

Modelli formali per i sistemi distribuiti

Formal models for distributed systems

Mirko Viroli

DEIS

Università degli Studi di Bologna in Cesena

mviroli@deis.unibo.it



Organizzazione del seminario

- L'importanza dei modelli per l'ingegneria dei sistemi complessi
- Il problema dei sistemi distribuiti
- La formalizzazione dei sistemi "non distribuiti"...
 - Algoritmi e loro formalizzazione
 - Reduction Systems
- La formalizzazione dei sistemi "distribuiti"
 - Il problema dell'interazione
 - Macchine Interattive
- Le algebre di processi
 - discussione dei principali concetti dei sistemi distribuiti

1/5

L'importanza dei modelli per
l'ingegneria dei sistemi complessi

I modelli formali e l'ingegneria

- Modello : dà una rappresentazione astratta dei particolari di interesse di un sistema
- Esempi:
 - modelli fisici: rappresentazioni in scala..
 - modelli matematici: teoria delle probabilità..
- Quali modelli matematici per l'ingegneria?
 - descrivono in modo formale gli aspetti più complessi di un sistema da costruire
- Nell'ingegneria informatica:
 - danno una rappresentazione astratta e formale di entità e concetti, come:
 - comunicazione, algoritmi.. reti, computer,..

Separazione dei “Concerns”

- I modelli non devono rappresentare tutto il sistema

Tipicamente:

- Separazione delle problematiche in aspetti (concerns) il più possibile ortogonali fra loro
- Per ogni aspetto di interesse si definisce un modello che:
 - lo rappresenti come concetto “chiave”
 - che astragga da altri aspetti meno importanti
 - ossia: che si ponga al giusto livello di astrazione
- Per risolvere problemi estremamente complessi
 - si divide in diversi livelli di astrazione, affrontati in sequenza
 - ad esempio: top-down, bottom-up, o combinati

Ciclo di vita del software

- Analisi
 - Funzionamento del software
- Design
 - Organizzazione del software
- Implementazione
 - Costruire il software
- Validazione
 - Funziona correttamente?
- Che relazione fra fase e modello?
- I modelli formali possono essere d'ausilio in ogni fase
 - In questo seminario ci concentriamo sull' analisi/specifica

2/5

Il problema dei sistemi distribuiti

Cos'è un sistema "distribuito"?

- Generalmente:
 - un sistema composto da diverse entità poste in luoghi diversi che interagiscono tra loro
- Tipici aspetti dei sistemi distribuiti:
 - comunicazione: trasmissione di informazione fra entità
 - sincronizzazione: nozione di "evento" comune
 - concorrenza: evoluzione contemporanea delle entità
 - non determinismo: latenza reti, perdita messaggi
- Alcune di queste problematiche esistono anche in sistemi non distribuiti:
 - Architetture ad eventi: GUIs, Component-Based, ...
- Il punto chiave:
 - il concetto di INTERAZIONE: momento di sincronia/comunicazione fra entità concorrenti.

La complessità delle interazioni

- Come rappresentare le interazioni fra i componenti di un sistema distribuito:
 - quale informazione portano con loro?
 - qual è la causa che le genera?
 - da chi (e se) vengono ricevute/percepite, quando?
 - come raggiungono il destinatario?
 - quali relazioni fra gruppi di interazioni? (causalità, consistenza,..)?
 - che relazione fra interazione e cambiamento di stato?
- Come costruire un sistema ingegneristico che affronti questi aspetti?
- I modelli per i sistemi distribuiti devono consentire di comprenderne le problematiche, descriverle, progettarle, implementarle, validarle, etc...

Obiettivi (o utopie?)

- Caratterizzare in modo formale il comportamento interattivo di un componente software
 - es.: emette un messaggio ogni secondo e ne riceve uno ogni due
- Aggregare componenti tra loro
 - in modo sincrono/asincrono, con quale “colla”?
- Ottenere il comportamento del sistema risultante
- Cosa succede se sostituisco un componente con un altro?
 - il sistema funziona meglio, peggio, non funziona più....
- Data una descrizione formale ottenere in modo automatico un sistema dal corrispondente comportamento
- Riusare sistemi legacy con un approccio black-box:
 - osservando il suo comportamento interattivo

3/5

Formalising non-distributed
systems

Algorithm vs. Interaction

Two aspects of computing

- algorithmic:
 - how data are manipulated and transformed by a computing machine
- interactive:
 - how a computing machine interacts with others (or with humans)
 - accepting (partial) inputs
 - providing (partial) outputs

In general, each S/W component provides both aspects

- some are intrinsically algorithmic: computing MCD
- some are intrinsically interactive: a clock
- some have both: generating prime numbers until stopped

Computing = Transforming

- In its most traditional acceptation, a computation is a transformation (or translation) of information:
 - Input value: a sequence over Σ , i.e. an element of Σ^*
 - Output value: similarly, an element of Δ^*
- Example:
 - reversing a sequence:
 - $\Sigma=\{1,2,3\}$, 1223 is translated to 3221
- Algorithm:
 - a mechanical process made of a finite sequence of actions producing an output from an input

Algorithms and functions

- Can any computation be modelled in that way?
 - Each data structure can be represented as a string of a language (e.g. its representation in the PC memory)
 - Each algorithm accepts a data structure and produces a new one
 - Recognising the string of a language: $\Sigma^* \rightarrow \text{Boolean}$
 - Translating a string: $\Sigma^* \rightarrow \Delta^*$
 - Sorting a vector: $V \rightarrow V$
 - Searching for an element into a vector: $V \times E \rightarrow \text{Boolean}$
 - ...
- So, an algorithm can always be described as a mathematical function
 - From numerable sets (“no more elements than N”)
 - Possibly a partial function, due to the termination problem
- Question: does a “web server” realise an algorithm?

Formalisms for algorithms

- There are many:
 - Lambda Calculus
 - Automata
 - ASF
 - PDA
 - Turing Machine
 - Register Machine
 - ...
- What do they have in common?
 - they have an input
 - they elaborate on that input
 - they (possibly) terminates yielding an output

Reduction Systems

- Formally $\langle S, \rightarrow \rangle$, with $\rightarrow \subseteq S \times S$
 - S , information to elaborate, (a numerable set)
 - $\rightarrow$, elaboration
- Notation:
 - $\langle s, s' \rangle$ in $\rightarrow$, is written $s \rightarrow s'$
 - $s \rightarrow s' \rightarrow s'' \dots \rightarrow s^n$, is written $s \rightarrow^* s'$
- Computing:
 - starts from an “ s ” representing “ $f(\text{input})$ ”
 - applies $\rightarrow$ until reaching a final value “output”
- Loops can occurs: in particular, in any Turing-complete formalisms
- Can be used to deal with any formalism for Algorithms

Definition “by rule” of a RS

- First, given a numerable set S
 - mathematically, or by a BNF grammar
- Then, give rules defining the behaviour of “ $\rightarrow$ ”

$$\frac{\textit{condition}}{\sigma \rightarrow \sigma'}$$

- σ and σ' denote elements of S , but may contain some variable
- “condition” is any predicate on that variables
- $s \rightarrow s'$ if there’s a substitution of variables to terms so that:
 - “condition” is satisfied, and σ and σ' becomes s and s'

Example

- Want to define algorithms over sequences of integers...
 - $s ::= \varepsilon \mid n \mid s;s \mid ,$ e.g.: $s=1;2;3;7;8$
- Just consider one rule:

$$\frac{i > j}{s; i; j; s' \rightarrow s; j; i; s'}$$

- Which algorithm?
- Bubble-Sort (non-deterministic)
 - $3;2;1 \rightarrow 2;3;1 \rightarrow 2;1;3 \rightarrow 1;2;3$
 - $3;2;1 \rightarrow 3;1;2 \rightarrow 1;3;2 \rightarrow 1;2;3$

The framework of RS

- Can be used to:
 - give an abstract and compact representation of algorithms
 - allows for a mathematical treatment
- Which applicability?
 - the execution is “blind”
 - during execution
 - cannot provide an input value
 - to a certain extent
 - cannot observe partial results (which meaning?)
 - cannot drive non-determinism
- For algorithms this is OK, but what about real systems?
- What about compositionality?

4/5

Formalising distributed systems

The problem of interaction

- In order to describe modern systems
 - distributed, interactive, open
 - web application, pervasive computing, component-based systems
- ...we need a different framework, capturing:
 - interaction during computation
 - communicating information
 - synchronizing
 - composability of subsystems
 - environment

A paradigm-shift

Some outstanding papers:

- Peter Wegner, Why Interaction Is More Powerful Than Algorithms, Communications of the ACM., May 1997
<http://www.cs.brown.edu/people/pw/papers/ficacm.ps>
- Robin Milner, Elements of Interaction - Turing Award Lecture, Communications of the ACM., Jan 1993
download from www.acm.org, or ask to me

Transition systems

- A transition system: $\langle S, \rightarrow, A \rangle$ with $\rightarrow \subseteq S \times A \times S$
 - S , state of the system of interest, numerable
 - A , actions visible from outside, numerable
 - $\rightarrow$, represents change state + executing action
- Notation:

$$s \xrightarrow{a} s' \quad \langle s, a, s' \rangle \in \rightarrow$$

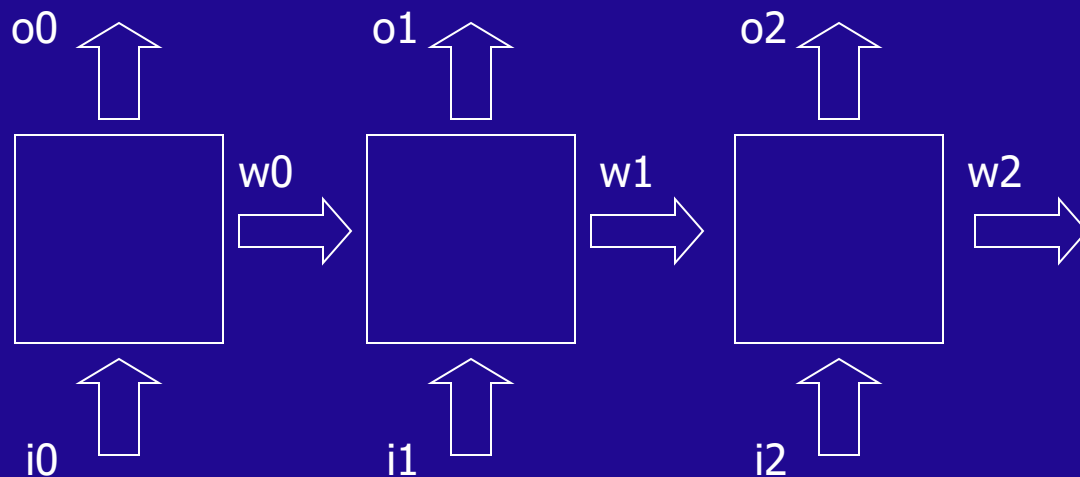
- Read “ from s move to s' by action a ”

Computation by interaction

- Computation:
 - from an initial state
 - as actions occur the state changes
 - (observe a final state?)
- What are actions? Can be interpreted as:
 - internal changes (algorithmic computations).. silent actions
 - receiving an input (a stimulus, a message)
 - producing an output (sending message)
- This change of perspective influenced
 - automata
 - (core) languages

Persistent Turing Machine

- Three tapes: input/work/output
- Computational Model
 - takes the inputs
 - elaborates on it, using work, and producing an output
 - for the next input, work is persistent
 - (resembles “systems theory”?)



Computing interaction histories

- Simple characterization as a TS
 - $\langle [\text{Work}], \rightarrow, \{\text{in}(n), \text{out}(n)\} \rangle$
- TM characterized as a function $F: N \rightarrow N$
- PTM characterized as a function $F: N^* \rightarrow N^*$
 - takes the sequence of inputs
 - produces the sequence of outputs
- Isn't it the same as a TS?
 - there is a 1:1 mapping of N^* to N

Wrapping interaction...

- N^* is numerable as N
 - Hence a TM can always simulate a PTM
- Each “closed” interactive machine can be wrapped into an algorithm
- However this simulation makes the hypothesis that
 - inputs are always represented before computation starts
 - output are provided all at the end
 - what about causality?
- PTM can also deal with infinite streams
 - enables reasoning about liveness properties
 - will a given situation ever occur?
 - (need infinite streams...)
- Expliciting interaction is required in order to have composability!

5/5

Process algebras

Algebras and Process Algebras

Ideas:

- capturing the few concepts of interaction
 - (very-high abstraction level)
- describing a process behaviour by a simple BNF syntax
 - an interactive behaviour can be given a very compact representation
- providing a semantics based on TS
 - from a process state moving to another process state by means of the occurrence of an interaction
- What is an algebra $(P, 0, +, |, \dots)$
 - A set P
 - A null element $(0, \text{the process doing nothing})$
 - A number of closed operators on P ($p + p' = p''$)
- Why? Reusing existing techniques
 - simplification, equivalence, reasoning....

Process algebras

- Supporting composability
 - defining a process
 - composition as an operator over two processes
 - explicit representation of communications
- Algebra of processes
 - operators of composition, choice, action execution,...
 - “parallel” of two processes as a process (states)
 - process moves to another (state) by executing an action
- Typically, described by giving the whole syntax and semantics
 - say, 8 operators, 10 semantic rules, 25 congruence rules
 - hard to grasp every notion

Our description

- In this seminar, an incremental description
 - starting from the simpler concepts
 - gentle introduction to the idea of process algebra
- Defining increasing algebras, each introducing a concept
 - Action execution (the elementary actions of a process)
 - Choice (the feature of non-determinism)
 - Parallel composition (interleaved concurrency)
 - Corresponding names (synchrony and events)
 - Replication (infinite behaviours)
 - Agent definition (library of specifications)
 - Name restriction (defining a system's boundary)
 - Value passing (communicating information)
 - Name passing (changing topology)

Operator 1: action execution

- Give a set of actions $a \in A$ (i/o, silents... whatever)
- Define
 - $P ::= 0 \mid a.P$
 - e.g.: $P_s = a.a'.a''.a.0$ (more simply: $a.a'.a''.a$)
 - P_s executes a , then a' , then a'' , then again a , then terminates
- Which (operational) semantics?

$$a.P \xrightarrow{a} P$$

- Evolution:

$$a.a'.a''.a \xrightarrow{a} a'.a''.a \xrightarrow{a'} a''.a \xrightarrow{a''} a \xrightarrow{a} 0$$

- Only one admissible interaction history $[a, a', a'', a]$

Actions and environment

$$a.a'.a''.a \xrightarrow{a} a'.a''.a \xrightarrow{a'} a''.a \xrightarrow{a''} a \xrightarrow{a} 0$$

- If an action is used to represent a communication
 - receiving a message
 - sending a message
- then, who is the peer that receives/sends it?
- We use the concept of environment
 - is an abstraction used to model the set of peers that interact with the process we are representing
- For a process sending a message, the environment should be willing to receive it!

Towards multiple histories

- Is it sensible to have systems allowing for just one interaction history?
- For instance:
 - we may want to model an unpredictable source of events
 - a system may receive at a time different kinds of messages
- In general:
 - as in PTM we want to characterise a system in terms of all its admissible interaction histories...
 - we need an operator for non-determinism
- Non-determinism:
 - technically, more evolutions are allowed
 - practically, it enables the idea of unpredictability! (typical of distribution)

Operator 2: Non-deterministic choice

- Introduced by extending our algebra of processes:
 - $P ::= 0 \mid a.P \mid P + P$
 - e.g.: $P_s = a.a' + a''.a$
 - P_s executes (a , then a') or (a'' then a)

- Semantics

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$$

- With the hypothesis: $(P+0)=P$, $P+Q=Q+P$,
 $(P+Q)+R=P+(Q+R)$
 - any P can be represented as a sum of sequences of actions
 - e.g.: $a.a.a.a + a.a.a.a + a.a$
- these are called congruence rules

Choices..

- Example: $P = a.a' + a''.a$

$$a.a' + a''.a \xrightarrow{a''} a \xrightarrow{a} 0$$

$$a.a' + a''.a \xrightarrow{a} a' \xrightarrow{a'} 0$$

- Two possible interaction histories, $[a,a']$ and $[a'',a]$
- Typically, we say that P has the observable behavior $\{[a,a'],[a'',a]\}$
 - amounts to the “completed-trace observational semantics”
 - we have actually many of such semantics...

Congruence rules

- Technically, congruence rules define an equivalence relation over processes
 - saying that the two processes are “literally” the same
- This is used to make operational semantics more compact and clean
 - operational rules: how an operator affects interactions
 - congruence rules: how an operator affects the structure of processes
- In the case of choice:
 - operational rule: choice allows only one process to carry on
 - congruence rules: a process as a finite choice between subprocesses (choice as a n-ary associative and communicative operator)

Operator 3: Parallel composition

- Another extension to our algebra
 - $P ::= \dots \mid P \mid P$
 - e.g.: $P_r = P_s \mid a.a'' = (a.a' + a''.a) \mid a.a''$
 - P_r is the composition of P_s and $a.a''$

- Semantics

$$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}$$

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$$

- Congruence: $(P \mid 0) = P$, $P \mid Q = Q \mid P$, $(P \mid Q) \mid R = P \mid (Q \mid R)$
 - any P can be represented as a sum of parallel of sequences of actions
 - e.g.: $(a.a.a.a \mid a.a.a \mid a.a) + (a.a.a.a \mid a) + (a.a)$
- This schema is called interleaved concurrency

Interleaved concurrency

- Consider the process $Pr = (a.a' + a''.a) | a.a'' = Q | R$
- There are main available evolutions:

$$(a.a' + a''.a) | a.a'' \xrightarrow{a} (a.a' + a''.a) | a'' \xrightarrow{a} a' || a''$$

$$(a.a' + a''.a) | a.a'' \xrightarrow{a} a' | a.a'' \xrightarrow{a'} a.a''$$

$$(a.a' + a''.a) | a.a'' \xrightarrow{a''} a | a.a'' \xrightarrow{a} a.a''$$

- Q and R separately proceed on their evolution

On distribution and concurrency

- In distributed systems there can be no notion of global time!
 - You cannot think of having a synchronised clock on each different site
 - You can only synchronise them by sending messages, which can be lost and introduce latency
 - This is typical of asynchronous systems, i.e. actual distributed ones
- If we don't have global time, we cannot have global state
- What does it mean that $P|Q$ represents two processes are in the state P and Q at the “same time”?
- Can the idea of process represents a distributed system's state?

The distributed state

- The concept of distributed state is introduced to fit the problem
 - fortunately, as these formalisms are anyway very useful!
- Relies on the notion of observation
 - we are interested only in the phenomena that are indeed observables
 - many relationship with respect to Heisenberg (still to be studied)
- Say a process $P_1 | \dots | P_n$ is meant to represent the state of a system distributed on the sites $T_1, \dots, T_n$
- $S = P_1 | \dots | P_n$ at any time is called the distributed state
 - what an observer navigating in the system can observe

Actions in the distributed state

- If S moves to S' by an action a :
 - S should be observable in some way
 - a should be an action execution locally to a site T_i
 - S' is what an observer perceives if transits across T_i after a is executed
- How can we deal with communication between distributed entities?
 - see later
- Real-time, explicit location, etc..., require extensions to this framework.

Which concept of synchrony

- Until now, process execute concurrently but do not influence each other
- What is synchrony?
 - when two processes are said to synchronize?
 - when there is an event they both perceive..
- That is:
 - Process R reaches a certain state R' and waits for the event to occur
 - Process S reaches a certain state S' and waits for the event to occur
 - When both processes reached their point, they synchronize, so they can both proceed
- Synchrony makes sense only for processes staying at the same site!!

Synchrony

- Synchrony can be modelled by introducing a binary relation over actions: e.g. for some action “a” there is an action “a” related to it.
 - in $P|Q$, P executes “a” only if Q executes “a”
 - $P|Q$ globally and atomically moves to $P'|Q'$ in a silent way

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\underline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

- A useful interpretation
 - P wants to receive a stimulus by someone
 - Q sends that stimulus
 - actions τ means that $P|Q$ evolves internally

Silent actions and the environment

- Suppose we have two kinds of actions
 - interactions (with the environment), either sending and receiving stimulus
 - silent actions
- Interactions:
 - need the environment to be compliant
 - for a message to be sent, the environment should be willing to receive it
- Silent actions:
 - can be supposed to occur independently from the environment
 - in a way that is invisible to it

Finiteness

- So far, processes execute and eventually terminate
 - they are finite.
- We may want to describe processes that indefinitely have some interactive behaviour
- For instance:
 - a web server reply to requests many many times (we don't know how many)
 - possibly it can stop because of a particular interaction
- We may need to deal with infinite histories....

Operator 4: Replication

- Another extension to our algebra
 - $P ::= \dots \mid !P$
 - e.g.: $Pr = !a.a'$
 - Pr executes a , then a' , then again a and a' , and so on...
 - history $[a, a', a, a', a, a', a, a', \dots]$
- Semantics by the equivalence

$$!P \equiv P \mid !P$$

- Each time it is useful, $!P$ can be rewritten as $!P \mid P$
- Example:

$$(!a) \mid \underline{a} \mid \underline{a} \equiv (!a) \mid a \mid \underline{a} \mid \underline{a} \xrightarrow{\tau} (!a) \mid \underline{a} \equiv (!a) \mid a \mid \underline{a} \xrightarrow{\tau} (!a)$$

Operator 4b: Agent Definition

- An alternative definition of replication
 - more expressive from a programmer's viewpoint
- The definition of a process is to be associated to a number of agent definitions of the kind:
 - $A(v_1, \dots, v_n) := P$ (for some process P , possibly using $v_1, \dots, v_n$)
- Then the syntax of processes is extended by:
 - $P ::= \dots \mid A(a_1, \dots, a_n)$

$$A(a_1, \dots, a_n) \equiv P\{a_1/v_1, \dots, a_n/v_n\} \quad \text{if} \quad A(v_1, \dots, v_n) := P$$

- Example:

$$\textit{Recur}(v) := v.\textit{Recur}(v)$$

$$\textit{Recur}(a)|\underline{a}|\underline{a} \equiv a.\textit{Recur}(a)|\underline{a}|\underline{a} \xrightarrow{\tau} \textit{Recur}(a)|\underline{a} \equiv a.\textit{Recur}(a)|\underline{a} \xrightarrow{\tau} \textit{Recur}(a)$$

Compositionality

- Obtained by parallel composition and corresponding actions
- Example: consider processes

$$- P = a.a'.a''.a''', \quad Q = \underline{a}.\underline{a'}.\underline{a'''}, \quad R = \underline{a''}$$

- An evolution of P:

$$a.a'.a''.\underline{a'''} \xrightarrow{a} a'.a''.\underline{a'''} \xrightarrow{a'} a''.\underline{a'''} \xrightarrow{a''} \underline{a'''} \xrightarrow{\underline{a'''}} 0$$

- An evolution of P|Q

$$a.a'.a''.\underline{a'''} | \underline{a}.\underline{a'}.\underline{a'''} \xrightarrow{\tau} a'.a''.\underline{a'''} | \underline{a'}.\underline{a'''} \xrightarrow{\tau} a''.\underline{a'''} | \underline{a'''} \xrightarrow{\underline{a'''}} \underline{a'''} | \underline{a'''} \xrightarrow{\tau} 0$$

- P|Q|R can terminate only by silent actions

Composability and Environment

- Process P by itself only proceeds by actions a and $\bar{a}$...
 - can be interpreted as the environment of P provides and gets the requested actions
 - p executes action a , the environment provides $\bar{a}$
 - p executes action $\bar{a}$, the environment provides a
- By composing to Q , some of this environment is now specified:
 - P proceeds a bit thanks to Q , but after a while it blocks again requiring the environment
- $P|Q|R$ does not need the environment to collaborate!
- A new problem, what if the environment still provides a or a' to $P|Q|R$ while it evolves?
 - $P|Q|R$ does no longer reach termination by itself

Operator 5: Name restriction

- By name restriction you can define an action that is private to a given process... technically, the environment cannot interact by “a”
- Extending the algebra:
 - $P ::= \dots \mid (a)P$
 - e.g.: $P = (a)a.a'$, in P a is private, a' is public
- which semantics?
 - has to do with renaming of actions....
 - technical argument
- Two ideas:
 - You can always rename a private action of a process
 - Processes with disjoint private actions can be composed without problems
- Uses the concept of free and bound actions
 - free action: a public one, without a restriction outside
 - bound action: a private one

Semantics of name restriction

$$\begin{aligned}
 (a)0 &\equiv 0 \\
 (a)(b)P &\equiv (b)(a)P \\
 ((a)P) + Q &\equiv (a)(P + Q) \quad \text{if } a \notin FV(Q) \\
 ((a)P)|Q &\equiv (a)(P|Q) \quad \text{if } a \notin FV(Q)
 \end{aligned}$$

$$P|Q \equiv P\{b/a\}|Q \quad \text{if } b \notin FV(P|Q) \wedge a \notin FV(P)$$

- Allows you to forget about the order of restrictions and about zero
- Says you can extend restriction if no conflicts occur
- In the case of conflicts you can rename an action

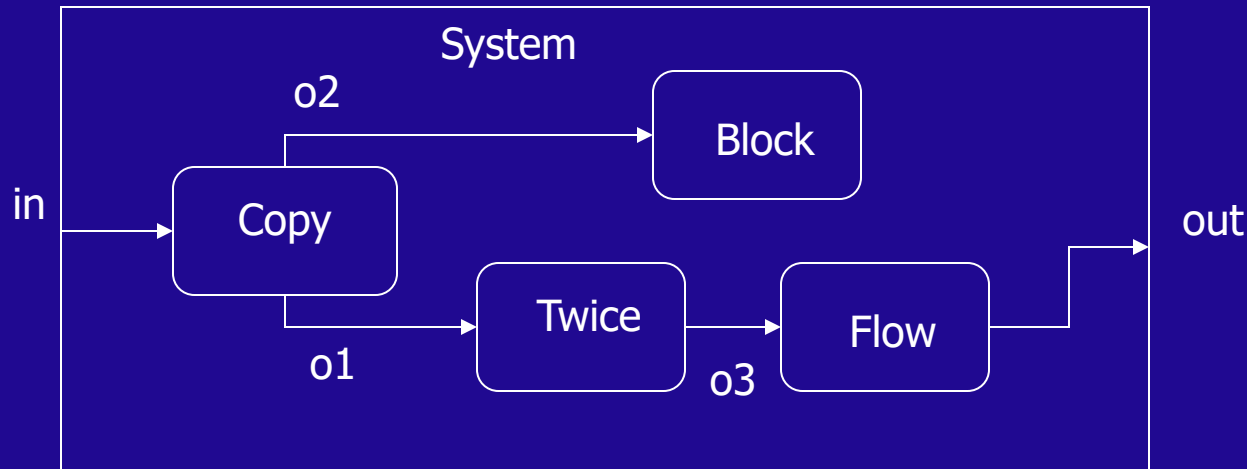
$$((a)(a.a'|\underline{a}.a''))|\underline{a'}.a'' \xrightarrow{\tau} ((a)(a'|a''))|\underline{a'}.a'' \equiv (a)(a'|a''|\underline{a'}.a'') \xrightarrow{\tau} \xrightarrow{\tau} (a)0 \equiv 0$$

$$((a)(a|\underline{a}|a'))|a|\underline{a'} \equiv ((b)(b|\underline{b}|a'))|a|\underline{a'} \equiv (b)(b|\underline{b}|a'|a|\underline{a'})$$

Putting things together

- We defined a number of concepts
- We can use the process algebra so far described to represent the architecture of a system
 - system composed of interconnected subparts
 - each interacting with each other by sending and receiving stimulus (say, messages without an actual meaning)
- and to represent how it evolves as interactions occur

A complete example



$$\begin{aligned}
 Twice(in, out) &:= \underline{\underline{!in.out.out}} \\
 Copy(in, out_1, out_2) &:= \underline{\underline{!in.out_1.out_2}} \\
 Block(in) &:= \underline{\underline{!in}} \\
 Flow(in, out) &:= \underline{\underline{!in.out}} \\
 System(in, out) &:= (o1)(o2)(o3)(\\
 &\quad Copy(in, o1, o2)|Twice(o1, o3)|Flow(o3, out)|Block(o2))
 \end{aligned}$$

$$System(in, out) \mid \underline{\underline{in}} \xrightarrow{\tau^*} System(in, out) \mid \underline{\underline{out}} \mid \underline{\underline{out}}$$

Value passing

- We also want to deal with communication of values, not just stimulus
- To do that, we first consider actions as being of three kinds
 - $x(v)$, receiving a value from channel named “ x ”, which is stored in variable “ v ”
 - (similar to an argument of a method...)
 - $\underline{x}v$, sending value “ v ” from channel named “ x ”
 - (similar to a method invocation)
 - τ as usual

Semantics

- Operational rule:

$$\frac{P \xrightarrow{xw} P' \quad Q \xrightarrow{x(v)} Q'}{P|Q \xrightarrow{\tau} P'|(Q'\{w/v\})}$$

- Example:

$$x(v).\underline{y}v|\underline{x}5 \xrightarrow{\tau} \underline{y}5$$

How can we deal with asynchrony

- Say that we want to realise a communication between processes located at different sites
- Each single action should involve only a process situated into a local site
- Idea, breaking a communication in two parts
 - first, emitting the message, consumed by an intermediate process
 - later, that process sends the message to the receiver
 - as if the intermediate process had virtually moved from one site to the other
 - allows for representing lost of messages, decoupling, unordering,..
- Interpretation:
 - the intermediate as the communication infrastructure
 - the intermediate as the message itself (as an entity)
 - the sender not interested in any reply

Asynchrony

- Instead of
 - $P = \underline{x}v.Q, \quad R = x(w).S$
 - so that $P|R \rightarrow Q|S\{v/w\}$
- Doing
 - $P = (\underline{x}v|Q), \quad R = x(w).S$
 - so that $P|R = \underline{x}v|Q|R \rightarrow Q|S\{v/w\}$
- $\underline{x}v$ represents the message itself

CCS (Calculus for Communicating Systems)

- Turing Award for Robin Milner (1982...)

$$P ::= 0 \mid \tau.P \mid a(v).P \mid \bar{a}v.P \mid (P|P) \mid (P + P) \mid P/a \mid !P$$

- Features:
 - synchronous communications between processes
 - compositionality, non-determinism by choices, replication
- Fully exploited to verify properties of interactive systems, protocols..
- Is not Turing-complete.... cannot represent recursion....
- What do we still lack?
 - interconnection of processes are fixed
 - how do we deal with mobility (physical or virtual)?

Pi-calculus

- Is a relatively simple modification to CCS
 - structures of processes may change
 - allows to represent any algorithm (Turing-complete)
 - it's the candidate for being the standard foundation of interactive languages
- Basic idea:
 - We do not have values to be communicated
 - What processes exchanges are just channel names...
 - who receives it, has a new channel to communicate to
 - when the name is no more used.. the link disappears
- Syntax and semantics?
 - basically similar to CCS (only technical differences)

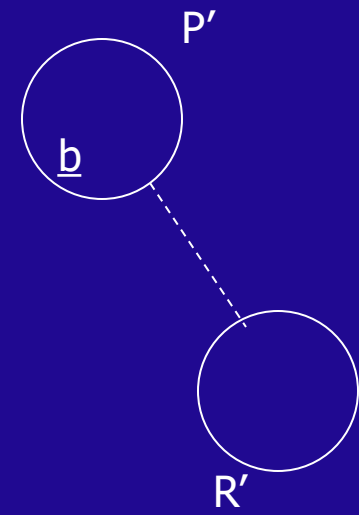
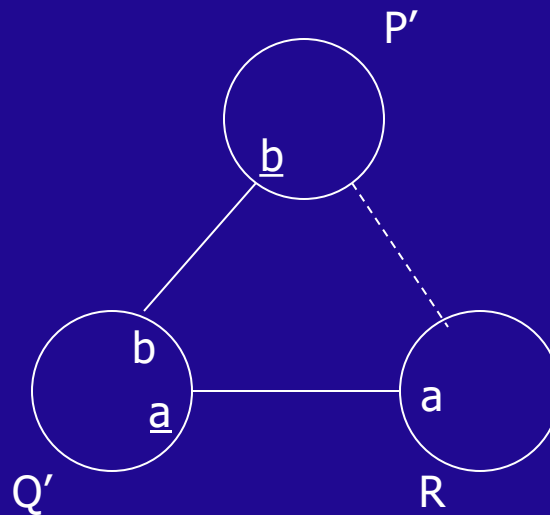
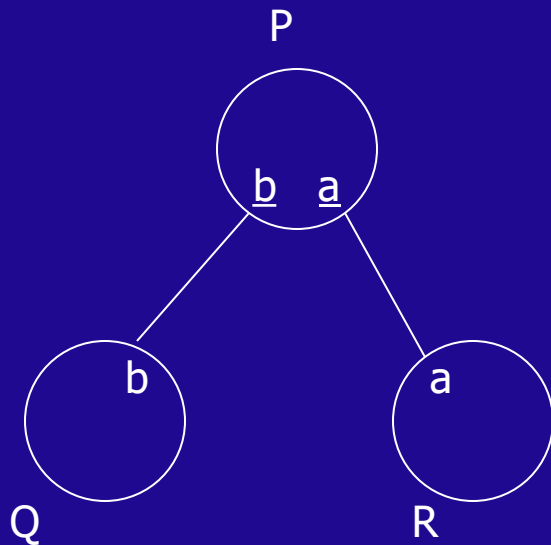
Process Mobility

$$P := \bar{b}a.\bar{b}5.P'$$

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

$$R := a(x).R'$$

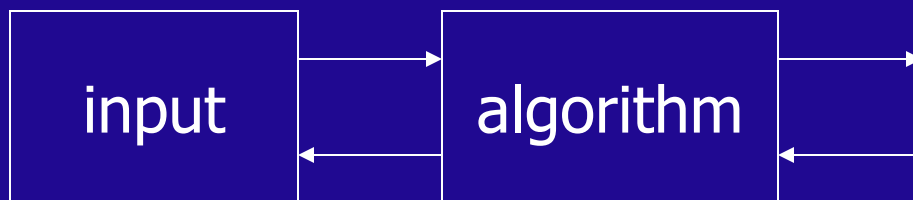
$$System := P \mid Q \mid R$$

$$System \xrightarrow{\tau} \xrightarrow{\tau} P' \mid \bar{a}5.0 \mid R \xrightarrow{\tau} P' \mid R'$$


Defining algorithms with Pi

In the pi-calculus:

- data structures can be represented as processes at given ports, that exhibit an interactive behaviour
- an algorithm is realised by a process wrapping a data structure
- reproducing the interactive behaviour of the intended output by exploiting the input's one



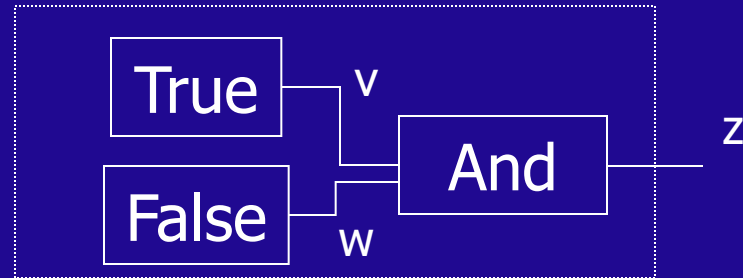
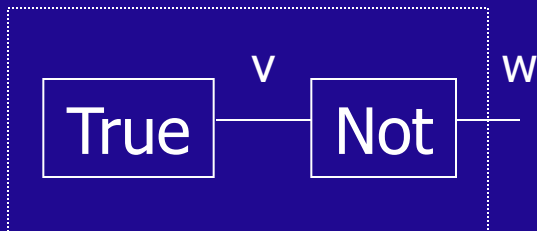
Logics in Pi

- Boolean values

- $\text{True}(x) := !x(t).x(f).\underline{t}$ ($\underline{t}$ is the same as $\underline{t}_\varepsilon$)
- $\text{False}(x) := !x(t).x(f).\underline{f}$

- Operations:

- $\text{Not}(x,y) := !x(t).x(f).\underline{y}f.\underline{y}t$
- $(v)(\text{True}(v)|\text{Not}(v,w)) \approx \text{False}(w)$
- $\text{And}(x,y,z) := (c)(d)z(a).z(b).\underline{x}c.\underline{x}d.(d.\underline{b}+c.(\underline{y}a.\underline{y}b))$
- $(v)(w)(\text{True}(v)|\text{False}(w)|\text{And}(v,w,z)) \approx \text{False}(z)$



Other encodings?

- Mathematics
 - numbers as processes with channels (s,z)
 - Succ, Sum, Prod, Factorial, as processes
- Aggregations:
 - composition: by connecting processes
 - recursion: by replication and by topology changes
 - partiality: by recursions that do not terminate
- Hence... π is Turing-complete
 - it can be used to represent any recursive partial function
 - it can represent any algorithm
 - it also represents interactions and composition

Isn't it too big?

$$P ::= 0 \mid \tau.P \mid a(v).P \mid \bar{a}v.P \mid (P|P) \mid (P + P) \mid P/a \mid !P$$

...at least, with respect to lambda-calculus...

- Does it capture interaction at the core level?
- There are studies that make it simpler
 - only asynchronous communications
 - only sending new names
 - no name restriction, no choice
- ai-pi-calculus

$$P ::= \bar{x}(y) \mid x(y).P \mid (P|P) \mid !P$$

Which applications?

- Similar to lambda-calculus for languages
 - providing a foundation to languages with interactive abilities
 - reasoning about interactive programs
 - ...
- Really useful on systems
 - describing the behaviour of interactive systems
 - proving properties of interest on protocols
 - ...
- Typically:
 - not encoding things in Π (useful if you use verifiers...)
 - but writing our own language using similar features...

The semantics of Linda

- Linda is a coordination model based on tuple-spaces...
- Linda can be seen as a language
 - for building the interactive part of systems
- A coordinated system:
 - processes performing primitives $\text{out}(x)$, $\text{rd}(x)$, $\text{in}(x)$
 - one tuple-space where tuples are put and dropped...
- A coordinated system can be seen as a process algebra,
 - its evolutions described by an operational semantics based on transition systems

Syntax

Processes: e.g. $\text{out}(x).\text{rd}(y).\text{in}(z).0$

$$\begin{array}{lcl} \pi & ::= & out \mid in \mid rd \\ \alpha & ::= & \pi(x) \\ P & ::= & 0 \mid \alpha.P \end{array}$$

Systems: e.g. $\text{out}(x).\text{rd}(z) \parallel y \parallel z \parallel \text{in}(y)$

$$S ::= 0 \mid x \mid P \mid S \parallel S$$

$$\begin{array}{l} 0 \parallel S \equiv S \quad S \parallel S' \equiv S' \parallel S \\ (S \parallel S') \parallel S'' \equiv S \parallel (S' \parallel S'') \end{array}$$

Semantics

$$out(x).P \parallel S \longrightarrow P \parallel S \parallel x$$

$$rd(x).P \parallel S \parallel x \longrightarrow P \parallel S \parallel x$$

$$in(x).P \parallel S \parallel x \longrightarrow P \parallel S$$

- $out(x)$, puts x in the dataspace and let P to carry on
- $rd(x)$, if x occurs in the dataspace, P can carry on
- $in(x)$, removes x and then lets P to carry on
- a lot of things borrowed from pi-calculus...

What do we do in coordination?

- Many many applications and infrastructures
- Some semantic study
 - Exploring the framework “Coordination as a Service”
 - the tuple space as an interactive abstraction...
 - Which expressiveness?
 - Which engineering methodology?
 - Encoding more complex coordination models (e.g. LogOp) on top of lower-level abstractions
 - ReSpecT tuple centres...
 - How to preserve the intended semantics...

Some References

- On interaction:

Peter Wegner, Why Interaction Is More Powerful Than Algorithms,
Communications of the ACM, May 1997

<http://www.cs.brown.edu/people/pw/papers/ficacm.ps>

- On pi-calculus:

- R.Milner, “Communicating and Mobile Systems: the pi-calculus”,
Cambridge University Press

- On process-algebras:

- Bergstra, Ponse, Smolka, “Handbook of Process Algebra”, North-Holland

- Bibliography on pi-calculus:

- <http://liinwww.ira.uka.de/bibliography/Theory/pi.html>