropriate composition operations to build more complex process behaviour from (existing) simpler ones.

Regular Expressions as Process Algebras

Syntax of Regular Expressions

$E \ ::= \ 0 \mid 1 \mid a \mid E+E \mid E \cdot E \mid E^*$ *with* $a \in A$ *and* $-$ *below* $-$ $\mu \in A \cup \{\varepsilon\}$

Operational Semantics of Regular Expressions

(Tic) $$\frac{}{1 \xrightarrow{\varepsilon} 1}$$

(Atom) $$\frac{}{a \xrightarrow{a} 1}$$

(Sum$_1$) $$\frac{e \xrightarrow{\mu} e'}{e+f \xrightarrow{\mu} e'}$$

(Sum$_2$) $$\frac{f \xrightarrow{\mu} f'}{e+f \xrightarrow{\mu} f'}$$

(Seq$_1$) $$\frac{e \xrightarrow{\mu} e'}{e \cdot f \xrightarrow{\mu} e' \cdot f}$$

(Seq$_2$) $$\frac{e \xrightarrow{\varepsilon} 1}{e \cdot f \xrightarrow{\varepsilon} f}$$

(Star$_1$) $$\frac{}{e^* \xrightarrow{\varepsilon} 1}$$

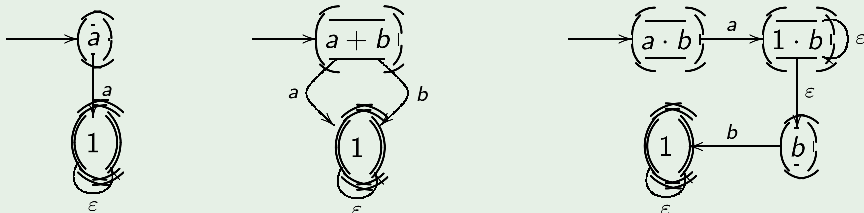
(Star$_2$) $$\frac{e \xrightarrow{\mu} e'}{e^* \xrightarrow{\mu} e' \cdot e^*}$$

We assume $a \in A$ and $\mu \in A \cup \{\varepsilon\}$.

The automaton associated to a regular expression

The SOS inference rules implicitly defines a particular automaton for each regular expression e (essentially a fragment of the whole LTS):

- the initial state is e (we shall often omit to mark it)
- the set of labels is A
- the set of states consists of all r.e. that can be reached starting from e via a sequence of transitions
- the transition relation is the one induced from the SOS inference rules
- the only final state is 1 (we shall often omit to mark it)



CCS Basics (Sequential Fragment)

- *Nil* (or 0) process (the only atomic process)
- action prefixing ($a.P$)
- names and recursive definitions ($\triangleq$)
- nondeterministic choice ($+$)

This is Enough to Describe Sequential Processes

Any finite LTS can be described (up to isomorphism) by using the operations above.

CCS Basics (Parallelism and Renaming)

- parallel composition ($|$)
  (synchronous communication between two components = handshake synchronization)
- restriction ($P \setminus L$)
- relabelling ($P[f]$)

Definition of CCS (channels, actions, process names)

Let

- $\mathcal{A}$ be a set of channel names (e.g. *tea*, *coffee* are channel names)
- $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ be a set of labels where
  - $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$
    (elements of $\mathcal{A}$ are called names and those of $\overline{\mathcal{A}}$ are called co-names)
  - by convention $\overline{\overline{a}} = a$
- $Act = \mathcal{L} \cup \{\tau\}$ is the set of actions where
  - τ is the internal or silent action

  (e.g. τ, *tea*, $\overline{coffee}$ are actions)
- $\mathcal{K}$ is a set of process names (constants).

Definition of CCS (expressions)

$P :=$	K	$\mid$	process constants ($K \in \mathcal{K}$)
	$\alpha.P$	$\mid$	prefixing ($\alpha \in Act$)
	$\sum_{i \in I} P_i$	$\mid$	summation (I is an arbitrary index set)
	$P_1 \mid P_2$	$\mid$	parallel composition
	$P \setminus L$	$\mid$	restriction ($L \subseteq \mathcal{A}$)
	$P[f]$	$\mid$	relabelling ($f : Act \to Act$) such that

- $f(\tau) = \tau$
- $f(\overline{a}) = \overline{f(a)}$

The set of all terms generated by the abstract syntax is the set of CCS process expressions (and is denoted by $\mathcal{P}$).

Notation

$$P_1 + P_2 = \sum_{i \in \{1,2\}} P_i \qquad Nil = 0 = \sum_{i \in \emptyset} P_i$$

Precedence

Precedence

1. restriction and relabelling (tightest binding)
2. action prefixing
3. parallel composition
4. summation

Example: $R + a.P | b.Q \setminus L$ means $R + \big((a.P) | (b.(Q \setminus L))\big)$.

Definition of CCS (defining equations)

CCS program

A collection of defining equations of the form

$$K \triangleq P$$

where $K \in \mathcal{K}$ is a process constant and $P \in \mathcal{P}$ is a CCS process expression.

- Only one defining equation per process constant.
- Recursion is allowed: e.g. $A \triangleq \overline{a}.A \mid A$.

Structural Operational Semantics for CCS

Structural Operational Semantics (SOS)—G. Plotkin 1981

Small-step operational semantics where the behaviour of a system is inferred using syntax driven rules.

Given a collection of CCS defining equations, we define the following LTS $(Proc, Act, \{\xrightarrow{a} \mid a \in Act\})$:

- $Proc = \mathcal{P}$ (the set of all CCS process expressions)
- $Act = \mathcal{L} \cup \{\tau\}$ (the set of all CCS actions including τ)
- transition relation is given by SOS rules of the form:

$$\text{RULE} \quad \frac{premises}{conclusion} \quad conditions$$

SOS rules for CCS ($\alpha \in Act$, $a \in \mathcal{L}$)

$$\text{ACT} \quad \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad\qquad \text{SUM}_j \quad \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_j} \; j \in I$$

$$\text{COM1} \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad\qquad \text{COM2} \quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$\text{COM3} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\text{RES} \quad \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \; \alpha, \overline{\alpha} \notin L \qquad\qquad \text{REL} \quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$\text{CON} \quad \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'} \; K \triangleq P$$

Deriving Transitions in CCS

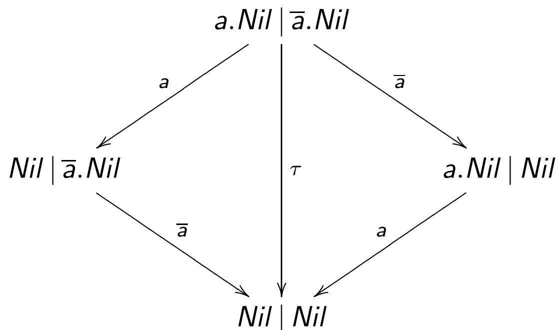
Let $A \triangleq a.A$. Then

$$\big((A \,|\, \overline{a}.Nil) \,|\, b.Nil\big)[c/a] \xrightarrow{c} \big((A \,|\, \overline{a}.Nil) \,|\, b.Nil\big)[c/a].$$

Why?

$$\text{REL}\ \dfrac{\text{COM1}\ \dfrac{\text{COM1}\ \dfrac{\text{CON}\ \dfrac{\text{ACT}\ \dfrac{}{a.A \xrightarrow{a} A}}{A \xrightarrow{a} A}\ A \triangleq a.A}{A \,|\, \overline{a}.Nil \xrightarrow{a} A \,|\, \overline{a}.Nil}}{(A \,|\, \overline{a}.Nil) \,|\, b.Nil \xrightarrow{a} (A \,|\, \overline{a}.Nil) \,|\, b.Nil}}{\big((A \,|\, \overline{a}.Nil) \,|\, b.Nil\big)[c/a] \xrightarrow{c} \big((A \,|\, \overline{a}.Nil) \,|\, b.Nil\big)[c/a]}$$

LTS of the Process $a.Nil \mid \overline{a}.Nil$



Value Passing CCS

Main Idea

Handshake synchronization is extended with the possibility to exchange data (e.g., integers).

$$\overline{pay(6)}.Nil \mid pay(x).\overline{save(x/2)}.Nil \mid Bank(100)$$

$$\downarrow \tau$$

$$Nil \mid \overline{save}(3).Nil \mid Bank(100)$$

$$\downarrow \tau$$

$$Nil \mid Nil \mid Bank(103)$$

Parametrized Process Constants

For example: $Bank(total) \triangleq save(x).Bank(total + x)$.

Translation of Value Passing CCS to Standard CCS

$$\longrightarrow$$

Value Passing CCS

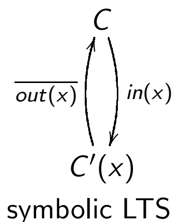
$$C \triangleq in(x).C'(x)$$

$$C'(x) \triangleq \overline{out(x)}.C$$

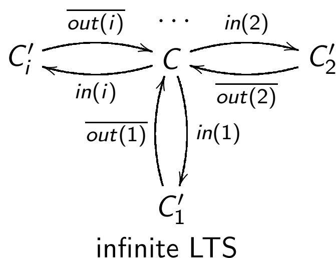
Standard CCS

$$C \triangleq \sum_{i \in \mathbb{N}} in(i).C'_i$$

$$C'_i \triangleq \overline{out(i)}.C$$



symbolic LTS



infinite LTS

CCS Has Full Turing Power

Fact

CCS can simulate a computation of any Turing machine.

Remark

Hence CCS is as expressive as any other programming language but its use is to rather describe the behaviour of reactive systems than to perform specific calculations.

Strong Bisimilarity

Let $(Proc, Act, \{\xrightarrow{a} \mid a \in Act\})$ be an LTS.

Strong Bisimulation

A binary relation $R \subseteq Proc \times Proc$ is a strong bisimulation iff whenever $(s,t) \in R$ then for each $a \in Act$:

- if $s \xrightarrow{a} s'$ then $t \xrightarrow{a} t'$ for some t' such that $(s', t') \in R$
- if $t \xrightarrow{a} t'$ then $s \xrightarrow{a} s'$ for some s' such that $(s', t') \in R$.

Strong Bisimilarity

Two processes $p_1, p_2 \in Proc$ are strongly bisimilar ($p_1 \sim p_2$) if and only if there exists a strong bisimulation R such that $(p_1, p_2) \in R$.

$$\sim \; = \bigcup \{R \mid R \text{ is a strong bisimulation}\}$$

Example – Buffer

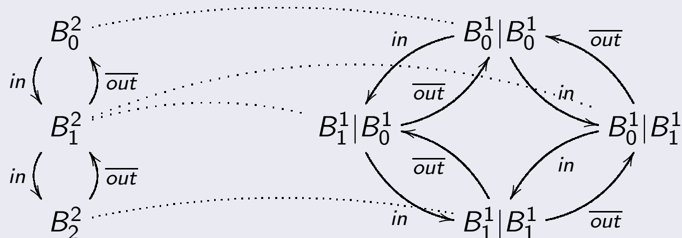
Buffer of Capacity 1

$B_0^1 \triangleq in.B_1^1$
$B_1^1 \triangleq \overline{out}.B_0^1$

Buffer of Capacity n

$B_0^n \triangleq in.B_1^n \; B_i^n \triangleq in.B_{i+1}^n + \overline{out}.B_{i-1}^n \quad \text{for } 0 < i < n$
$B_n^n \triangleq \overline{out}.B_{n-1}^n$

Example: $B_0^2 \sim B_0^1|B_0^1$



Example – Buffer

Theorem

For all natural numbers n: $$B_0^n \sim \underbrace{B_0^1|B_0^1|\cdots|B_0^1}_{n\ times}$$

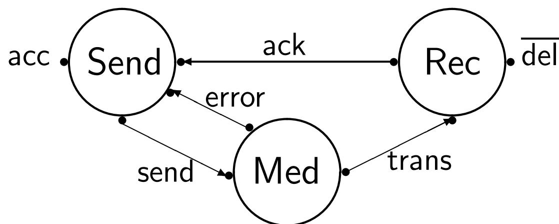
Proof.

Construct the following binary relation where $i_1, i_2, \ldots, i_n \in \{0, 1\}$.

$$R = \{\big(B_i^n,\ B_{i_1}^1|B_{i_2}^1|\cdots|B_{i_n}^1\big) \mid \sum_{j=1}^{n} i_j = i\}$$

- $\big(B_0^n,\ B_0^1|B_0^1|\cdots|B_0^1\big) \in R$
- R is strong bisimulation

Case Study: Communication Protocol



$$
\begin{aligned}
\text{Send} &\triangleq \text{acc.Sending} \\
\text{Sending} &\triangleq \overline{\text{send}}.\text{Wait} \\
\text{Wait} &\triangleq \text{ack.Send} + \text{error.Sending}
\end{aligned}
\qquad
\begin{aligned}
\text{Rec} &\triangleq \text{trans.Del} \\
\text{Del} &\triangleq \overline{\text{del}}.\text{Ack} \\
\text{Ack} &\triangleq \overline{\text{ack}}.\text{Rec}
\end{aligned}
$$

$$
\begin{aligned}
\text{Med} &\triangleq \text{send.Med}' \\
\text{Med}' &\triangleq \tau.\text{Err} + \overline{\text{trans}}.\text{Med} \\
\text{Err} &\triangleq \overline{\text{error}}.\text{Med}
\end{aligned}
$$

Verification Question

$$\mathsf{Impl} \triangleq (\mathsf{Send} \,|\, \mathsf{Med} \,|\, \mathsf{Rec}) \setminus \{\mathsf{send}, \mathsf{trans}, \mathsf{ack}, \mathsf{error}\}$$

$$\mathsf{Spec} \triangleq \mathsf{acc}.\overline{\mathsf{del}}.\mathsf{Spec}$$

Question

$$\mathsf{Impl} \overset{?}{\approx} \mathsf{Spec}$$

1. Draw the LTS of Impl and Spec and prove (by hand) the equivalence.
2. Use appropriate tools, e.g. TAPAS.

From CCS to π-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.

$$C \triangleq \overline{askPizza}.\overline{pay}.pizza \qquad P \triangleq askPizza.pay.\overline{pizza}.P$$

1. The occasional client C asks for a pizza, pays and takes it away.
2. The "pizzaiolo" P receives the request for the pizza, gets the money, delivers the pizza and waits for another customer.

From CCS to π-calculus - 2

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

$$C \triangleq \overline{askPizza}\langle margherita\rangle.\overline{pay}\langle 5\ Euro\rangle.pizza$$

$$\begin{aligned} P \triangleq\ & askPizza(x).pay(y).\textbf{if}\ y = price(x)\ \textbf{then}\ \overline{pizza}.P \\ & \textbf{else if}\ y > price(x)\ \textbf{then}\ \overline{pizza}.\overline{output}\langle y - price(x)\rangle.P \\ & \textbf{else}\ \overline{askMoney}\ldots \end{aligned}$$

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 π-calculus - 3

With π-calculus we can do more: home delivery of pizza!

$$\begin{array}{lcl} C & \triangleq & \overline{askPizza}\langle myHome\rangle.\overline{pay}.myHome(x).\overline{eat}\langle x\rangle \\ P & \triangleq & askPizza(y).pay.(\nu pizza)\overline{y}\langle pizza\rangle.P \end{array}$$

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

$$\begin{array}{llcl} \multicolumn{4}{l}{\overline{askPizza}\langle myHome\rangle.\overline{pay}.myHome(x).\overline{eat}\langle x\rangle \mid} \\ & & & askPizza(y).pay.(\nu pizza)\overline{y}\langle pizza\rangle.P \\ \xrightarrow{\tau} & \overline{pay}.myHome(x).\overline{eat}\langle x\rangle & \mid & pay.(\nu pizza)\overline{myHome}\langle pizza\rangle.P \\ \xrightarrow{\tau} & \overline{pay}.myHome(x).\overline{eat}\langle x\rangle & \mid & pay.(\nu pizza)\overline{myHome}\langle pizza\rangle.P \\ \xrightarrow{\tau} & myHome(x).\overline{eat}\langle x\rangle & \mid & (\nu pizza)\overline{myHome}\langle pizza\rangle.P \\ \xrightarrow{\tau} & (\nu pizza)(\overline{eat}\langle pizza\rangle & \mid & P) \end{array}$$

$\pi-$calculus: a calculus of names

π-calculus is a general model of (interaction-based) concurrent computation and represents a theoretical setting for studying concurrent mobile systems by an appropriate handling of names.

- Links are created by sharing names.
- Actions are used to change system connectivityover time.

Names are:

1. channels
2. identifiers
3. values (data)
4. references
5. locations
6. encryption keys
7. ...

Names can be:

1. created and destroyed
2. sent around to share information
3. acquired to communicate with previously unknown processes
4. tested to take decisions
5. used as private means of communication, e.g. to share secret

Syntax of π-calculus

Let $\mathcal{N}$ be countably infinite set of names.

$$
\begin{array}{rcll}
\text{(Processes)}\ P & ::= & S & \text{sum} \\
& | & P_1|P_2 & \text{parallel composition} \\
& | & (\nu x)P & \text{name restriction} \\
& | & !P & \text{replication} \\
\\
\text{(Sums)}\ S & ::= & \mathbf{0} & \text{inactive process (nil)} \\
& | & \pi.P & \text{prefix} \\
& | & S_1 + S_2 & \text{choice} \\
\\
\text{(Prefixes)}\ \pi & ::= & \overline{x}\langle y\rangle & \text{sends } y \text{ on } x \\
& | & x(z) & \text{substitutes for } z \text{ the name received on } x \\
& | & \tau & \text{internal action} \\
& | & [x = y]\pi & \text{matching: tests equality of } x \text{ and } y
\end{array}
$$

Notation, Comments and Remarks

- $(\nu z)P$ is alike CCS restriction $P\backslash z$.
- $!P$ 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 **if** $x = y$ **then** $\pi.P$.
- Occurrences of $\mathbf{0}$ will sometimes be omitted, thus, e.g., $\overline{x}\langle y\rangle.\mathbf{0}$ will be written $\overline{x}\langle y\rangle$.
- $x(z)$ indicates input, while $\overline{x}\langle y\rangle$ indicates output.
- In $x(z).P$ e $(\nu z)P$, 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*.
- $fn(P)$ and $bn(P)$ are the sets of all free, resp. bound, names of P.
- Processes are considered up to *alpha-conversion* ($=_\alpha$) which permits renaming bound names into *fresh* (not previously used) ones:

$$(\nu d)\overline{a}\langle d\rangle.a(y).y(z).\mathbf{0} =_\alpha (\nu c)\overline{a}\langle c\rangle.a(x).x(z).\mathbf{0}.$$

Name Binding

Names Bound by Input Prefix

The binding of a name z in a π-calculus process $x(z).P$ resembles the binding of a variable z in a λ-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 $(\nu z)\overline{x}\langle z\rangle.Q$ can *extrude* the scope of the restricted name z by passing it on channel x to other processes
- the process $(\nu z)(\ \overline{x}\langle z\rangle|z(y).Q\ )$ 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 σ 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 σ such that $x_i\sigma = y_i$ for $i = 1, \ldots, n$ and $x\sigma = x$ otherwise.

Capture Avoidance

When applying σ 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 $P[{}^{z}\!/\!{}_{x}]$ when $P = y(z).\overline{z}\langle x\rangle$?
$P[{}^{z}\!/\!{}_{x}] = y(z).\overline{z}\langle z\rangle$ is a wrong answer!

Definition (α-Conversion)

1. 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.
2. 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.
3. 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.
4. Two processes P and Q are *α-convertible*, written $P =_{\alpha} Q$ if Q can be obtained from P by a finite number of changes of bound names.

We have $P =_{\alpha} y(w).\overline{w}\langle x\rangle$ and $P[{}^{z}\!/\!{}_{x}] =_{\alpha} y(w).\overline{w}\langle z\rangle$ is a right answer.

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 α-conversion first).

Application of σ to Prefixes

$$
\begin{array}{rcl}
(\overline{x}\langle y\rangle)\sigma & = & \overline{x\sigma}\langle y\sigma\rangle \\
(x(z))\sigma & = & x\sigma(z) \\
\tau\sigma & = & \tau \\
([x=y]\pi)\sigma & = & [x\sigma = y\sigma]\pi\sigma
\end{array}
$$

Application of σ to Processes

$$
\begin{array}{rcl}
\mathbf{0}\sigma & = & \mathbf{0} \\
(\pi.P)\sigma & = & \pi\sigma.P\sigma \\
(S_1 + S_2)\sigma & = & S_1\sigma + S_2\sigma \\
(P_1|P_2)\sigma & = & P_1\sigma|P_2\sigma \\
((\nu z)P)\sigma & = & (\nu z)P\sigma \\
(!P)\sigma & = & !P\sigma
\end{array}
$$

$$
y(z).\overline{z}\langle x\rangle[^z/_x] =_\alpha y(w).\overline{w}\langle z\rangle
$$

An LTS for π-calculus

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

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

$$(COM)\ \frac{P \xrightarrow{ab} P' \qquad Q \xrightarrow{\bar{a}b} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$(SUM)\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$(RES)\ \frac{P \xrightarrow{\alpha} P'}{(\nu b)P \xrightarrow{\alpha} (\nu b)P'}\ b \notin n(\alpha)$$

$$(REP)\ \frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

$$(CLO)\ \frac{P \xrightarrow{ab} P' \qquad Q \xrightarrow{\bar{a}(b)} Q'}{P \mid Q \xrightarrow{\tau} (\nu b)(P' \mid Q')}\ b \notin fn(P)$$

$$(OPEN)\ \frac{P \xrightarrow{\bar{a}b} P'}{(\nu b)P \xrightarrow{\bar{a}(b)} P'}\ a \neq b$$

$$(PAR)\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ bn(\alpha) \cap fn(Q) = \emptyset$$

$$(EQ)\ \frac{\pi.P \xrightarrow{\alpha} P'}{[a = a]\pi.P \xrightarrow{\alpha} P'}$$

Plus Symmetric Rules for (COM), (SUM), (CLO) and (PAR).

Structural Congruence for π-calculus - I

Why Structural Congruence?

The syntax of π-calculus processes is to some extent *too concrete* (even when taken up-to α-conversion):

1. The order in which we compose processes in parallel should not matter.
2. The order in which we compose processes in sums should not matter.
3. 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:

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

Structural Congruence for π-calculus - II

$$P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$$

$$S + \mathbf{0} \equiv S \qquad S_1 + S_2 \equiv S_2 + S_1 \qquad S_1 + (S_2 + S_3) \equiv (S_1 + S_2) + S_3$$

$$!P \equiv P \mid !P \qquad [a = a]\pi.P \equiv \pi.P$$

$$(\nu a)\mathbf{0} \equiv \mathbf{0} \qquad (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P \qquad \frac{a \notin fn(P)}{P \mid (\nu a)Q \equiv (\nu a)(P \mid Q)}$$

$$P \equiv P \qquad \frac{P \equiv Q}{Q \equiv P} \qquad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R} \quad \text{(equivalence)}$$

$$\frac{P =_\alpha P'}{P \equiv P'} \qquad \frac{P \equiv P'}{\mathbb{C}[P] \equiv \mathbb{C}[P']} \quad \text{(congruence)}$$

Canonical Form

For each π-calculus process P there exist:

1. a finite number of names $x_1, ..., x_k$,
2. a finite number of sums $S_1, ..., S_n$, and
3. a finite number of processes $P_1, ..., P_m$ such that

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

The structural congruence allows one to rearrange the syntactic terms describing π-calculus processes so that any two possible interacting sub-terms 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

(R-Com) $$(R + a(x).P) \mid (R + \overline{a}\langle b\rangle.Q) \longmapsto P[{}^{b}/_{x}] \mid Q$$

(R-Par) $$\frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q}$$

(R-Res) $$\frac{P \longmapsto P'}{(\nu a)P \longmapsto (\nu a)P'}$$

(R-Struct) $$\frac{P \equiv Q \quad Q \longmapsto Q' \quad Q' \equiv P'}{P \rightarrow P'}$$

Electoral Propaganda

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

Assume we want to campaign for Walter V. and have set up the following scenario :

Naive Campaigning

$$
\begin{array}{rcl}
Speaker & \triangleq & \overline{air}\langle vote\ walter\rangle \\
Microphone & \triangleq & air(x).\overline{wire}\langle x\rangle \\
Loudspeaker & \triangleq & wire(y).\overline{highvolume}\langle y\rangle \\
Ad & \triangleq & Speaker \mid Microphone \mid Loudspeaker
\end{array}
$$

This system will evolve as follows:

$$
\begin{array}{rcl}
Ad & \longmapsto & \overline{wire}\langle vote\ walter\rangle \mid Loudspeaker \\
 & \longmapsto & \overline{highvolume}\langle vote\ walter\rangle
\end{array}
$$

Electoral Propaganda and Intrusions

$$
\begin{array}{rcl}
Speaker & \triangleq & \overline{air}\langle vote\ walter\rangle \\
Microphone & \triangleq & air(x).\overline{wire}\langle x\rangle \\
Loudpeaker & \triangleq & wire(y).\overline{highvolume}\langle y\rangle \\
Ad & \triangleq & Speaker \mid Microphone \mid Loudspeaker
\end{array}
$$

Let $\quad Rival \triangleq wire(z).\overline{wire}\langle vote\ silvio\rangle$

$$
\begin{array}{rcl}
Ad \mid Rival & \longmapsto & \overline{wire}\langle vote\ walter\rangle \mid Loudspeaker \mid Rival \\
& \longmapsto & \overline{wire}\langle vote\ silvio\rangle \mid Loudspeaker \\
& \longmapsto & \overline{highvolume}\langle vote\ silvio\rangle
\end{array}
$$

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

$$
SecureAd \triangleq (\nu\, air, wire)(Speaker \mid Microphone \mid Loudspeaker)
$$

Establishing Secure Communication Channels

Consider two processes Alice e 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:

$$\begin{array}{lcl} A & \triangleq & (\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle mess\rangle \\ S & \triangleq & !c_{AS}(x).\overline{c_{BS}}\langle x\rangle \mid !c_{BS}(y).\overline{c_{AS}}\langle y\rangle \\ B & \triangleq & c_{BS}(z).z(w).< use\ z > \end{array}$$

$$(\nu c_{AS}, c_{BS})(A|S|B) \longmapsto\longmapsto\longmapsto (\nu c_{AS}, c_{BS}, c_{AB})(S \mid\ < use\ mess >)$$

Synchrony vs Asynchrony

π-calculus has three important (distinguishing?) features:

blocking input: A process P prefixed by an input action is blocked until the input in not received. P needs to be suspended because it 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.

First Alternative

Syntax is left unchanged (only, we ignore $+$)

$$P \quad ::= \quad \mathbf{0} \mid a(x).P \mid \overline{a}\langle b\rangle.P \mid P_1|P_2 \mid (\nu a)P \mid [a=b]P \mid\ !P$$

and the following two rules do replace the (R-COM) rule.

$$\overline{a}\langle b\rangle.P \longmapsto \overline{a}\langle b\rangle \mid P \qquad\qquad a(x).P \mid \overline{a}\langle b\rangle \longmapsto P[^b/_x]$$

Second Alternative

The syntax above is modified by dropping the suffix of output actions

$$P \quad ::= \quad \mathbf{0} \mid a(x).P \mid \overline{a}\langle b\rangle \mid P_1|P_2 \mid (\nu a)P \mid [a=b]P \mid\ !P$$

and the (R-COM) rule is replaced by

$$a(x).P \mid \overline{a}\langle b\rangle \longmapsto P[^b/_x]$$

Encodings

Obvious Questions

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

To answer these questions for any pairs of calculi/laguages encodings have been devised together with criteria for assessing the quality of the chosen translations.

Translating asynchronous π-calculus into its synchronous variant is trivial.

WHY?

Because the former is a sub-calculus of the latter

For the other direction see next slide ...

Translating π-calculus into its asynchronous variant

$$
\begin{aligned}
[\![\varnothing]\!] &\triangleq \varnothing \\
[\![P_1|P_2]\!] &\triangleq [\![P_1]\!] \mid [\![P_2]\!] \\
[\![!P]\!] &\triangleq ![\![P]\!] \\
[\![(\nu n)P]\!] &\triangleq (\nu n)[\![P]\!] \\
[\![[a=b]P]\!] &\triangleq [a=b][\![P]\!] \\
[\![\overline{a}\langle b\rangle.P]\!] &\triangleq (\nu c)(\overline{a}\langle c\rangle \mid c(y).(\overline{y}\langle b\rangle \mid [\![P]\!])) \quad \text{with } c, y \notin fn(P) \\
[\![a(x).P]\!] &\triangleq a(z).(\nu d)(\overline{z}\langle d\rangle \mid d(y).[\![P]\!]) \quad \text{with } d, z \notin fn(P)
\end{aligned}
$$

A fresh name (c) is exchanged by exploiting the common channel (a); then channel c is then used to agree on a third channel to exchange the original intended values.

Translating π-calculus into its asynchronous π-calculus

$$\llbracket \overline{a}\langle b\rangle.P \rrbracket \triangleq (\nu c)(\overline{a}\langle c\rangle \mid c(y).(\overline{y}\langle b\rangle \mid \llbracket P \rrbracket)) \quad \text{with } c, y \notin fn(P)$$

$$\llbracket a(x).P \rrbracket \triangleq a(z).(\nu d)(\overline{z}\langle d\rangle \mid d(y).\llbracket P \rrbracket) \quad \text{with } d, z \notin fn(P)$$

A fresh name (c) is exchanged by exploiting the common channel (a). Then channel c is then used to agree on a third channel to exchange the original intended values:

1. The sender send on a a fresh name and waits to receive a new channel on this name, name b is then send on this third channel.
2. The receiver waits to receive on channel a and on the received name send a fresh name (an ack) and wait to receive on the ack that it has sent.