

**TRANSPARENT LOTOS**

*Technical Report C88-40*

*September 1988*

**Tommaso Bolognesi**

# Transparent LOTOS

Tommaso Bolognesi

CNUCE - C.N.R.

Technical Report C88-40

Copyright September 1988

# Preface

This is a collection of 41 foils on the ISO specification language LOTOS [1, 2]. I have used them for an introductory seminar on the language, at the FORTE '88 international conference on (standard) Formal Description Techniques (University of Stirling, September 6th, 1988). I believe that an ideal duration for their presentation, including reasonable interaction with the audience, is two hours and a half. If you intend to use this material, please let me know of any difficulties encountered in your presentation, and feel free to suggest improvements. I wish to express my gratitude to Ed Brinksma and Jan De Meer, who have indirectly contributed to some parts of the presentation. The formal specification of the Daemon Game example is completely due to Ken Turner.

Pisa, Sept. 19, 1988

Tommaso Bolognesi

- 
- [1] ISO - Information Processing Systems - Open Systems Interconnection - "LOTOS- A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", IS 8807, 1987.
  - [2] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems, Vol. 14, No 1, 1987 .

# Transparent LOTOS

Tommaso Bolognesi  
CNUCE / C.N.R. - Pisa

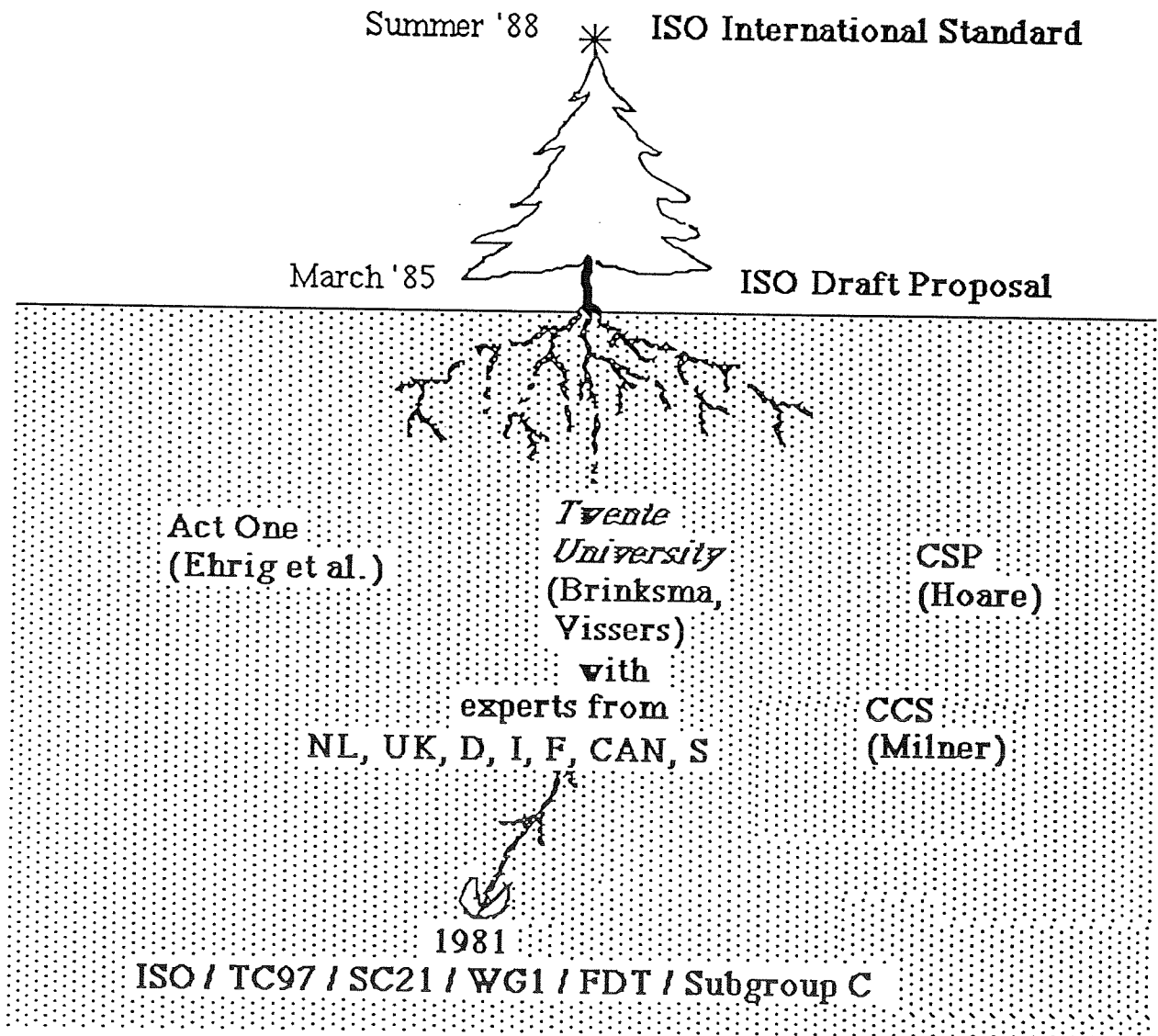
## Contents

1. General features of the language
  2. Specification of abstract data types
  3. Specification of process behaviours - I
- \*\*\*
4. Specification of process behaviours - II
  5. Daemon Game example

# 1. General features of the language

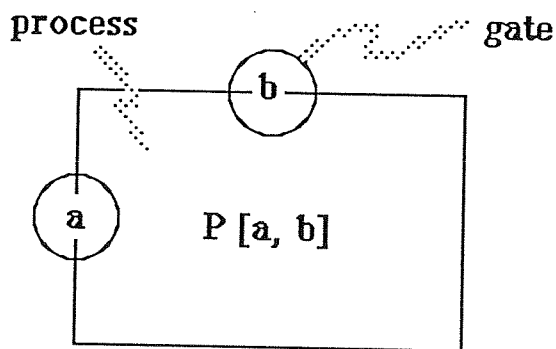
# The LOTOS tree

Language  
Of  
Temporal  
Ordering  
Specification (of communicating systems)

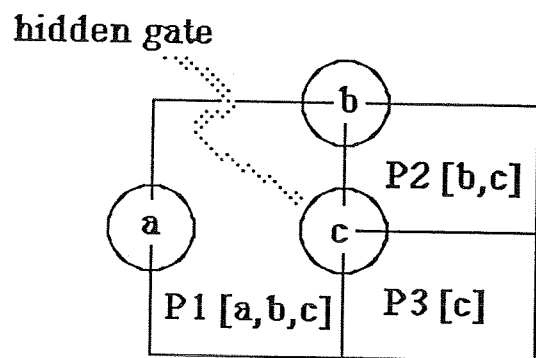


# LOTOS describes a system as a *process* which may:

- *interact* with its environment via interaction points called *gates*, that is, perform *observable actions* at the gates (observation = interaction);
- perform *unobservable (internal, hidden) actions*.



Outer view of process P



Internal structure of P

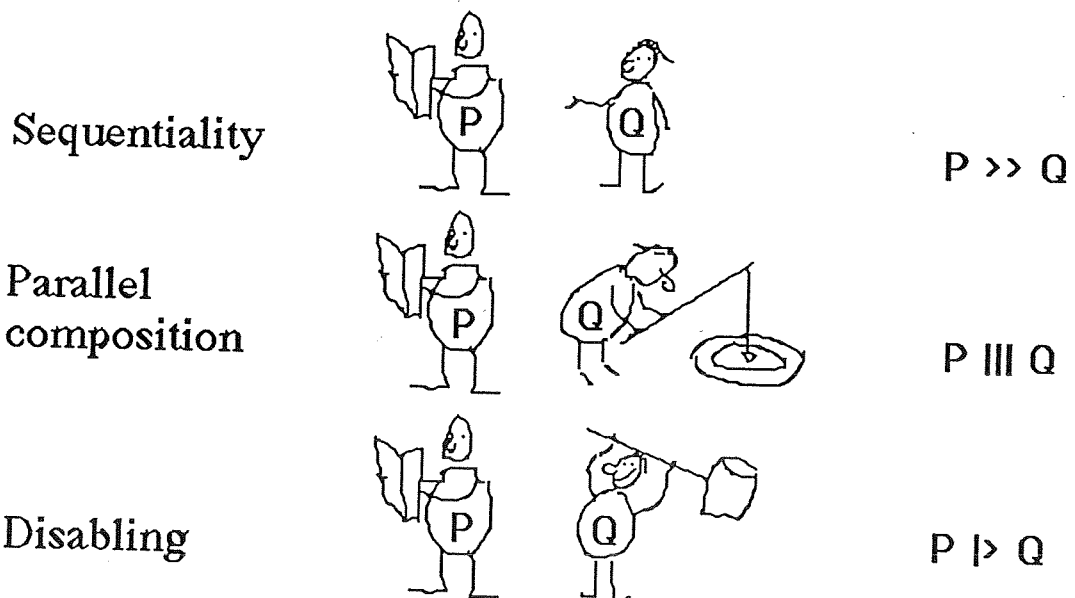
- An *observable action* consists of offering / accepting ("establishing") zero or more *values* at a *gate*.
- An *interaction* may occur when two or more processes are ready to perform the same *observable action*.
- An interaction may involve *data exchange*, and is an *instantaneous event* (synchronous communication).

## Algebraic nature

- Process behaviours are described by *algebraic expressions*, called

*behaviour expressions*.

Complex behaviours (processes) are expressed by composing more simple behaviour expressions (subprocesses) via the LOTOS *operators*. Examples:



- Behaviour expressions satisfy *algebraic laws*. Examples:

$$(P \triangleright Q) \parallel Q = P \triangleright Q$$

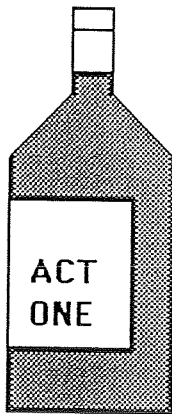
$$a; (P \parallel Q) \neq a; P \parallel a; Q$$

- LOTOS abstract data types  $\rightarrow$  multi-sorted *algebras*

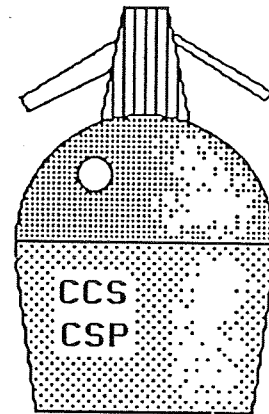
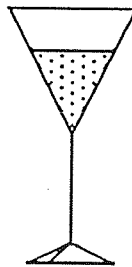


## Two components

- Ultimately, the behaviour of a process is expressed in terms of *which interactions* (observations) are possible with it, that is:
  - WHICH values are offered / accepted ...
  - WHERE / WHEN (in which *temporal order*) ?



Abstract data types(ADT)



Processes

*value expression:*  
WHAT is offered  
in the interaction ?

*behaviour expression:*  
WHEN/WHERE does  
the interaction occur ?

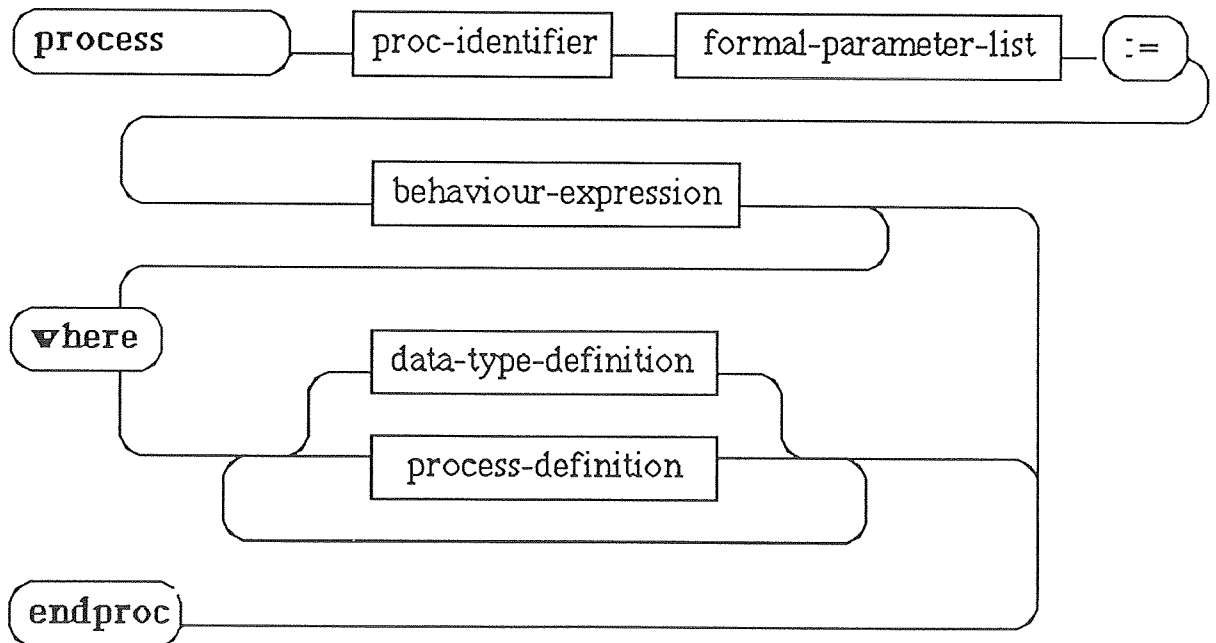
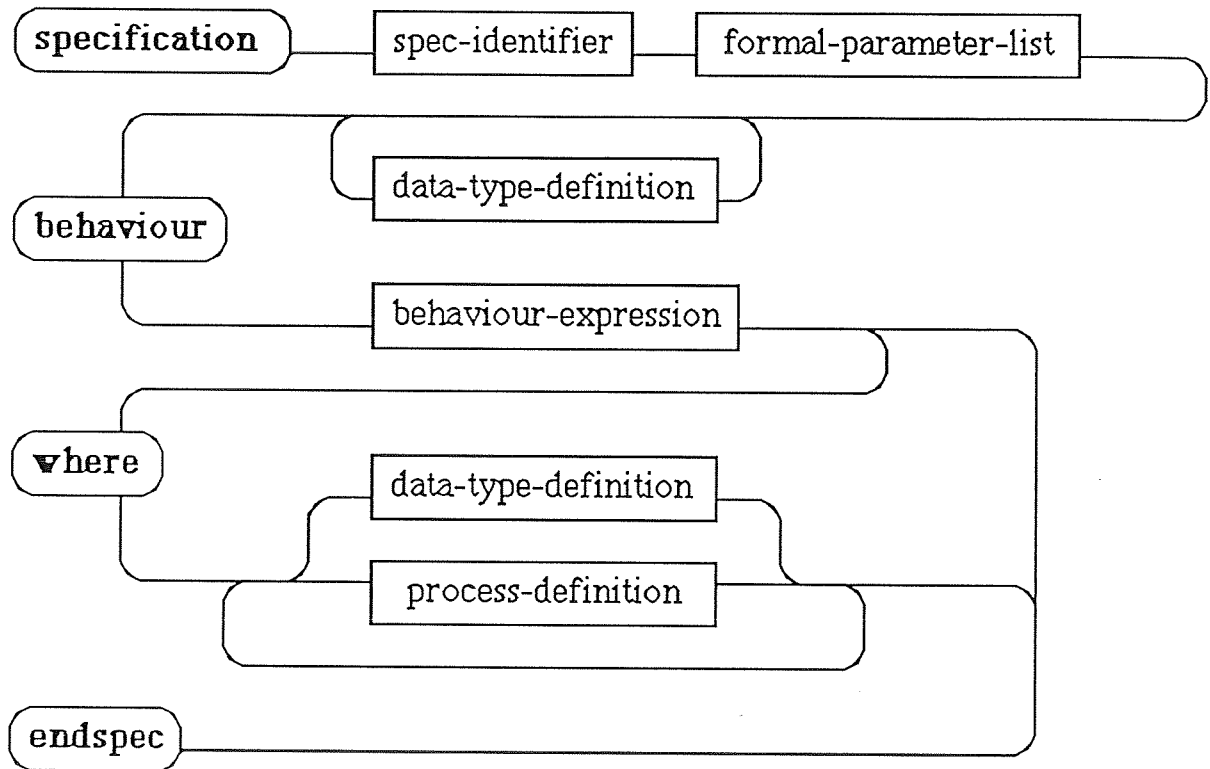
---

push((x + 1), stack)

**first** interact at gate 'a',  
**then** interact at gate 'b'

- LOTOS specification = ADT defs. + process defs.


# 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>*



```
g ! <value expr. 1>; exit (<value expr. 2>)  
[]  
[<bool. value expr. 3>] --> P[g] (<value expr. 4>)
```

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

## 2. Specification of abstract data types

- *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

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

**eqns forall** m, n : Nat

**ofsort** Nat

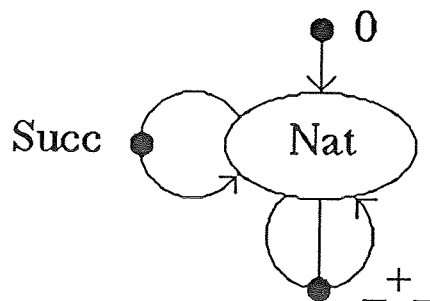
$m + 0 = m$

$m + \text{Succ}(n) = \text{Succ}(m) + n$

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

**endtype**

- 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)

$$0 + \text{Succ}(0) = \text{Succ}(0)$$

# Extensions of type definitions

**type** BasicNaturalNumber (\* Standard Library \*) **is**

VeryBasicNaturalNumber

**opns**    \_ \* \_, \_ \*\* \_    : Nat, Nat --> Nat

**eqns**    **forall** m, n : Nat

**ofsort** Nat

          m \* 0            =    0;

          m \* Succ(n)    =    m + (m \* n);

          m \*\* 0           =    Succ(0);

          m \*\* Succ(n)   =    m \* (m \*\* n);

**endtype**

- Extensions may also introduce new sorts.

Example (from "Daemon Game"):

**type** IdentifierType **is**

Boolean

**sorts** IdSort

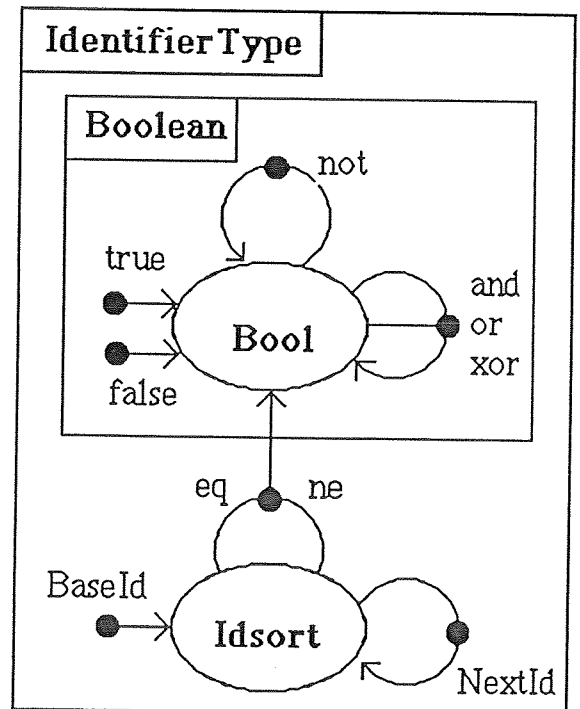
**opns**    BaseId        : ...

          NextId       : ...

          \_eq\_, \_ne\_   : ...

**eqns**    ...

**endtype**



(equations not shown)

# Parameterized types

```

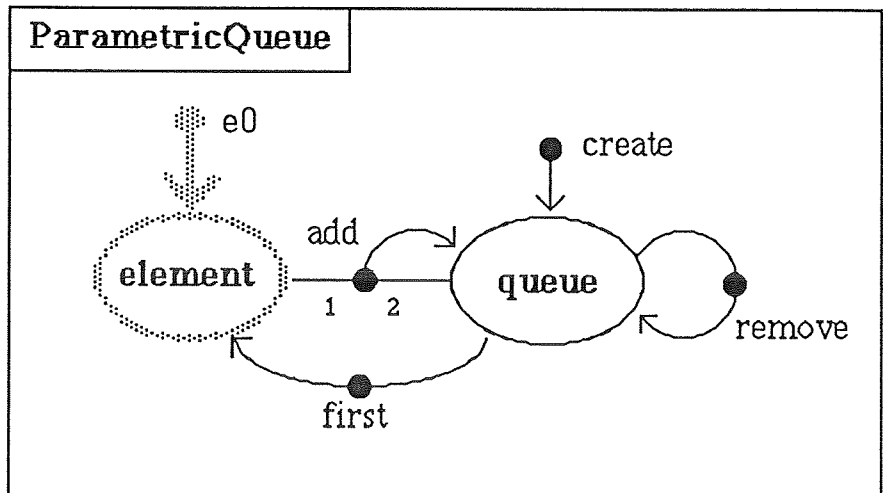
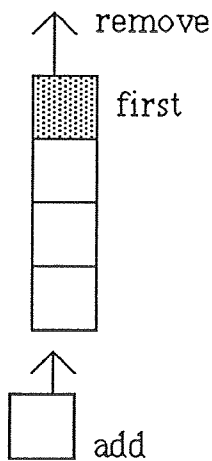
type    ParametricQueue  is

  formalsorts  element
  formalopns   e0        : --> element

  sorts        queue
  opns         create    : --> queue
               add       : element, queue --> queue
               first     : queue --> element
               remove    : queue --> queue

  eqns        ...

endtype
  
```



This definition characterizes only those properties of the objects of sort 'queue' which *do not depend on the nature of the elements* in the queue.

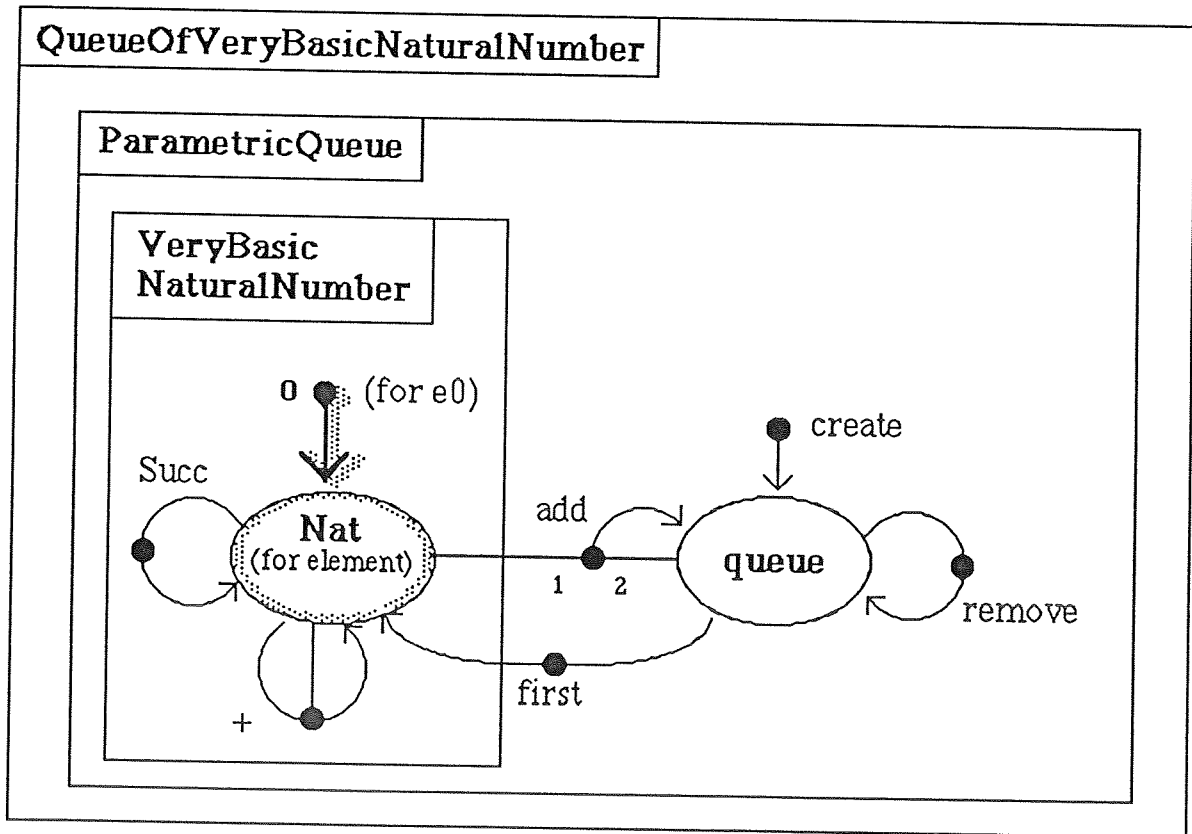


# Actualizing a parametric type

type QueueOfVeryBasicNaturalNumber is actualized by  
 ParametricQueue using  
 VeryBasicNaturalNumber

sortnames nat for element  
 opnames 0 for e0

endtype



All the properties of the actualizing type (its equations) are imported into the actualized type

- **Formal equations** can be used within a parametric type definition for imposing **semantic requirements** on candidate actualizing types.

### *Example*

A new formal operation is added to type ParametricQueue, and is required to be commutative:

```

type RefinedParametricQueue is
  ...
  formalopns   #
  formaleqns  forall x, y : element
                ofsort element
                x # y = y # x
  ...
endtype

```

The actualization below is correct because '+' is commutative (as implied by the equations of VeryBasicNaturalNumber):

```

type   QueueOfVeryBasicNaturalNumber   is
        RefinedParametricQueue         actualizedby
        VeryBasicNaturalNumber         using

        sortnames      nat for element
        opnnames      0 for e0
                       + for #
endtype

```

# Type renaming

- Creates a new type, *isomorphic* to an already defined one.
- Allows one to assign *mnemonic names* to sorts and operations which fit better into the desired application.

## Example

```
type    ParametricConnection  is
        ParametricQueue      renamedby

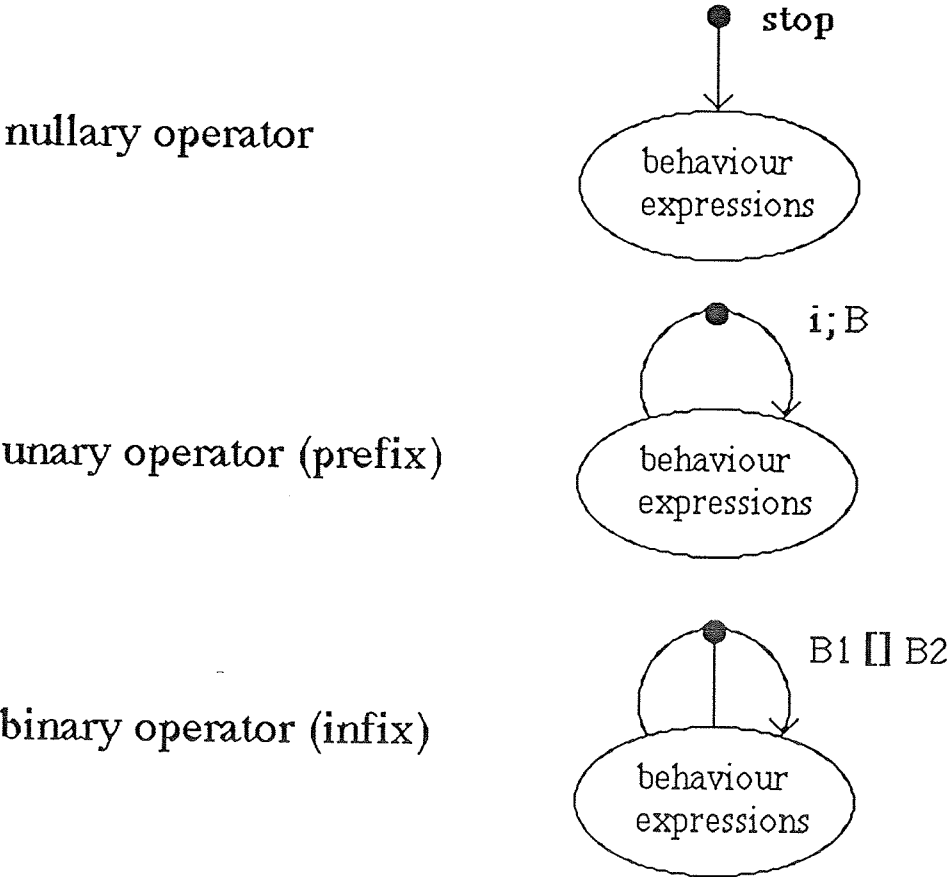
        sortnames  channel    for queue
                   message    for element (* formal *)
        opnnames   nomessage  for e0
                   send       for add
                   receive    for first
endtype
```

(the operations **create** and **remove** of type ParametricQueue are not renamed, but they are still available.)

### 3. Specification of process behaviours - I

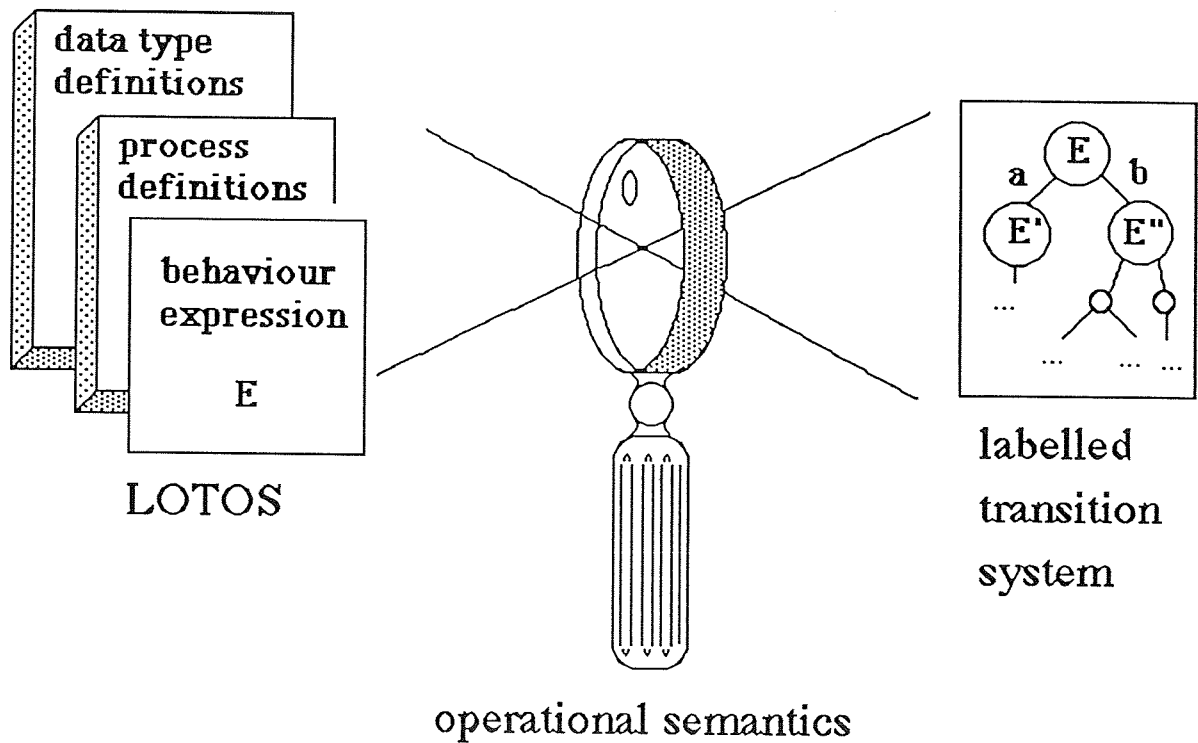
A *behaviour expression* is built by applying a (behavioural) operator to 0, 1 or 2 *behaviour expressions*.

*Examples:*



B, B1 and B2 are *behaviour expressions*

# Formal semantics of behaviour expressions



The *axioms* and *inference rules* of the operational semantics allow one to derive:

- the initial, alternative *actions* of behaviour E (e.g.: a, b)
- the *behaviour expressions* into which E is transformed after the occurrence of its initial actions (e.g.: E', E'')

- The inference rules identify the initial actions of E based on its *syntactic structure*.

*Example*

$$(E_1 \parallel E_2) \xrightarrow{???} ???$$

is defined in terms of

$$E_1 \xrightarrow{x} E_1' \quad \text{and} \quad E_2 \xrightarrow{y} E_2'.$$

- There are two types of action:
  - $i$   
the internal (unobservable, hidden) action
  - $g \langle v_1, \dots, v_n \rangle$   
the observable action of offering / accepting a tuple  $\langle v_1, \dots, v_n \rangle$  of 0 or more values of some sort at gate  $g$

inaction:

**stop**

No axiom is associated to the behaviour expression **stop**.

Hence this behaviour does not denote any action.

*Example:*

---

```
process broken_vending_machine : noexit :=  
    stop  
endproc
```

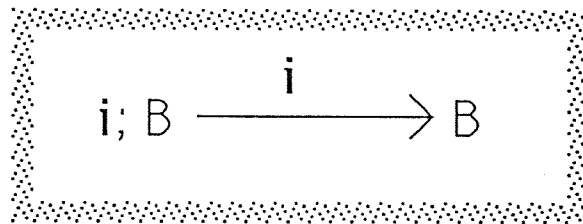
---



action prefix (unobserv.):

**i; B**

**Axiom:**



*Example:*

---

```
process broken_vending_machine : noexit :=  
    i; stop  
endproc
```

---

The only possible transition is

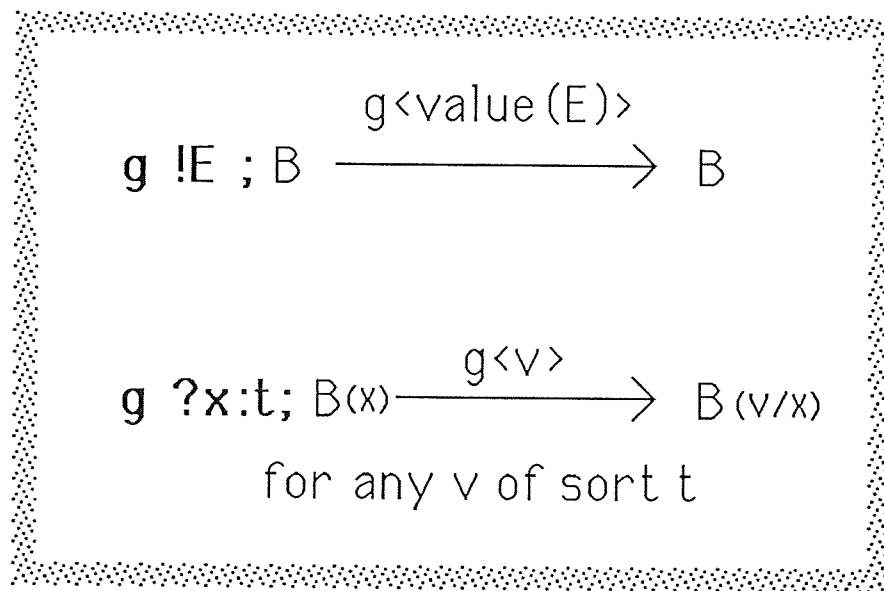
$i; stop \text{ ---}i\text{---} \rightarrow stop$

but the user (observer) cannot distinguish this machine from the previous one.

action prefix (observ.):

$g \dots; B$

**Axioms:**



(the obvious generalization to the case ' $g ?\dots!\dots?\dots; B$ ' applies)

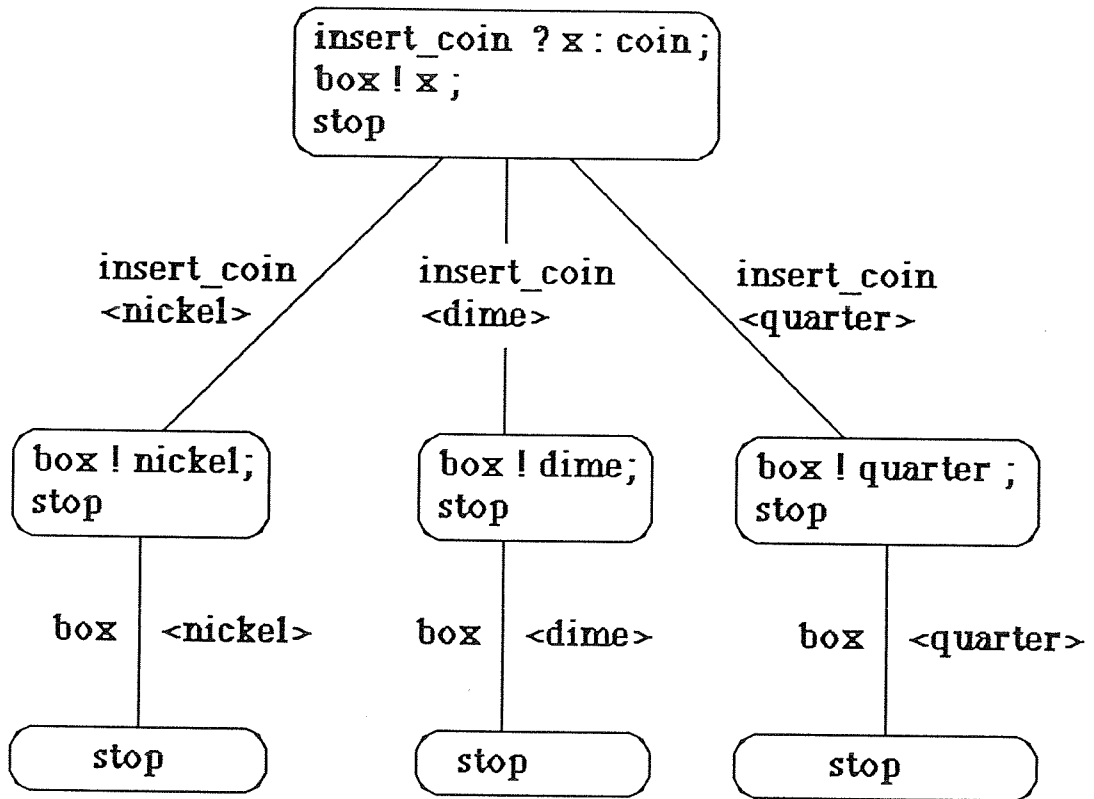
*Example:*

---

```
process useless_machine [insert_coin, box]: noexit :=  
    insert_coin ? x:coin;  
    box ! x;  
    stop  
endproc
```

---

Given a type with constants 'nickel', 'dime', 'quarter', of sort 'coin', the following transition tree can be derived:



The coin inserted at gate *insert\_coin* is returned at gate *box*.

Note the substitution of nickel/dime/quarter for *x* in expression "box ! x; stop".

choice:

$B1 \ [] \ B2$

**Inference rules:**

$$\frac{B1 \xrightarrow{a} B1'}{B1 \ [] \ B2 \xrightarrow{a} B1'}$$
$$\frac{B2 \xrightarrow{a} B2'}{B1 \ [] \ B2 \xrightarrow{a} B2'}$$

*Example:*

---

```
process trap_machine [insert_coin, box]: noexit :=  
  insert_coin ?x:coin;  
  ( box ! x; stop  
  [] i; stop  
  )  
endproc
```

---

It may happen that the inserted coin is never returned.

guarding:

$\langle \text{guard} \rangle \dashrightarrow B$

$\langle \text{guard} \rangle$  is

- i)  $\langle \text{value expression 1} \rangle = \langle \text{value expression 2} \rangle$
- ii)  $\langle \text{boolean expression} \rangle$

**Inference rule:**

$$\frac{B \xrightarrow{a} B' \text{ and the guard is True}}{[\langle \text{guard} \rangle] \dashrightarrow B \xrightarrow{a} B'}$$

*Example of guarding and process instantiation (recursion)*

---

```
process fair_machine [insert_coin, box] : noexit :=  
  insert_coin ?x:coin;  
  (  
    [eq(x, quarter)] -> box !gianduiotto;  
                                fair_machine [insert_coin, box]  
  [] [ne(x, quarter)] -> box ! x;  
                                fair_machine [insert_coin, box]  
  )  
endproc
```

---

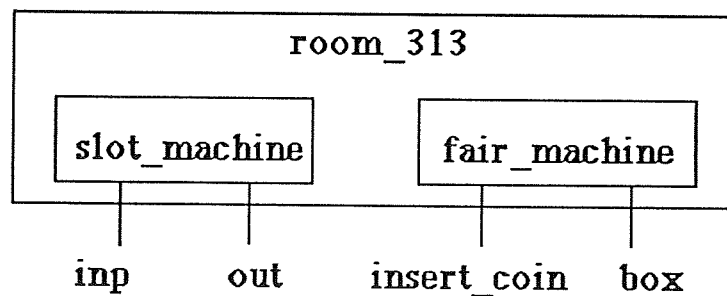
LOTOS gates are not typed

## 4. Specification of process behaviours - II

parallel comp. - interleaving: B1 ||| B2

Behaviours B1 and B2 are independent of each other, and any interleaving of their actions is possible.

### Example



*process* room\_313 [inp, out, insert\_coin, box] : noexit :=

*slot\_machine* [inp, out]  
||| *fair\_machine* [insert\_coin, box]

where

*process* slot\_machine [inp, out] : noexit :=

    inp ! dime;

    ( i; out ! nickel; slot\_machine [inp, out]

    [] i; out ! dime; slot\_machine [inp, out]

    [] i; out ! quarter; slot\_machine [inp, out]

    )

*endproc*

*process* fair\_machine [insert\_coin, box] : noexit :=

    ...

*endproc*

*endproc* (\* room\_313 \*)

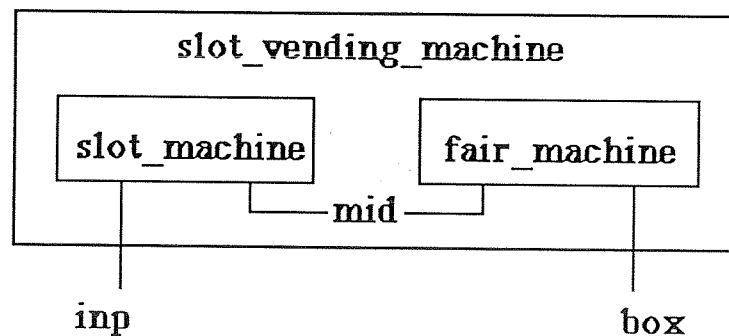
parallel comp.-  
synchronization:  $B1 \parallel [g1, \dots, gn] \parallel B2$

Behaviours B1 and B2 are independent of each other, but must interact at the *synchronization gates*  $g1, \dots, gn$ .

hiding:  $\text{hide } g1, \dots, gn \text{ in } B$

The actions of B occurring at the *hidden gates*  $g1, \dots, gn$  are transformed into the unobservable action  $\tau$ .

### *Example*



```
process slot_vending_machine [inp, box] : noexit :=
```

```
  hide mid in
```

```
    (slot_machine [inp, mid] \[mid]\ fair_machine[mid, box])
```

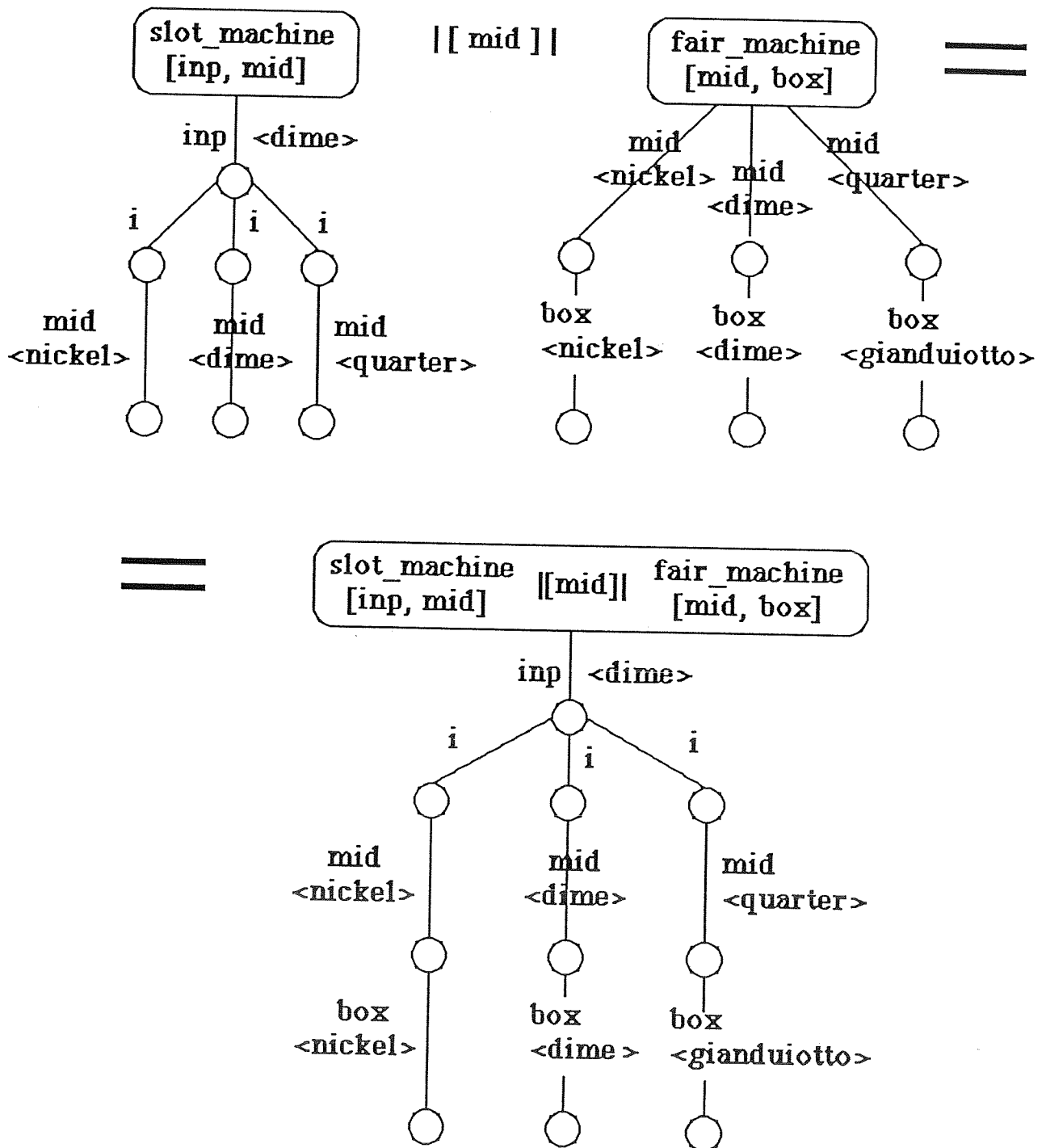
```
  where ...
```

```
endproc (* room_313 *)
```

Synchronization on all gates is expressed by  $B1 \parallel B2$ :  
the two composed processes are forced to proceed together.

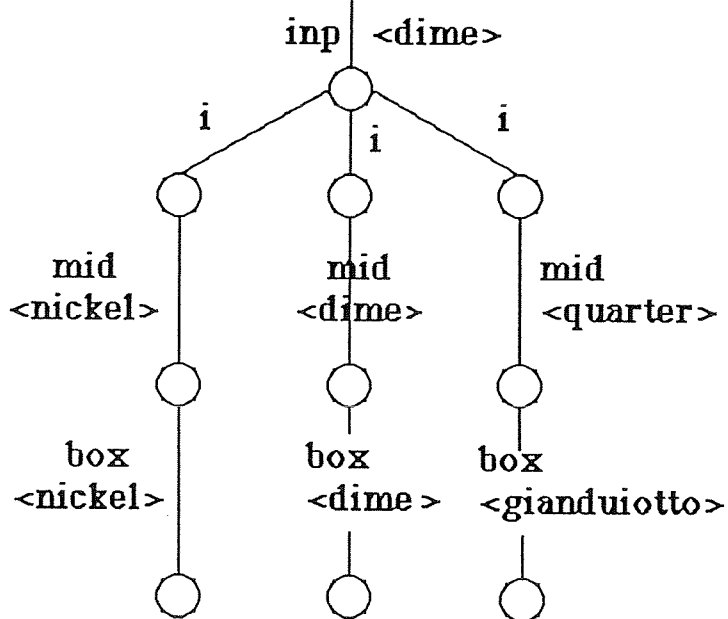
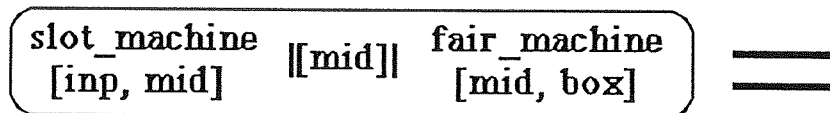


# Synchronization with transition trees



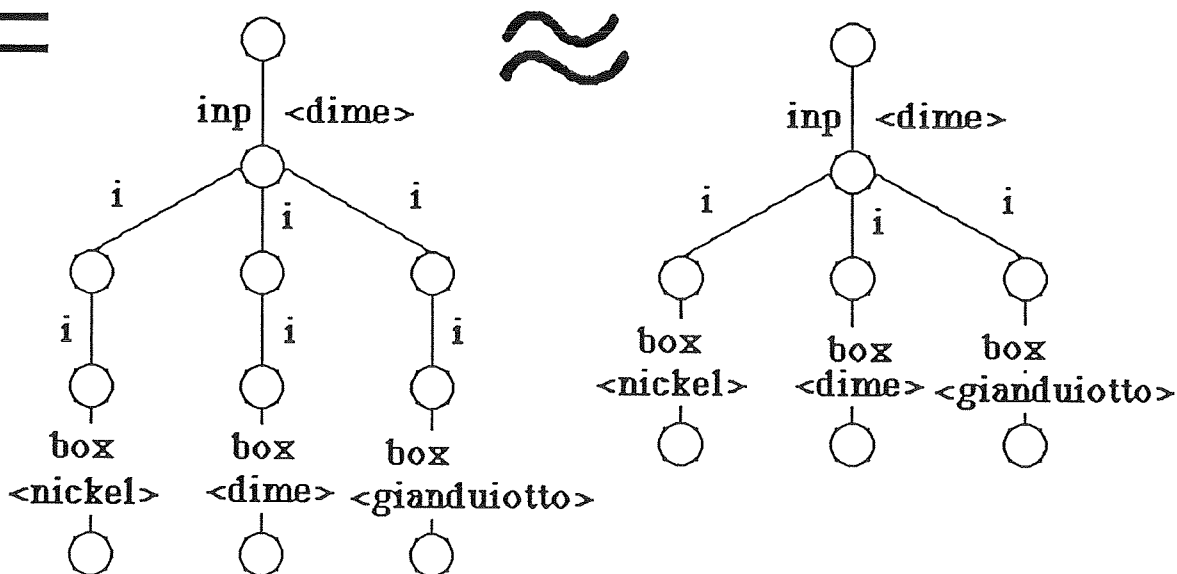
# Hiding on action trees

hide mid in



==

≈



## Sequential composition (enabling):

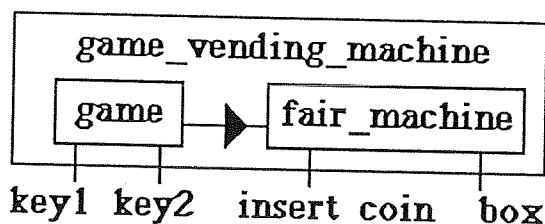
$B1 \gg [accept\ x1:t1, \dots, xn:tn\ in]\ B2$

The behaviour B2 is enabled if and when the behaviour B1 reaches a *successful termination*.

Successful termination:  $exit[(E1, \dots, En)]$

a self-explanatory, altruistic alternative to the **stop** process.  
Value expressions E1, ..., En match variables x1:t1, ..., xn:tn

### *Example*



```
process game_vending_machine
  [key1, key2, insert_coin, box] : noexit :=
    game[key1, key2] >> fair_machine[insert_coin, box]
where
  process game[key1, key2] : exit :=
    key1; game[key2, key1]
  [] key2; exit
  endproc

  process fair_machine[insert_coin, box] : noexit :=
    ...
  endproc
endproc
```

## Disabling:

B1 [ $\triangleright$ ] B2

Behaviour B1 may be disabled at any time (except after successful termination) by behaviour B2.

## *Example*

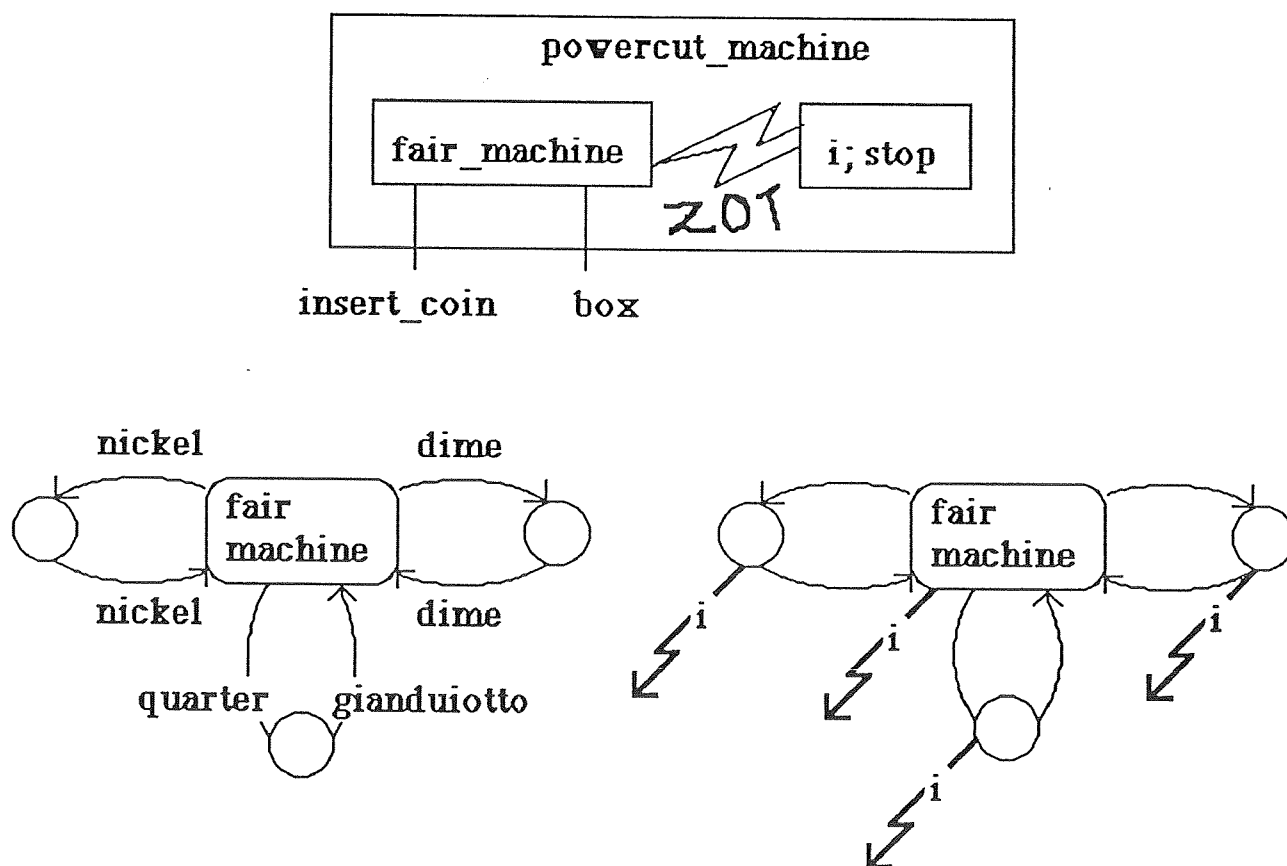
---

```
process powercut_machine [insert_coin, box] : noexit :=
```

```
    fair_machine[insert_coin, box]
  [ $\triangleright$ ] i; stop
```

```
where ...
endproc
```

---



Generalized choice:      **choice**  $x:t [] B(x)$

The choice is offered among all behaviours  $B(v/x)$ , where  $v$  is any value of sort  $t$ .

### *Example*

The process definition:

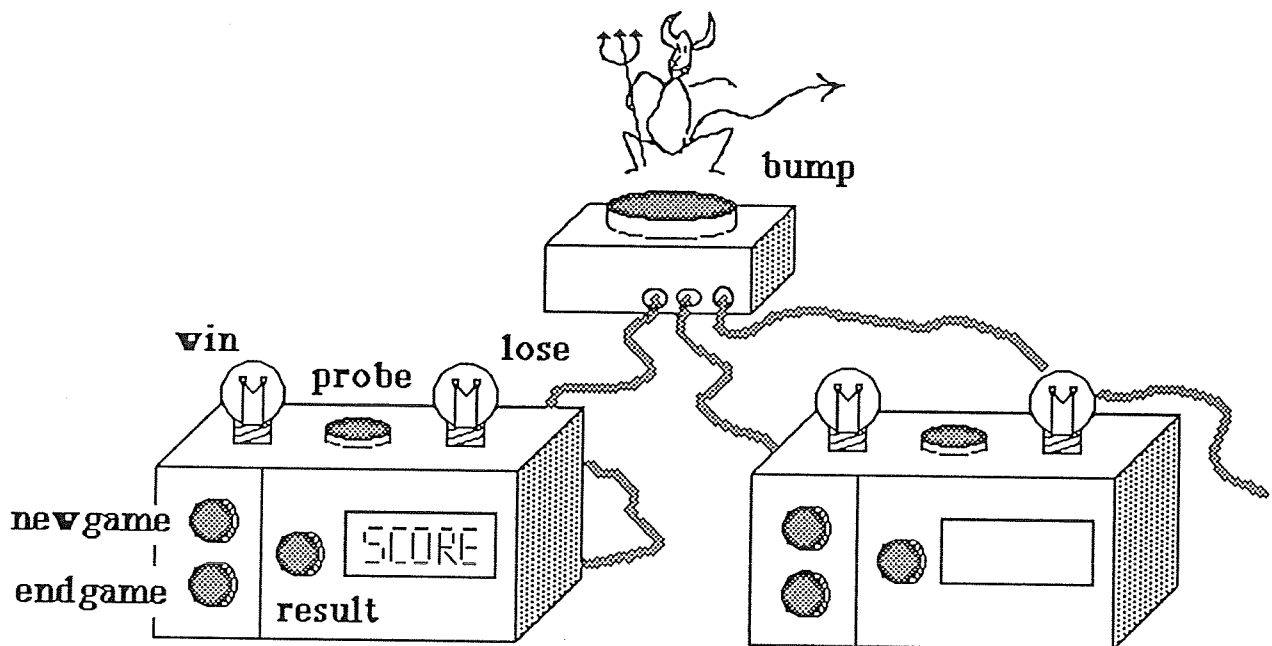
```
process slot_machine [inp, out] : noexit :=  
  inp ! dime;  
  choice x : coin []  
    i; out ! x; slot_machine [inp, out]  
endproc
```

is equivalent to the previous process definition:

```
process slot_machine [inp, out] : noexit :=  
  inp ! dime;  
  (  
    i; out ! nickel; slot_machine [inp, out]  
    [] i; out ! dime; slot_machine [inp, out]  
    [] i; out ! quarter; slot_machine [inp, out]  
  )  
endproc
```

## 5. Daemon Game example

## Daemon game - informal specification



- A daemon generates **bump** signals randomly.
- A player guesses, by the command **probe**, whether the number of bumps is
  - odd: the system signals **win**, and increases the player's score by one, or
  - even: the system signals **lose**, and decreases the player's score by one.
- A player may issue the **result** command, to see his **score**.
- At *login* (command **newgame**) the system allocates to the player a unique identifier, and  $\text{score} = 0$ ; at *logout* (command **endgame**) the identifier is de-allocated.

## Daemon game in LOTOS

### Fundamental design choices:

- *No central daemon.*

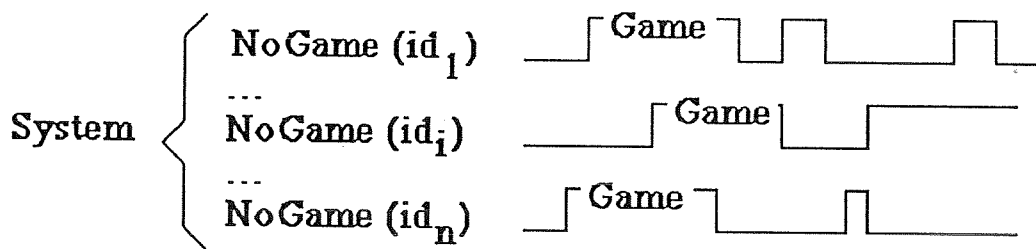
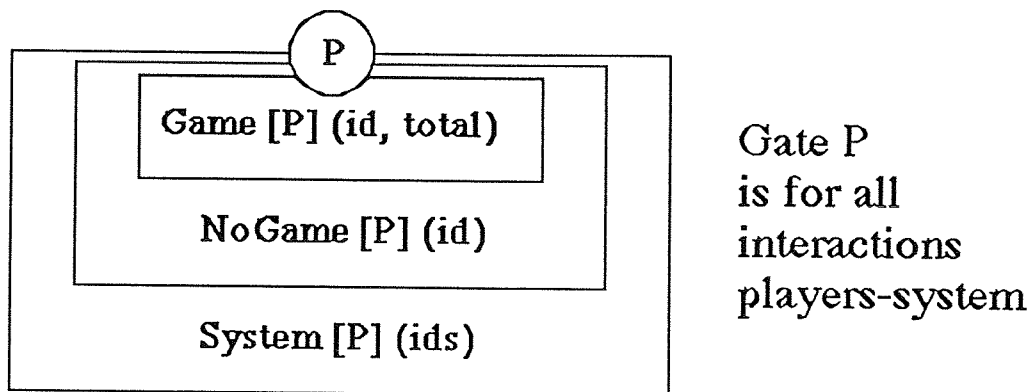
The spec. must describe the system behaviour *as experimented by the players*. Since experiments (e.g. **probe**) cannot be simultaneous, players cannot distinguish an architecture with a centralized daemon from one with many local daemons.

- *No explicit spec. of the local daemons*

The local daemon is experienced by the player as a random occurrence of signal **win** or **lose** from the system (==> no **bump** signals, no **bump** counter).



## Top level structure of the spec. - Processes



**System** describes the overall system as the independent composition of all permitted games.

**NoGame** describes the infinite sequence of games played under the same game-id. (login-id.). Unexpected commands issued between two games are also handled.

**Game** describes, for a single game-id., all legal interactions between system and player that may occur between a **newgame** command and the next **endgame** command.

## Top level structure of the spec. - Data types

<b>Identifier</b>	for distinguishing games by login ids.
<b>IdentifierSet</b>	for indicating the set of login-ids supported by the system.
<b>Integer</b>	for scoring.
<b>Signal</b>	for interactions players - system.

All observable actions at gate P are of type:

$$P \langle \text{id}, \text{sig} \rangle$$

where

id is the player involved;

sig is the command issued by the player, or the signal generated by the system.

## Daemon game - Data types

(\* The following type defines game-identifiers. An infinite set of pairwise distinct objects is defined. \*)

```
type      IdentifierType is Boolean

sorts     IdSort

opns      BaseId          : --> IdSort
           NextId          : IdSort --> IdSort
           _eq_, _ne_      : IdSort, IdSort --> Bool

eqns      forall Id, Id1, Id2 : IdSort
           ofsort Bool
           BaseId          eq BaseId          = True;
           BaseId          eq NextId(Id)      = False;
           NextId(Id)      eq BaseId          = False;
           NextId(Id1)     eq NextId(Id2)     = Id1 eq Id2;

           Id1 ne Id2      = not(Id1 eq Id2);

endtype
```

(\* The following type renames the library type Set, without affecting its formal components. \*)

```
type      IdentifierSetFormalType  is  
           Set                      renamedby
```

```
           sortnames  IdSetSort    for Set
```

```
endtype
```

(\* The following type defines sets of game-identifiers. It is an actualization of the parametric type IdentifierSetFormalType \*)

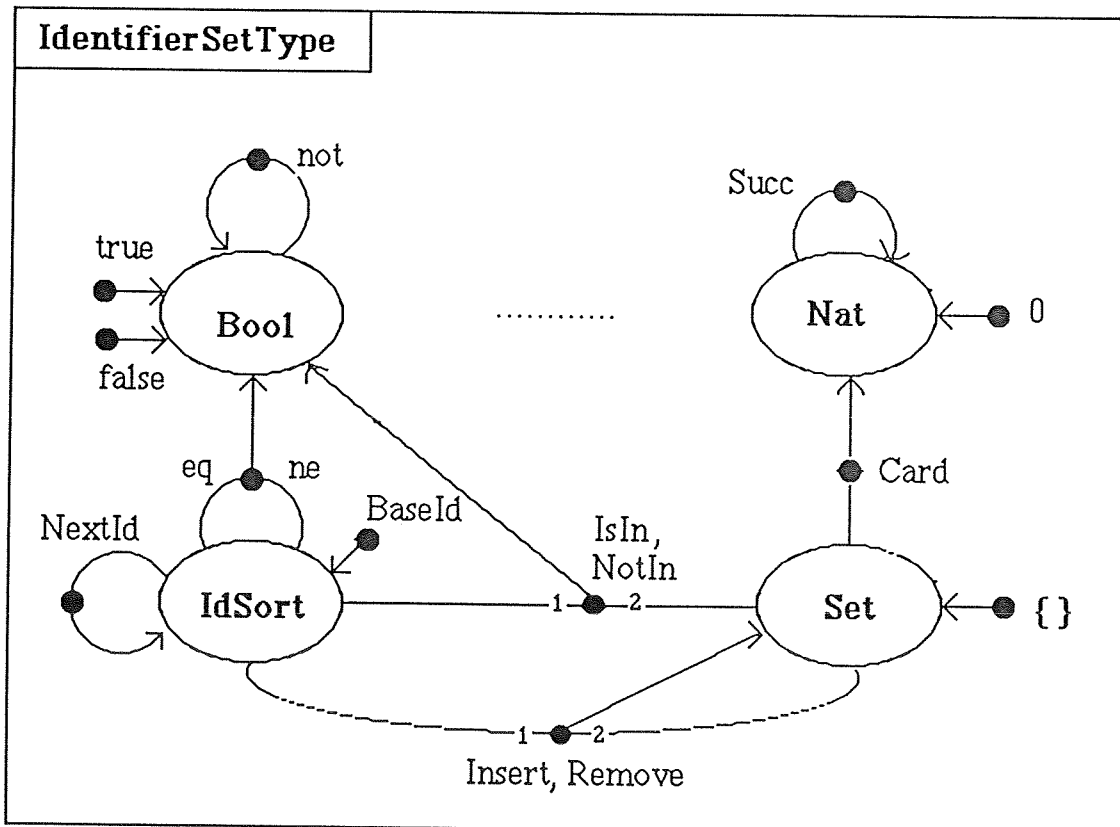
```
type      IdentifierSetType        is  
           IdentifierSetFormalType  actualizedby  
           IdentifierType, Boolean  using
```

```
           sortnames  IdSort    for  element  
                   Bool      for Fbool
```

```
endtype
```

(\* Signature of IdentifierSetType

(some of the operations not used in the spec. are not shown)



\*)

(\* "... you mean this one is not in the library ?... \*)

```
type      IntegerType is

  sorts    IntSort

  opns     0           : --> IntSort
           inc, dec    : IntSort --> IntSort

  eqns     forall      n : IntSort
           ofsort      IntSort
           inc(dec(n))   = n;
           dec(inc(n))   = n;

endtype
```

(\* The following type defines the signals between the players and the system \*)

```
type      SignalType  is IntegerType

  sorts    SigSort

  opns     Newgame, Endgame,
           Probe, Win, Lose,
           Result           : --> SigSort
           Score           : IntSort --> SigSort

endtype
```

## Daemon game - Processes

(\* The following process specifies the overall behaviour of the system as the independent composition of all permitted games \*)

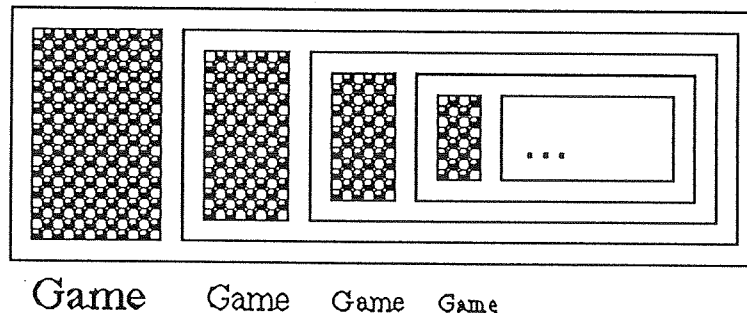
*process* System [P] (ids : IdSetSort) : *noexit* :=

*choice* id : IdSort []

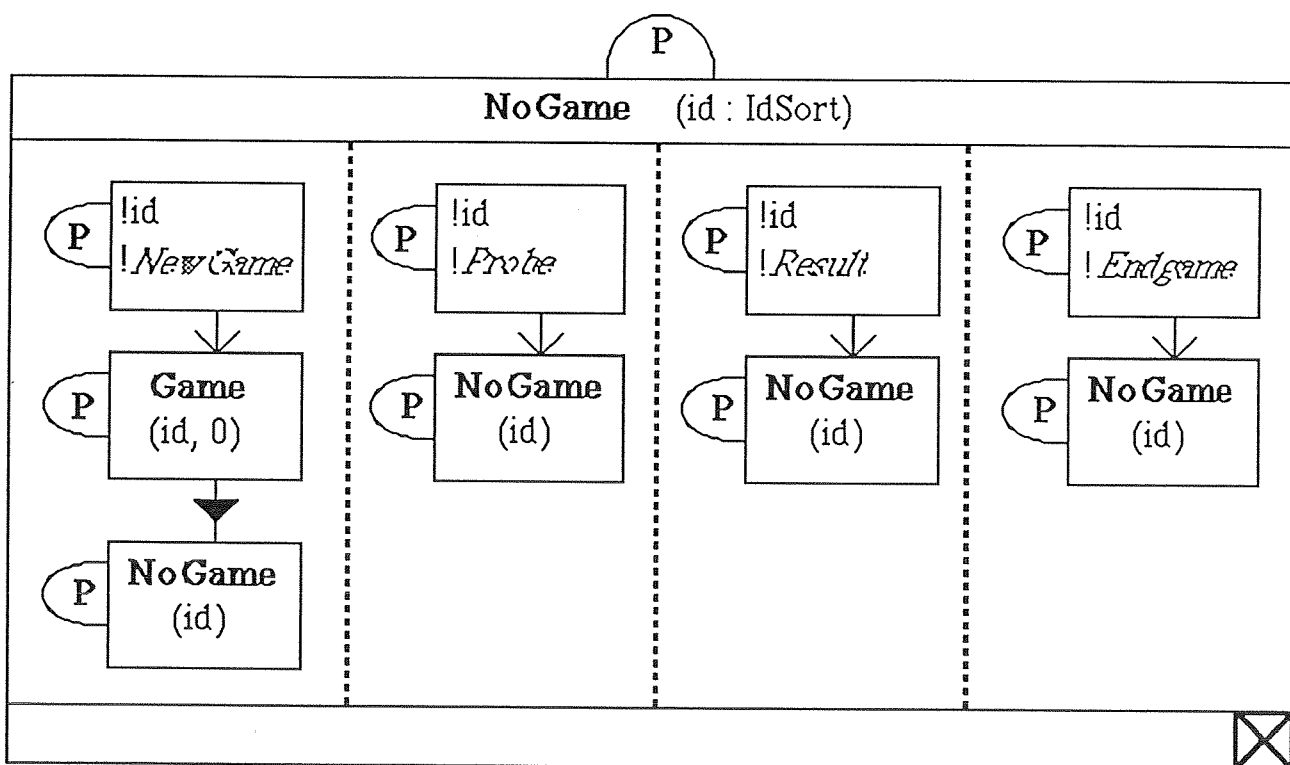
[id IsIn ids] -->

(NoGame[P] (id) ||| System[P] (Remove (id, ids)))

*endproc*



(\* The following process describes the infinite sequence of games played under the same game-id. One of the graphical syntaxes currently considered for LOTOS is adopted here; can you figure out the meaning of this definition ? (Hint: the dotted lines identify alternatives of the *choice* operator.) \*)





(\* the following process describes an individual game \*)

```
process Game [P] (id : IdSort, total : IntSort) : exit :=  
  
    P ! id ! Newgame;  
        Game [P] (id, total)  
[] P ! id ! Probe;  
    (    i; P ! Id ! Win; Game [P] (id, inc(total))  
      [] i; P ! Id ! Lose; Game [P] (id, dec(total))  
    )  
[] P ! id ! Result;  
    P ! id ! Score(total); Game [P] (id, total)  
[] P ! id ! Endgame;  
    exit  
  
endproc
```

## Complete Daemon game spec. in LOTOS

*specification Daemongame [P] (ids : IdSetSort) : noexit*

*library Boolean, Set endlib*

*type IdentifierType ... endtype*

*type IdentifierSetFormalType ... endtype*

*type IdentifierSetType ... endtype*

*type IntegerType ... endtype*

*type SignalType ... endtype*

*behaviour System [P] (ids)*

*where*

*process System [P] (ids : IdSetSort) : noexit :=  
... NoGame ...*

*where*

*process NoGame ... :=  
... Game ...*

*where*

*process Game ... :=*

*...*

*endproc (\* Game \*)*

*endproc (\* NoGame \*)*

*endproc (\* System \*)*

*endspec (\* Daemongame \*)*