ISO/IEC JTC1/SC21 Meeting - Florence, November 6th, 1989

An aerial view on LOTOS

(IS 8807)

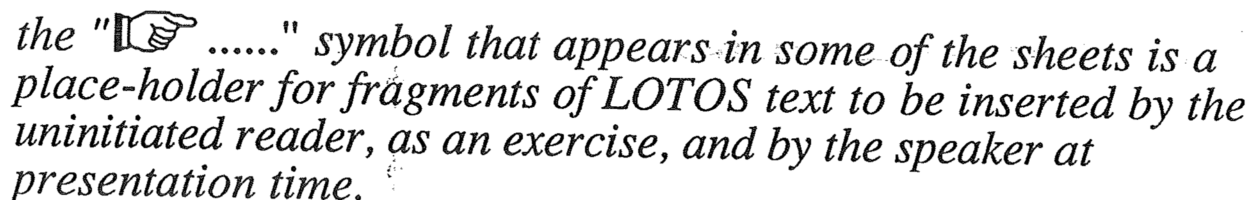
Tommaso Bolognesi

UNI/UNIPREA
C.N.R./CNUCE - Pisa

Contents

1.	Four general questions on LOTOS	4
2.	The "mental landscape" of the LOTOS user	13
3.	The two components of the language	3
4.	Defining abstract data types and expressing data values	3
5.	Defining processes and expressing their behaviours	2
6.	An example: specification of a switching node	5
7.	Existing LOTOS specifications and tools	4
		34

Warning:


the "☞ ......" symbol that appears in some of the sheets is a place-holder for fragments of LOTOS text to be inserted by the uninitiated reader, as an exercise, and by the speaker at presentation time.

1. Four general questions on LOTOS

Where does it stand as a specification language for concurrent / distributed systems ?

- Languages based on the model of *communicating, extended* Finite State Machines (FSM)

    - SDL (CCITT)
    - Estelle (ISO)

    ...

- Petri nets

    - condition / event
    - place/transition
    - with predicates
    - timed

    ...

- Process algebras

    - CCS (Milner)
    - CSP (Hoare)
    - LOTOS (ISO)
      Language Of
      Temporal Ordering Specification

- Logics

    - Modal
    - Temporal (linear-time, branching-time)

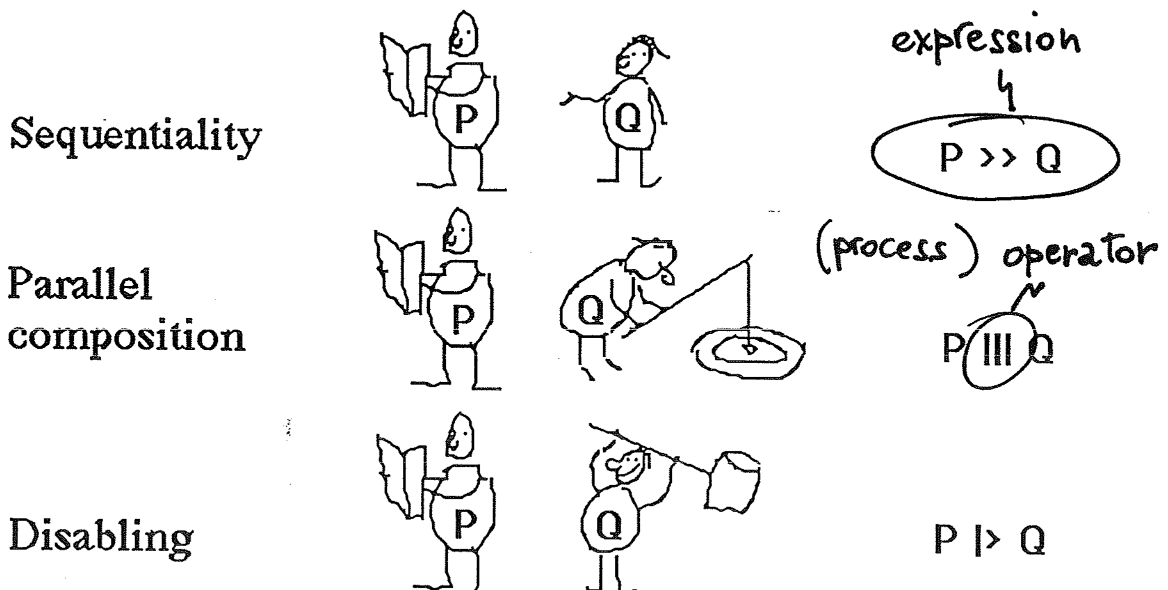
    ...

Why "algebraic" ?

LOTOS offers special *algebraic operators*
for building *algebraic expressions*
that describe:

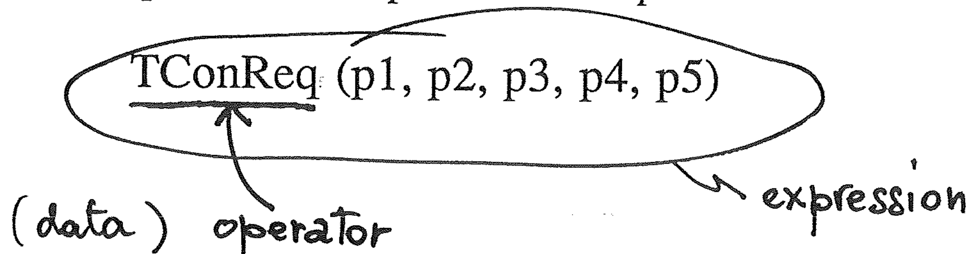
- **the behaviour of processes**

  Examples:



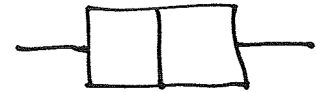
- **the data values / structures handled by the processes**

  Example: A Transport Service primitive

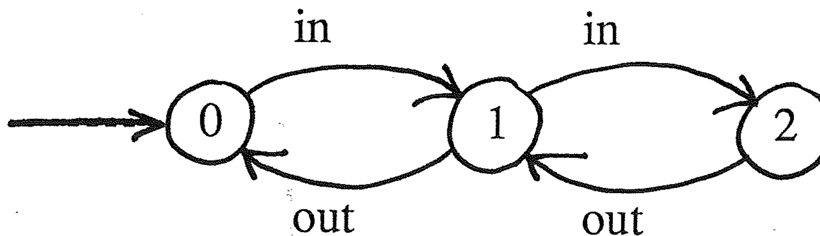


Should one forget about FSM's ? (no)

Example: design of a buffer of capacity 2



- **as a FSM:**



- **in LOTOS:**

```
        Buff-2[inp, out](0)

where
        process Buff-2[inp, out](s : state)  :  noexit  :=

                [s = 0] -->                  in;   Buff-2[inp, out](1)

        []      [s = 1] -->          (       in;   Buff-2[inp, out](2)
                                     []      out;  Buff-2[inp, out](0)
                                     )
        []      [s = 2] -->                  out;  Buff-2[inp, out](1)

        endproc
```

Which advantages in using algebraic expressions ?

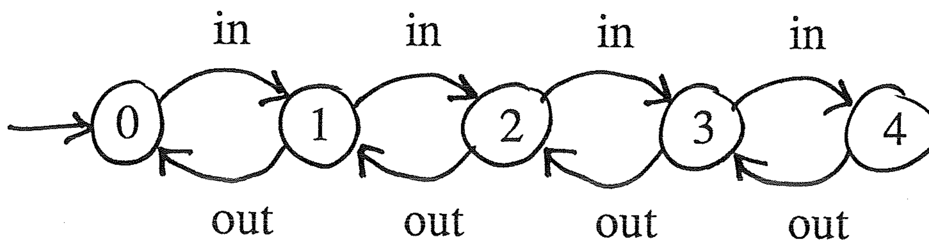
Modularity:

1 FSM	+	1 FSM	=	2	FSM's
1 Petri Net	+	1 Petri Net	=	2	Petri Nets
1 expressions	+	1 expression	=	**1**	expression

Expressions can be composed, by operators such as >>, |||, [>, thus obtaining more complex expressions

Example: design of a buffer of capacity 4

- **as a FSM** (re-designed from scratch)



- **as a LOTOS expression** (using process Buff-2)

Buff-2[inp, mid] |[mid]| Buff-2[mid, out]

2. The "mental landscape" of the LOTOS user

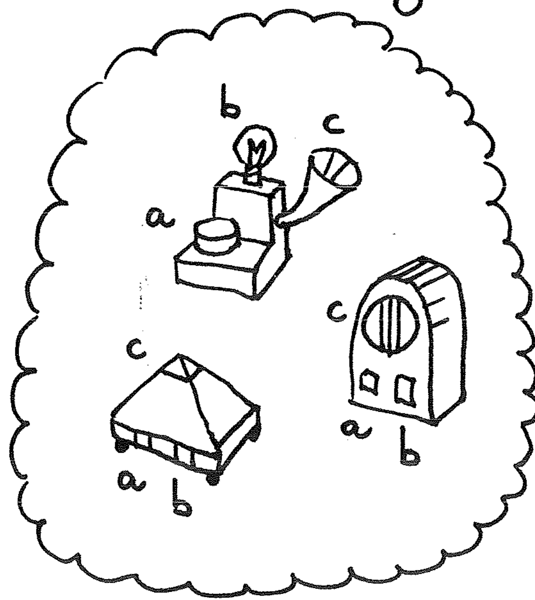
...or:

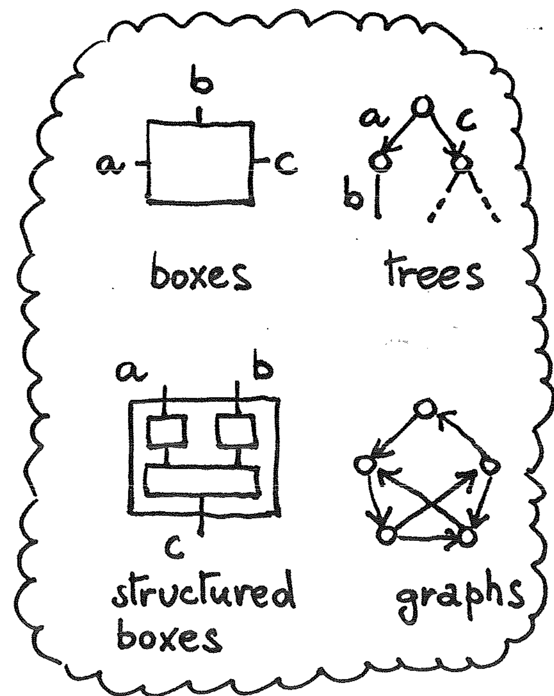
- what is inside the specifier's mind when he starts writing ?
- what is the LOTOS-oriented view of a system ?
- what can be easily and directly expressed in LOTOS ?

birth of idea

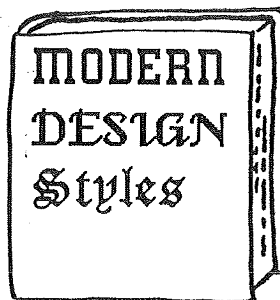
I want to design a photo-acoustic psycho-tester

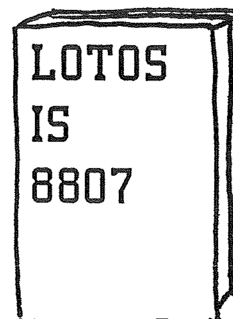
two alternative mental landscapes

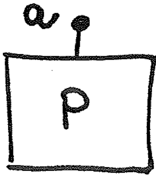




adequate manual for the design



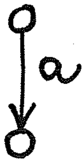




The behaviour of box/process P manifests itself at gate a, by 'a-events'

(boxes...)

process P[a] := <behaviour expression> endproc

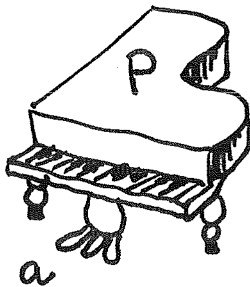


event 'a' occurs once

(...trees...)

a; stop

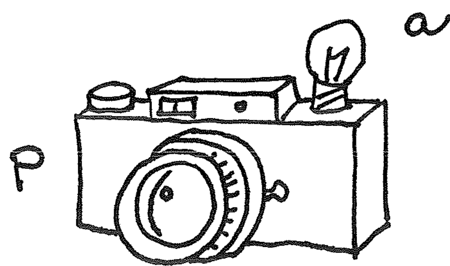
Example



One can successfully push pedal 'a' of piano P.

(experiment, interaction)

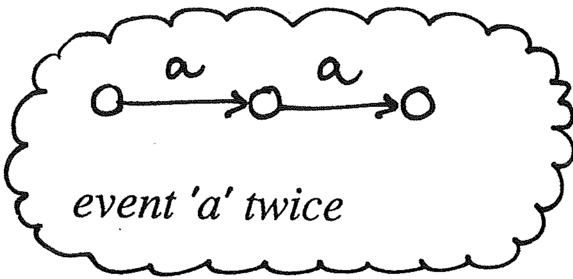
Example



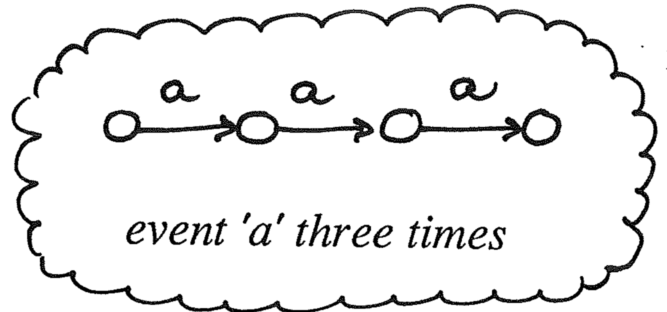
Camera P can flash once light 'a'.

(observation)

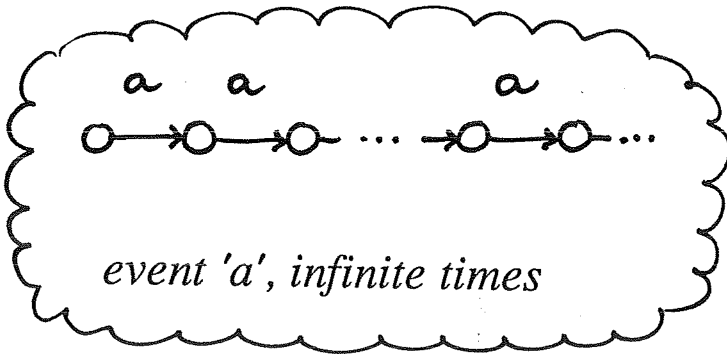
Further possible behaviours for P[a].



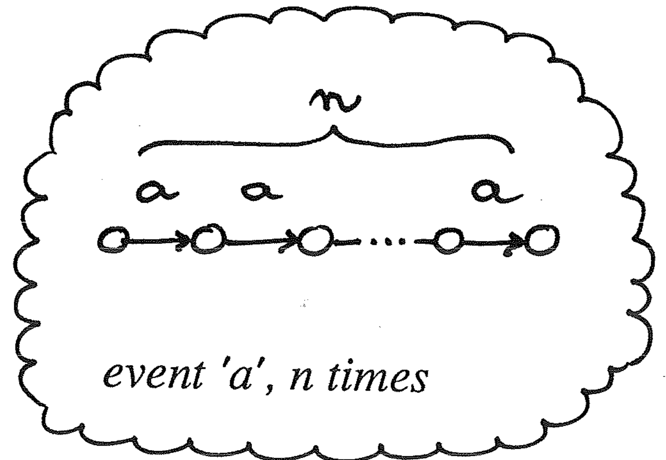
a; a; stop



☞ ......



☞ ......



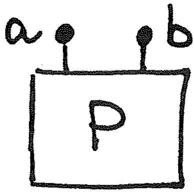
P[a](n)

where

process P[a](k : nat)

[k ne 0] --> a; P[a](k-1)

endproc

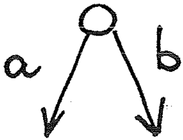


The behaviour of box/process P manifests itself at gates a and b

process P[a, b] := <beh. expr.> endproc

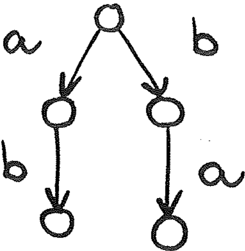
first event a, then event b

......



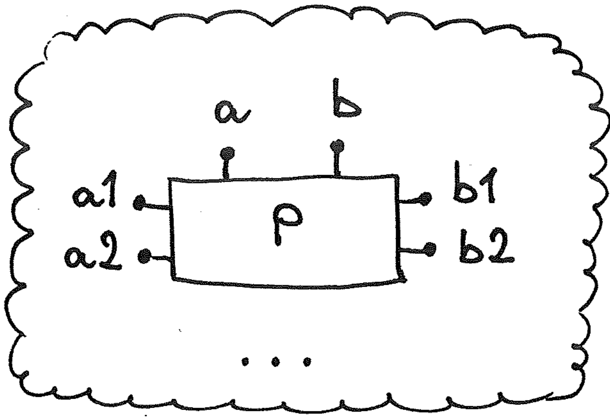
choice between a and b

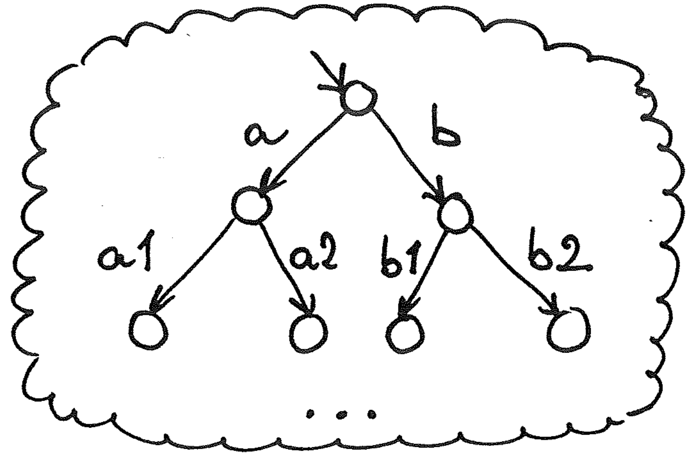
a; stop [] b; stop



a and b, in either order

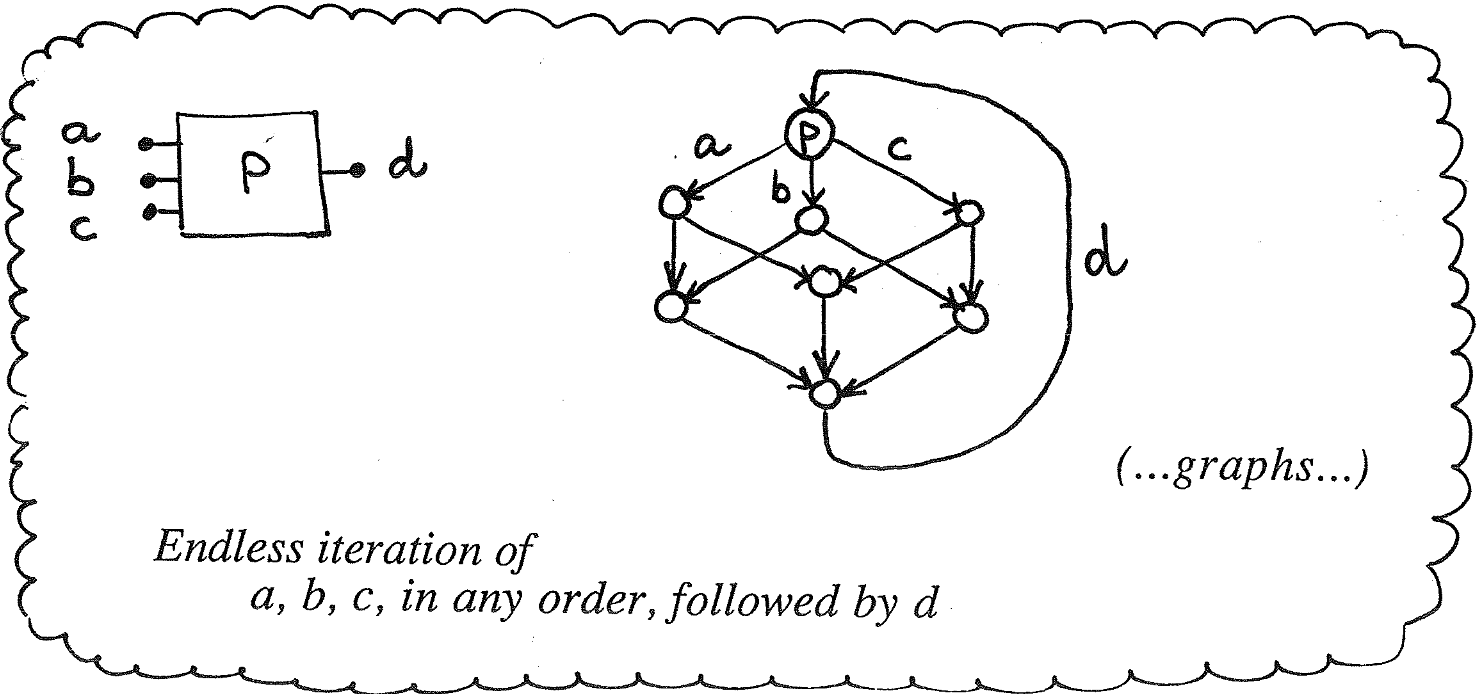
a; stop ||| b; stop





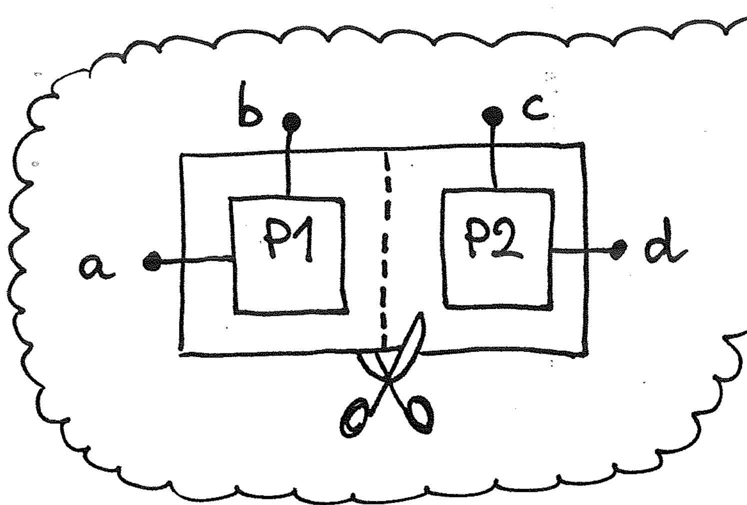
......

......



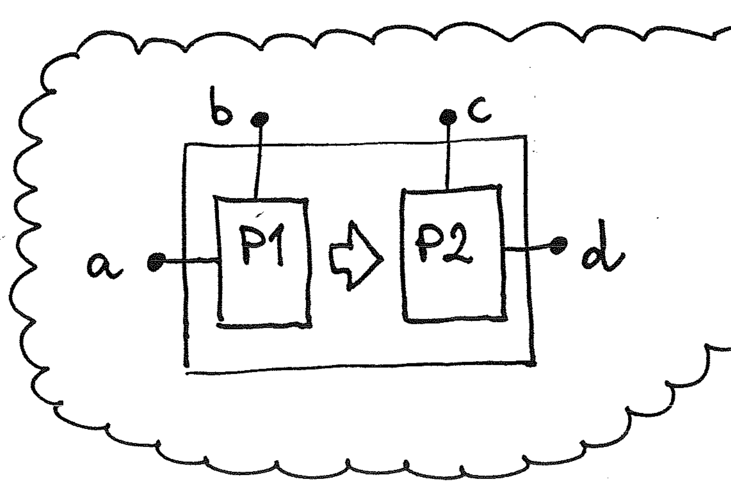

```
process P[a, b, c, d] : noexit :=

        (a; exit ||| b; exit ||| c; exit)    >>
        d ;
        P[a, b, c, d]
```

(...structured boxes...)



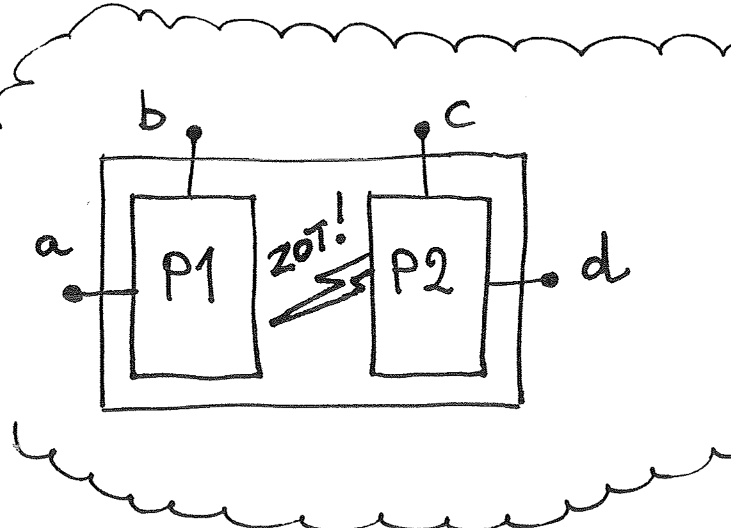
either behaviour P1 at a and b
or behaviour P2 at c and d

P1[a, b] [] P2[c, d]



first behaviour P1 at a and b
then behaviour P2 at c and d

☞ ......

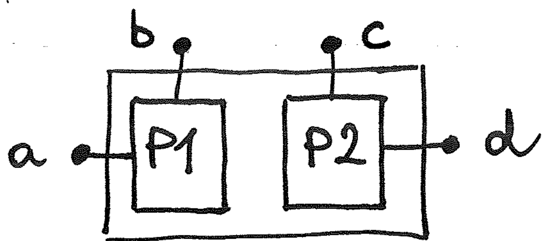


behaviour P1

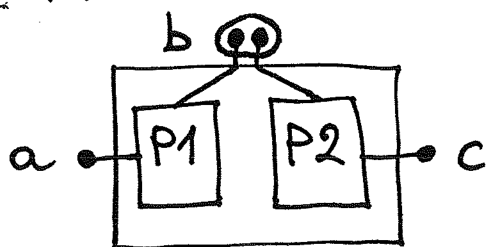
can be interrupted at any time by

behaviour P2

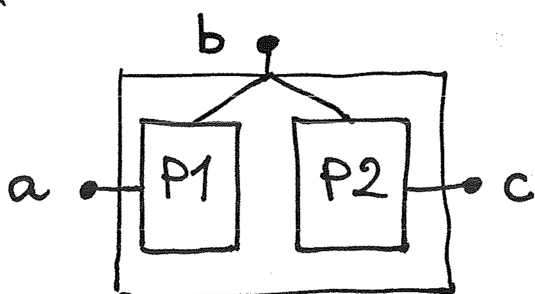
P1[a, b] [> P2[c, d]



behaviour P1 at a, b,
and
behaviour P2 at c, d
take place
independently of each other

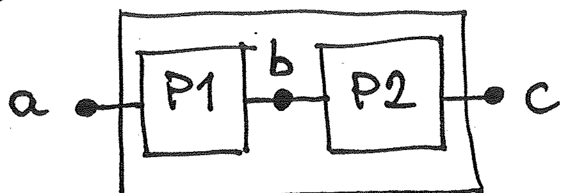


behaviour P1 at a, b,
and
behaviour P2 at b, c
take place
independently of each other



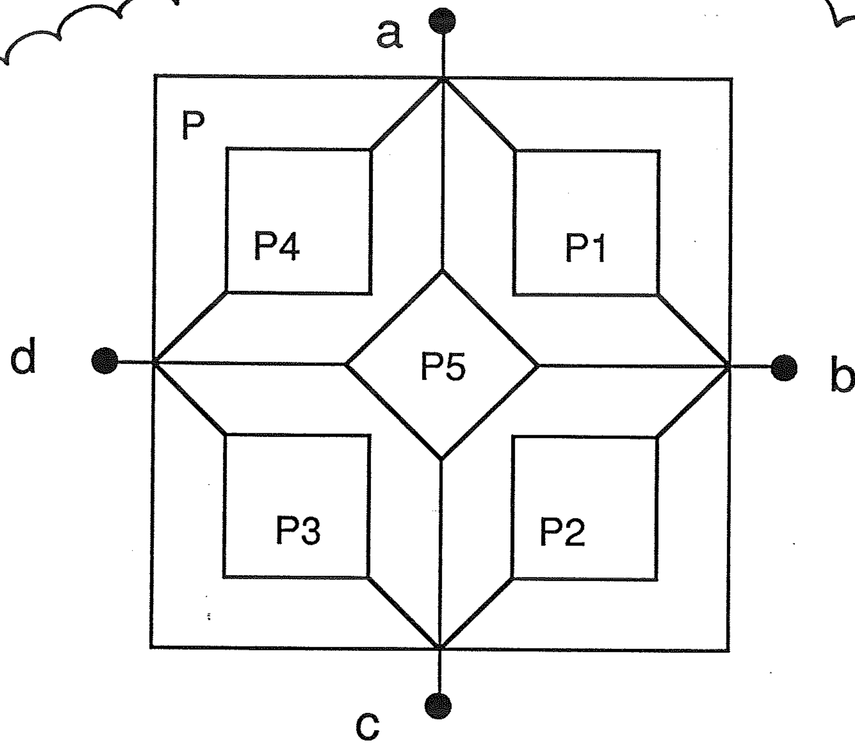
behaviour P1 at a, b,
and
behaviour P2 at b, c
take place
independently of each other
except that they
***synchronize** at gate b*

P1[a, b] |b]| P2[b, c]



As above,
furthermore gate 'b' is
***hidden**, that is,*
not available for synchron.
with external behaviours

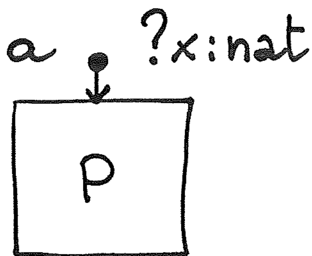
hide b in
P1[a, b] |b]| P2[b, c]



The behaviour of P at a, b, c, d,
is determined and constrained by the behaviours of

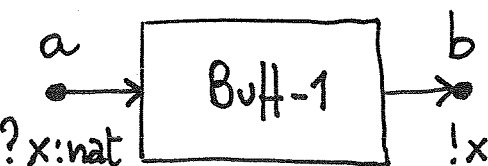
P1 at a, b,
P2 at b, c,
P3 at c, d,
P4 at d, a,
and P5 at a, b, c, d.

```
          (
(P4[d, a]   |[a]|   P1[a, b])
          |[d, b]|
(P3[d, c]   |[a]|   P2[c, b])
          )

       |[a, b, c, d]|

       P5[a, b, c, d]
```



P accepts (inputs) at gate 'a' any natural number x

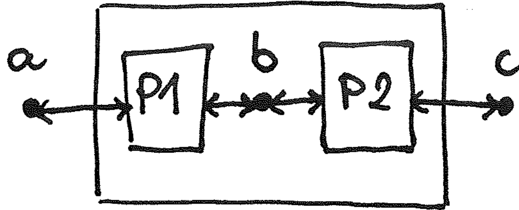
- *once* — a ?x : nat; stop
- *twice* — ......
- *infinite times* — ......
- *n times* — ......



first a accepts any natural x,
then b offers that x,
and this happens all the time.

Note: this is a buffer of capacity 1.

```
process Buff-1[a, b] : noexit :=

    a ?x : nat;     b !x;     Buff-1[a, b]

endproc
```

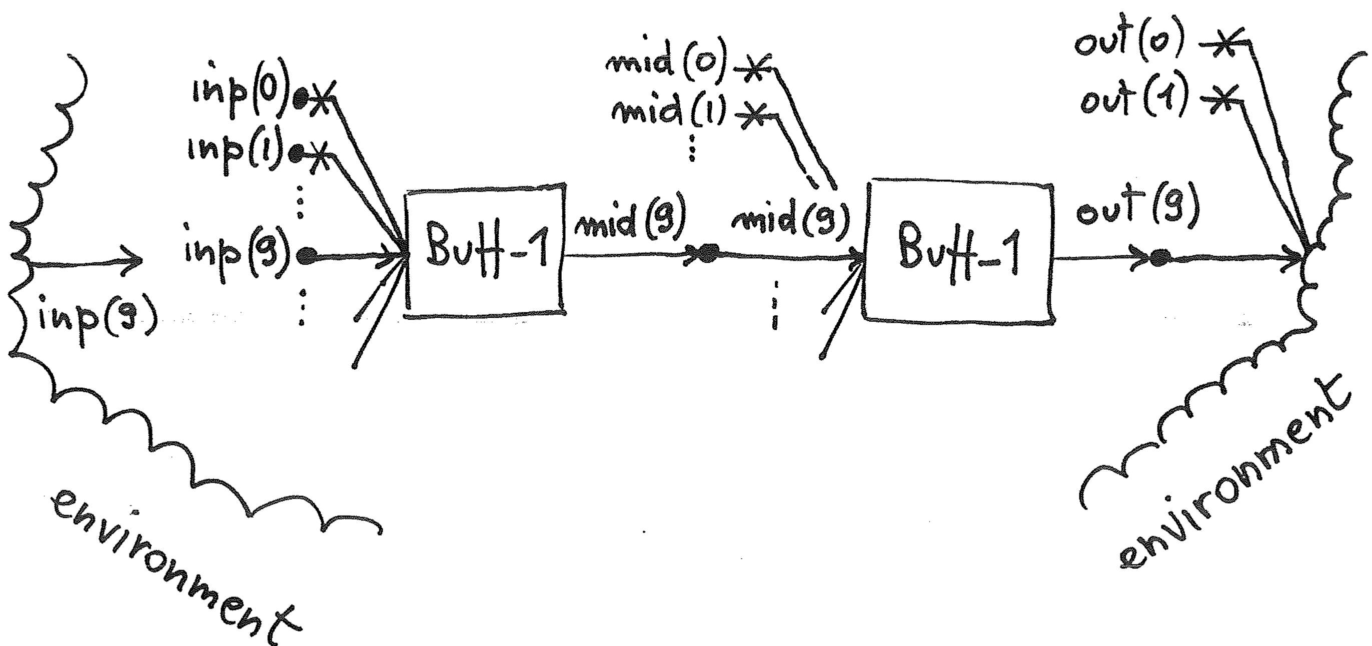


P1 may exchange data at a and b;
P2 may exchange data at b and c;
P1 and P2 act independently,
but must synchronize at b,
which is hidden to anybody else.

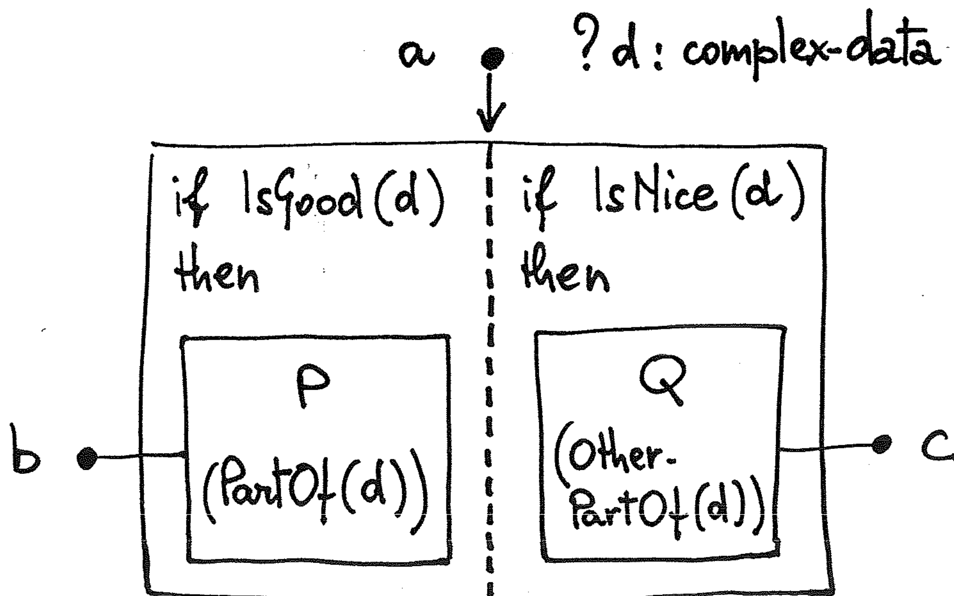
hide b in P1[a, b] |[b]| P2[b, c]

Note: we may build a Buff-2 out of two instances of Buff-1, as an alternative to the state-oriented Buff-2 of p. 1.3 (here we also specify data):

```
process Buff-2[inp, out] : noexit :=

    hide mid in
        Buff-1[inp, mid]  |[mid]|  Buff-1[mid, out]

endproc
```



A complex data d is accepted at a.
Then, depending on its qualities
either P takes place,
which processes a part of d,
or Q takes place,
which handles another part of d.



a ?d : complex-data;

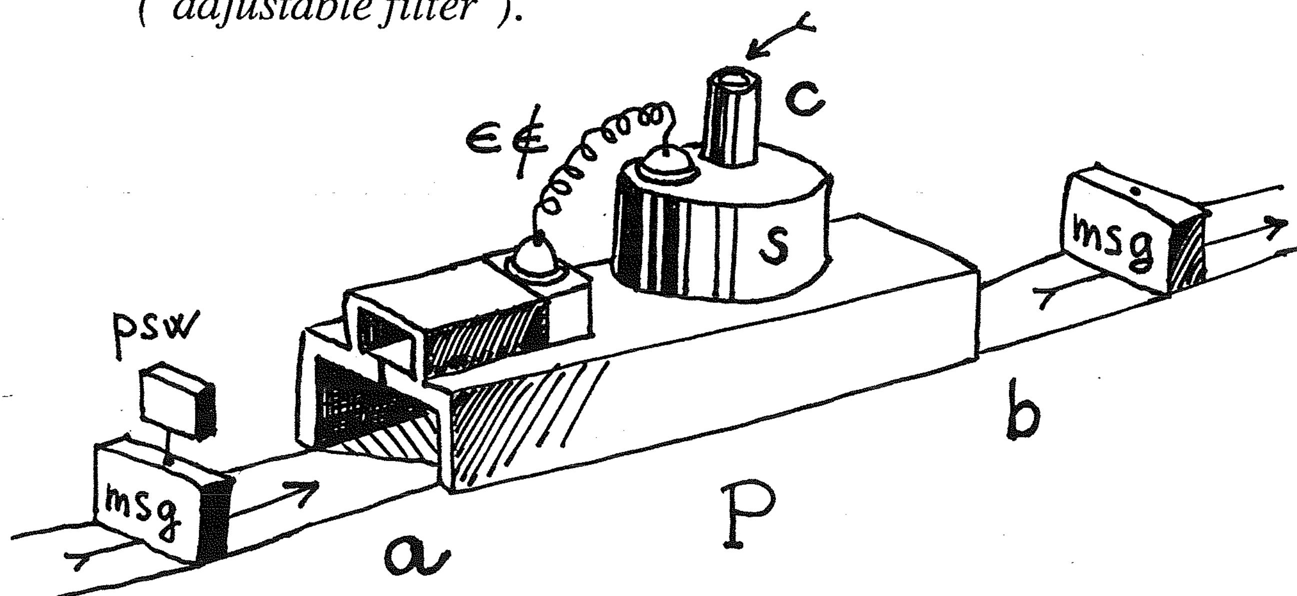
```
     [IsGood(d)]    ---> P[b] (PartOf(d))
[]   [IsNice(d)]    ---> Q[c] (OtherPartOf(d))
```

complex-data
IsGood
IsNice
PartOf
OtherPartOf

} are user-defined

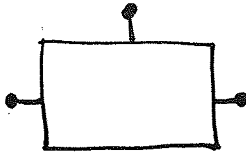
(...managers of password-sets...)

P maintains a deposit of passwords,
by inserting or deleting elements as requested at c;
furthermore, it accepts pairs message-password at a:
the message is offered at b only if its password is legal
("adjustable filter").

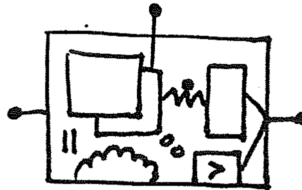



```
process P[a, b, c] (s : password-set) : noexit        :=

            c !INSERTION ?psw : password;
                                    P[a, b, c] (insert(psw, s))

      []    c !REMOVAL  ?psw : password;
                                    P[a, b, c] (remove(psw, s))

      []    a ?msg:message ?psw:password;
                  (  [psw IsIn s]-->  b !msg;
                                    P[a, b, c] (s)
                  [] [psw NotIn s]--> P[a, b, c] (s)
                  )

endproc
```

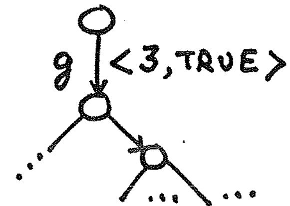
In conclusion, what is the LOTOS-oriented view of a system ?



- A system is a *process*, able to *interact* (= synchronize + exchange data) with its *environment* via *gates*.



- A process can be structured as a collection of processes which are combined in various ways (in particular, they are composed in parallel for interacting with one-another)



- The behaviour of a process can be seen as an 'action tree', possibly of infinite depth and branching, whose arcs are labelled by **observable (inter-)actions:**

  <gate-name> <list-of-data-values>
  (example: g <3, TRUE>)

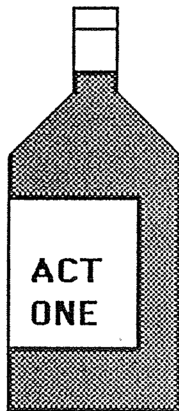
  or by the **unobservable action** 'i' (not discussed).

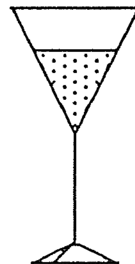
3. The two components of the language

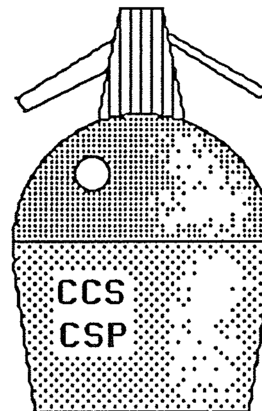
A LOTOS specification has two components

Definitions of
Abstract Data Types (ADTs)

Definitions of
processes / behaviours







Abstract data types(ADT)

Processes

value expression:

express WHICH values
are handled / exchanged
by processes

push((x + 1), stack)

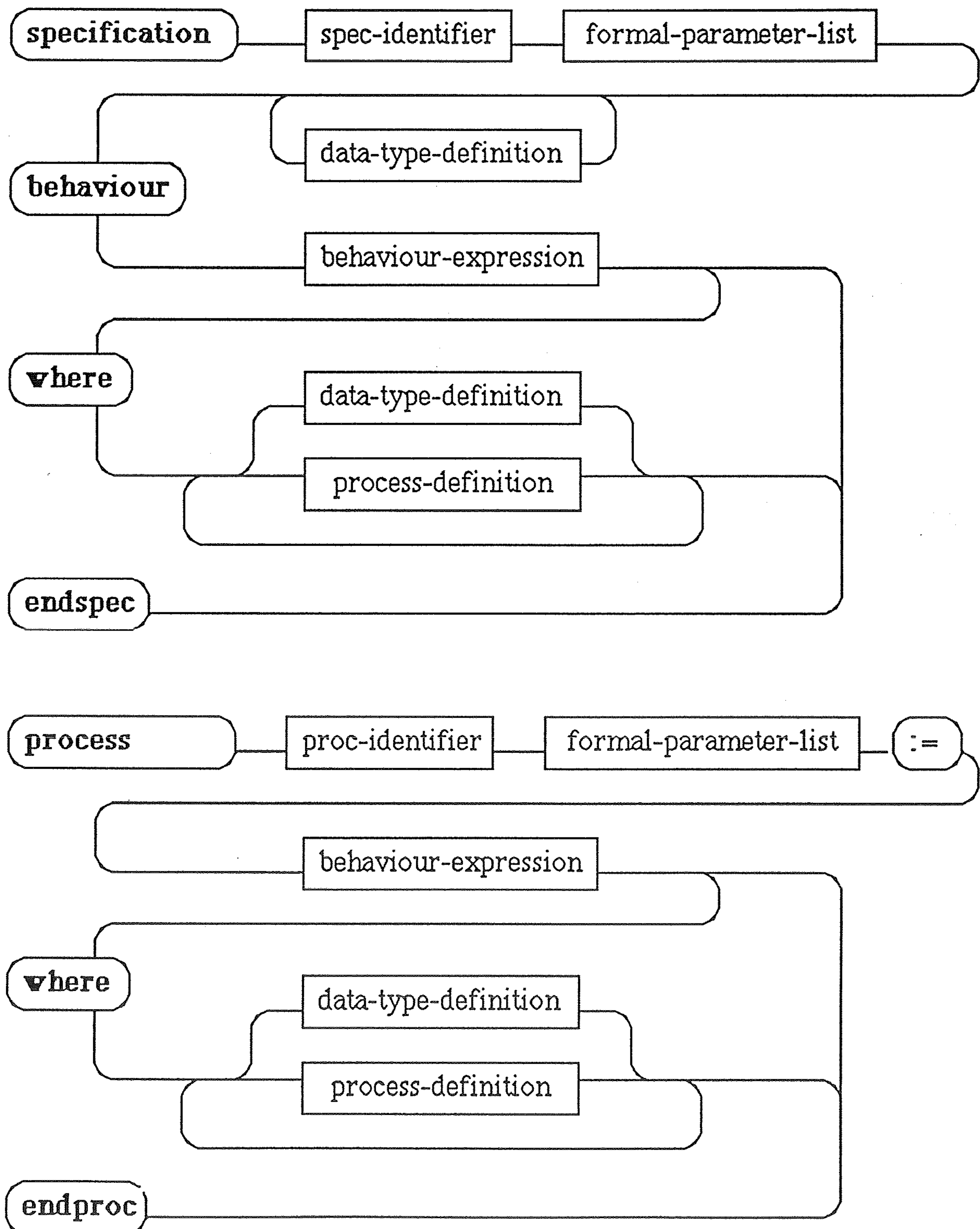
behaviour expression:

express WHEN/WHERE
does the interaction occur

either explicitly
a; b; stop

or implicitly
P[a,b] ||| Q[a,b]

Syntactic interplay of the two components



Value expressions may appear within *behaviour expressions* in four different places, for expressing:

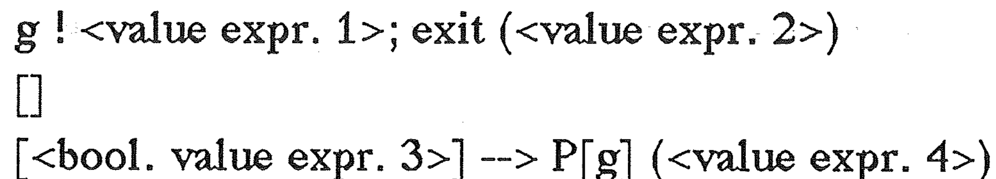
1) *values offered* at a gate
(<value expr. 1>);

2) *values offered* at the special 'successful termination' gate
(<value expr. 2>);

3) *conditions* for a behaviour to take place
(<bool. value expr. 3>);

4) *actual values* for instantiating a parametric process
(<value expr. 4>).

<behaviour expression>



- ***behaviour expressions*** are built up with
  - LOTOS predefined operators (e.g.: '[]');

- ***value expressions*** are built up with
  - user-defined operators, and
  - LOTOS predefined operators.

4. Defining abstract data types and expressing data values

- *Data type definitions* provide the ***syntax*** and the ***semantics*** of the *value expressions* to be used within *behaviour expressions*.

- *User-defined* data types appear within an actual LOTOS spec.;

- *Standard* data types appear in the standard library of data types, in IS8807, and can be referenced by an actual LOTOS spec.

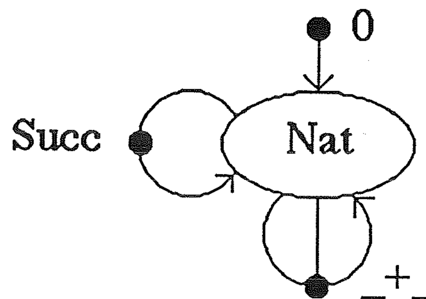
Example of *data type definition*

```
type  VeryBasicNaturalNumber  is
   sorts   Nat
   opns    0      : --> Nat
           Succ   : Nat --> Nat
           _+_    : Nat, Nat --> Nat
   eqns  forall  m, n : Nat
         ofsort  Nat
           m + 0          = m
           m + Succ(n) = Succ(m) + n
endtype
```

the **signature** defines the **syntax** of value expressions

the **equations** define the **semantics** of value expressions

- Graphic representation of the signature:



- Some correct *value expressions* of sort 'Nat':

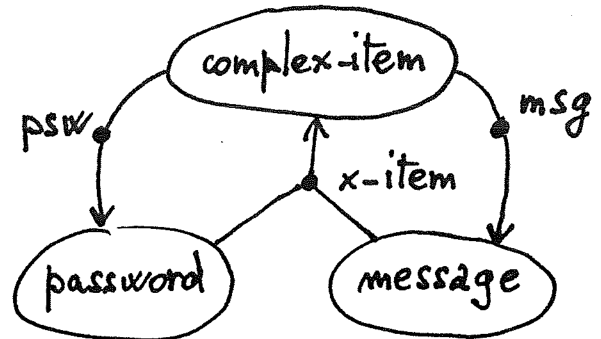
  0
  Succ(0)
  0 + Succ(0)
  ...

- Two value expressions of sort 'Nat' with the same semantics (one can be transformed into the other by applying both equations once, as rewrite rules)

0 + Succ(0) = Succ(0)

Another example of *data type definition*

(for the 'adjustable filter' of p. 2.12)



type complex-item **is**

sorts
complex-item, password, message

opns
(* constructor : *)
x-item: password, message --> complex-item

(* selector *)
psw: complex-item --> password

(* selector *)
msg: complex-item --> message

eqns
forall p: password, m: message

of sort password
psw(x-item(p, m)) = p

of sort message
msg(x-item(p, m)) = m
...

endtype

5. Defining processes and expressing their behaviours

- A *process definition* defines the *temporal ordering* of the interactions in which the process can engage at its gates.

- All processes are defined by the specifier (no standard library).

- Syntax of a process definition:

process <name> [<gate-list>]
(<parameters>) : <functionality> :=

<behaviour-expression>

where

<process-definition>'s and/or
<data-type-definition>'s

endproc

Fundamental behaviour expressions

name	*syntax*
inaction	**stop**
action prefix, *possibly with selection predicate*	g ?x : s !E *[E1 = E2]*; B
guard	[E1 = E2] ---> B
choice	B1 [] B2
successful termination *possibly with value passing*	**exit** *(s_1, ..., s_n)*
enabling *possibly with value passing*	B1 >> **accept** x_1:s_1...,x_n:s_n **in** B2
disabling	B1 [> B2
parallel composition	B1 |[g_1, ..., g_n]| B2
• full synchron.	B1 || B2
• pure interleaving	B1 ||| B2
process instantiation	P [g_1, ..., g_n] (E_1, ..., E_n)
hiding	**hide** g_1,...,g_n **in** B

- **B, B1, B2** stand for behaviour expressions
- **g, g_i** are gate names
- **E, Ei** are value expressions
- **s, s_i** are sorts ("types of value")
- **P** is a process name

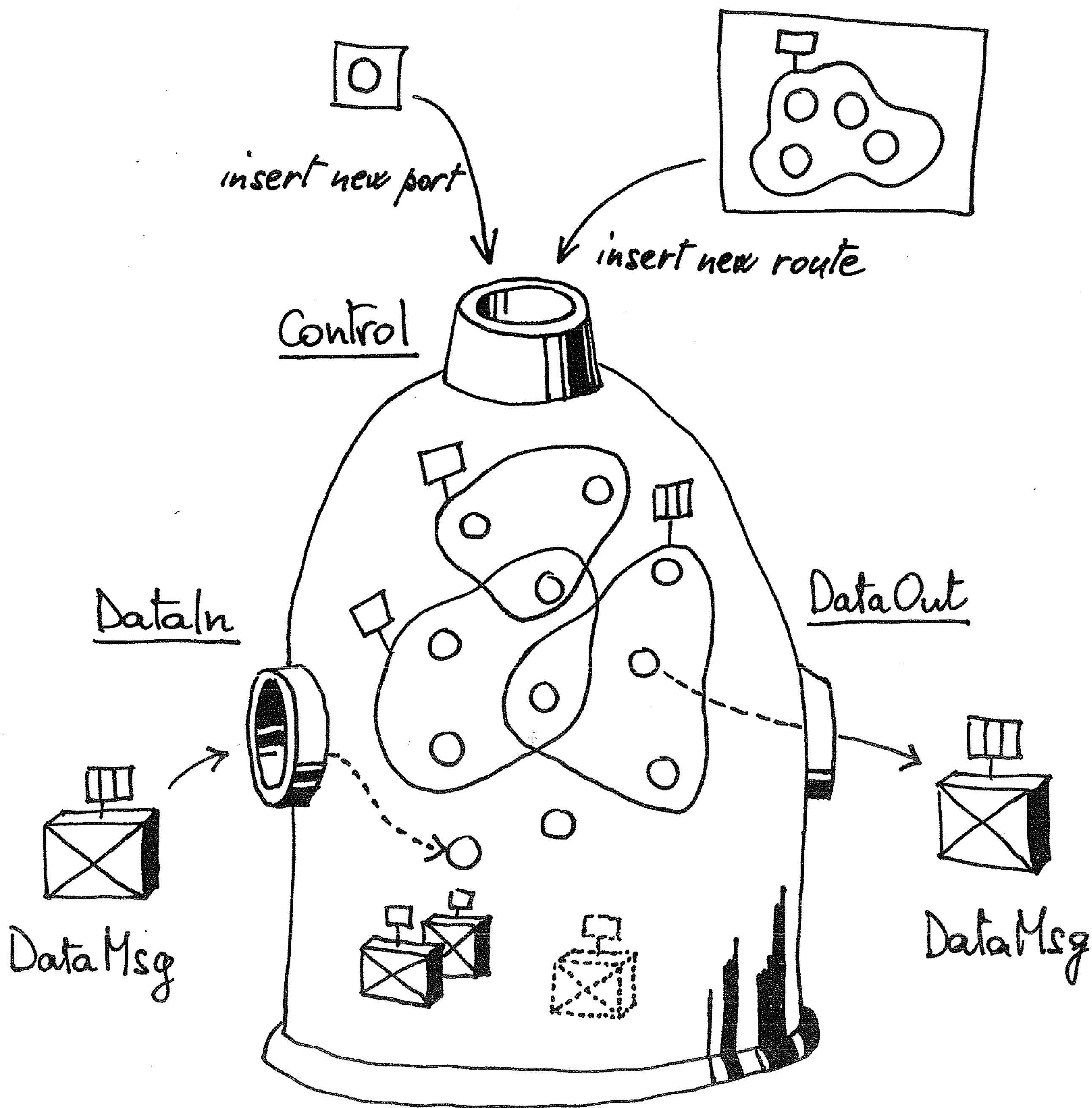
6. An example: specification of a switching node

Simplified version of
J. Quemada, A. Azcorra
"A Constraint Oriented Specification of Al's Node", in:

The Formal Description Technique LOTOS
P.H.J Van Eijk, C.A. Vissers, M. Diaz (editors)
North-Holland 1989.

6.1

Informal description



○ = port

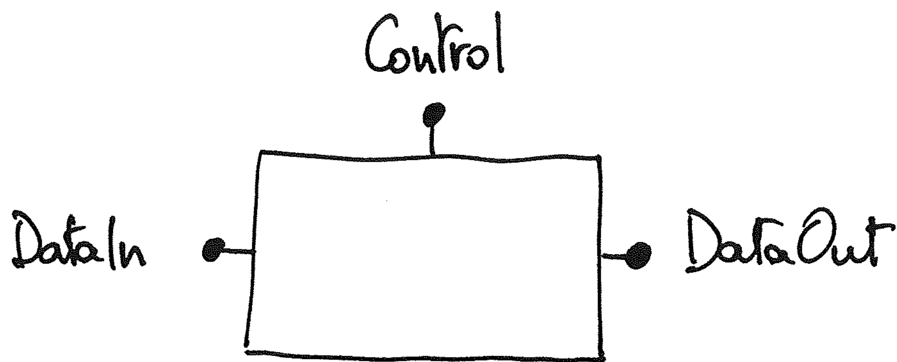
= route id.

= DataMsg

= route

= data

= vanished DataMsg



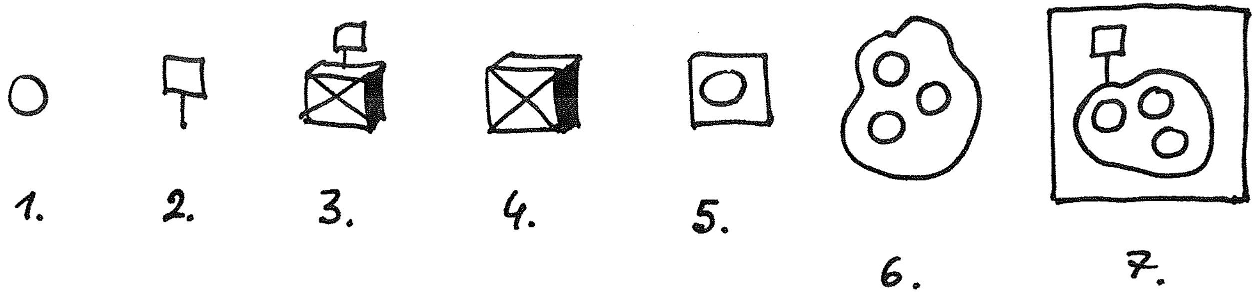
Gates and alphabet of observable actions (formal)

DataIn ?_:PortId ?_:DataMsg
(* at DataIn, *at* some PortId, some DataMsg is offered *)

DataOut ?_:PortId ?_:DataMsg
(* at DataOut, *at* some port, some data is offered *)

Control ?_:PortMsg
(* at Control, some PortMsg is offered *)

Control ?_:RouteMsg
(* at Control, some RouteMsg is offered *)



Data structures (informal)

1.	a PortId is a	Nat
2.	a RouteId is a	Nat
3.	a DataMsg is	*DataMsg*(RouteId, Data)
4.	a Data is an	OctetString
5.	a PortMsg is	*PortMsg*(PortId) (* an envelope ...*)
6.	a PortSet is a	Set of PortId's
7.	a RouteMsg is	*RouteMsg*(RouteId, PortSet)
	a RouteSet is a	Set of RouteMsg's

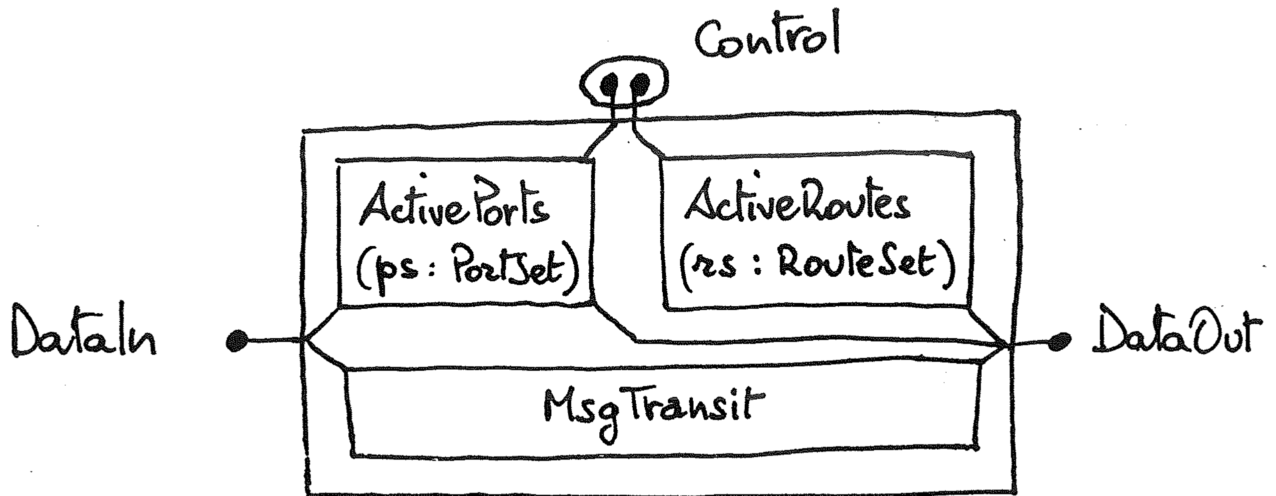
Furthermore

PortNum(PortMsg) gives the PortId of that PortMsg
(* opens the envelope *)

Consistent(PortId, DataMsg, RouteSet) is TRUE iff

if p is the PortId,
DataMsg(r-id, ...) is the DataMsg,

then some *RouteMsg*(r-id, {...,p,...})
is in the RouteSet.



Top level behaviour expression

MsgTransit [DataIn, DataOut]

|[DataIn, DataOut]|

(ActivePorts [Control, DataIn, DataOut]({} of PortSet)
|[DataOut]|
ActiveRoutes [Control, DataOut]({} of RouteSet)
)

Process MsgTransit

Repeatedly inputs at DataIn, at some PortId, some DataMsg, then outputs at DataOut, at *some* PortId, *that* DataMsg, or loses it; unlimited buffering capacity.

Process ActivePorts

Inp. at Control a PortMsg & updates set of active ports,
or inp. at DataIn, at some *active* port, some DataMsg,
or outp. at DataOut, at some *active* port, some DataMsg.

Process ActiveRoutes

Inp. at Control a RouteMsg & updates set of active routes,
or outputs at DataOut *some* DataMsg, at *some port consistent with* the route indicated in such DataMsg.

```
process MsgTransit [DataIn, DataOut] : noexit :=

   OneMsg[DataIn, DataOut]  |||  MsgTransit[DataIn, DataOut]
where
   process OneMsg[DataIn, DataOut] : noexit :=
         DataIn     ?port : PortId   ?msg : DataMsg;
              (     DataOut   ?port : PortId   !msg;      stop
              []    i;     (* message loss *);             stop
              )
   endproc
endproc
```

```
Process ActivePorts   [Control, DataIn, DataOut]
                      (ps : PortSet) : noexit       :=

    Control    ?newport : PortMsg;
         ActivePorts      [Control, DataIn, DataOut]
                          (Insert(PortNum(newport), ps))

 []  DataIn      ?port : PortId   ?msg : DataMsg [port IsIn ps]
         ActivePorts      [Control, DataIn, DataOut] (ps)

 []  DataOut    ?port : PortId    ?msg : DataMsg [port IsIn ps]
         ActivePorts      [Control, DataIn, DataOut] (ps)

endproc
```

```
Process ActiveRoutes [Control, DataOut]
                     (rs : RouteSet) : noexit    :=

    Control    ? newroute  :  RouteMsg;
         ActiveRoutes [Control, DataOut](Insert(newroute, rs))

    DataOut    ?port : PortId    ?msg : DataMsg
                                   [consistent (port, msg, rs)];
        ActiveRoutes [Control, DataOut](Insert(newroute, rs))

endproc
```

7. Existing LOTOS specifications and tools

LOTOS specifications have been produced of:

- Proway Highway interface (IEEE standard)
- IEEE LAN Service
- HDLC
- ISO connectionless internetting protocol
- ISO Network Service
- ISO Transport Protocol
- ISO Transport Service
- ISO Session Protocol
- ISO Session Service
- ISO Presentation Protocol
- ISO Transaction Processing Service
- Flow Control by Latency Protocol
- parts of ISO FTAM, MTS of X400
- Computer Integrated Manufacturing architectures/components (workstation controller)

... and several more

LOTOS TOOLS Produced by the ESPRIT/SEDOS Project

---

a) an integrated set of prototype tools, written in C, on UNIX BDS 4.2:

- editor (not structured)

- front end
  checking syntax + static semantic, handling ADT library, generating Abstract Syntax Trees (input form for other tools).

- simulator (HIPPO)
  simbolic execution of full LOTOS specifications;
  generation of term rewrite system (TRS) from ADT spec., eval. of terms;
  building/navigation of communication tree; state information display, etc.

- compiler (LIW, LOTOS implementation Workbench)
  translation from high level to machine-oriented spec.;
  compilation of ADT specs. into TRS's, for computation of normal forms;
  translation of processes into C coroutines; implementation of multiway synchronization.
  (early stage)

- pretty-printer

- cross-reference generaor

b) further prototypes written in logic/functional languages for **verification** of behavioural equivalences for basic LOTOS (*Squiggles*, *Tilt*), and of ADT properties (*Perlon*) and **testing** functions (*Cantest*).

Tools at Various Universities

- LIE (Lotos Integrated Editor) structure-editor
  - for creating / debugging standard **full LOTOS** specs.: syntax + static sem. check
  - handles ADT libraries
  - data type / process browser
  - pretty printing
  - based on Cornell Synthesizer Generator (CSG).

- structure-editor and transformation tool
  - similar to previous tool, but for **basic LOTOS**
  - includes *transformation function*, based on the laws for bisimulation congruence (IS8807, Annex B)
  - based on Cornell Synthesizer Generator (CSG).

- LOLA (LOtos LAboratory)
  for interactive / batch LOTOS to LOTOS transformations; written in Pascal

- TOY
  Compiler: from LOTOS to C coroutines.

- Interpreters / simulators
- Graphical LOTOS editors

Other experimental tools

LOTTE 4.5
a working environment for existing / original LOTOS tools. Written in *Pascal.*

SPIDER
Graphical simulator, textual/graphical LOTOS interface

LOTOS environment
Simulation / transformation for a subset of LOTOS. Written in *LISP*

SDS - Symbolic Debugging System
Testing / debugging implementations derived from LOTOS.

Graphical LOTOS editor
Based on tool generator LOGGIE.