ENABLING WITH VALUE TRANSMISSION

(See in B&B tutorial discussion re: exit + functionality).

Upon termination, a process can "exit" a set of values.
These can be passed on to an enabled process.

e.g.

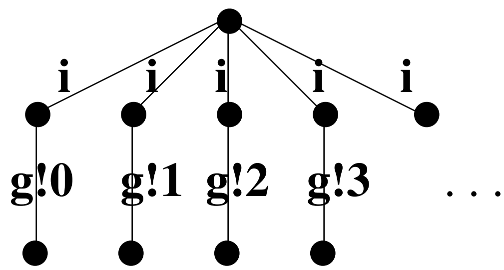
```
process P[a] : exit(nat, bool):=
    a ? x:nat ? y:nat;
        ( i; exit(x, true) [] i; exit(y, false) )
endproc

...

process Q[ . . . ]  ( a: nat,  b: bool ) :=

...

endproc

...

P[ . . . ]  >>  accept z: nat, w: bool in Q(z, w)
```

GENERALIZED CHOICE

EXAMPLE:

choice x: int [] **i** ; g!x

EQUIVALENT TO: **i**;g!0 [] **i**; g!1 [] **i**; g!2 . . .



WILL OFFER
A NONDETERMINISTICALLY
CHOSEN INTEGER
(COMMITMENT!)

choice x: int [] g!x

EQUIVALENT TO: g?x: int

OFFERS
ALL INTEGERS

the first may deadlock if it has to synchronize with an identical choice, while the second will not.

SELECTION PREDICATES

```
hide sap in
        sap ?x:nat [x<max]; B1(x)
        |[sap]|
        sap ?y:nat [y>min]; B2(y)
```

max > x=y > min

This process can make internal transitions to any of the processes

```
hide sap in
        B1(n) |[sap]| B2(n)
```

with 'n' in the open interval (min, max).

GUARDED EXPRESSIONS (SEE DIJKSTRA'S "guarded commands")

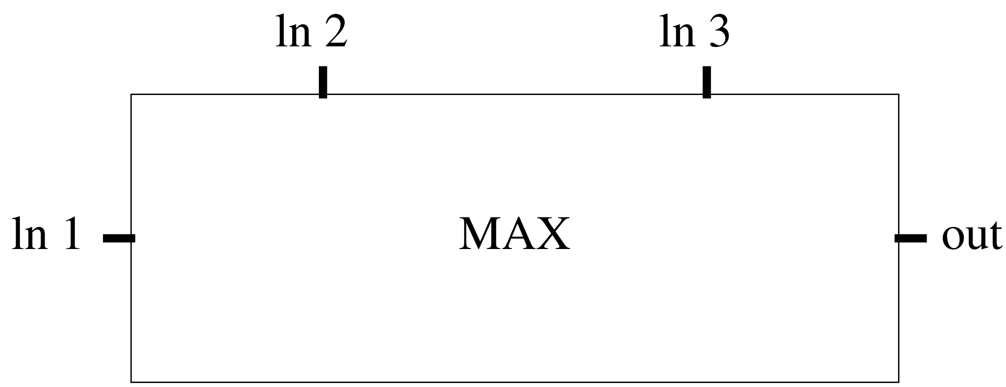
$$
\begin{array}{lll}
 & [x>0] & \to process_1 \\
[] & [x=5] & \to process_2 \\
[] & [x<9] & \to process_3
\end{array}
$$

Note: guards are not edges in the behavior tree

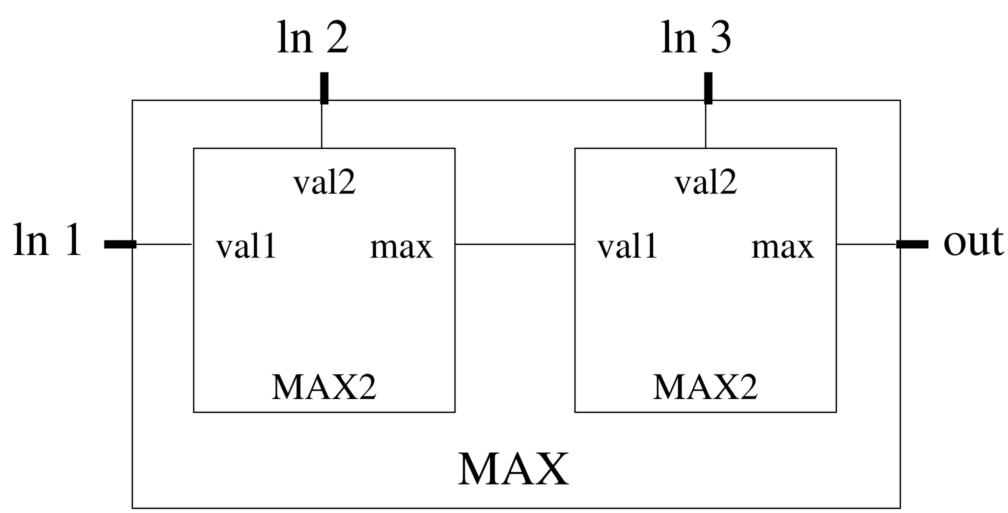
```
(* MAXIMUM OF 3 NUMBERS  Article by Bolognesi&Brinksma *)

specification Maximum[ in1, in2, in3, out ]  : noexit

type    integer is
        sorts int
        opns
           zero       :              -> int
           succ       :              -> int
           largest    :              -> int
        eqns    forall X, Y:  int
                ofsort  int
           largest   ( zero ,  X )  =  X;
           largest   ( X , zero  )  =  X;
           largest   ( succ(X) , succ(Y) )  =  succ( largest(X, Y) );
endtype

behavior
          hide mid in
            ( Max2 [in1, in2, mid]
          | [mid] |
               Max2 [mid, in3, out] )

          where process Max2 [val1, val2, max] :  noexit  :=
                      val1?X:int;   val2?Y:int;   max!largest(X, Y);   stop
                     []
                      val2?Y:int;   val1?X:int;   max!largest(X, Y);   stop
                    endproc
endspec
```

Problem: Define a Black Box that takes three natural numbers in any order, and outputs the largest of them.

note the asynchronous aspect



Decomposition: use two black boxes, each capable of taking two numbers in any order, and output the largest of them.



Synchronization A|[g]|B with values

A	B	Condition	Synchro. Type
$g ! E_1$	$g ! E_2$	$val(E_1) = val(E_2)$	value match
$g ! E_1$	$g ? x : t$	$sort(E_1) = t$	value pass E1 -> x
$g ? x : t_1$	$g ? x : t_2$	$t_1 = t_2$	value generation

Possible combinations:

!	!	processes must agree on a value
!	?	regular output-input case
?	?	neither process knows value (must be established by interaction mechanism: nondeterminism).

EXAMPLE (deadlock)

g !0 ; **exit** || g !1 ; **exit**

EXAMPLE (4 is offered to environment on gate c)

hide a, b **in**

(a ?x: nat ?y: nat;   b !largest(x, y);   **stop**

|[a,b]|

a !3 !4;   b ?x: nat;   c !x;   **stop)**

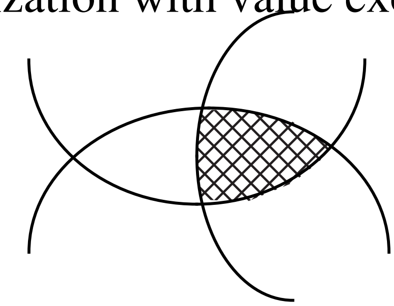
EXAMPLE (unknown nat is offered to environment on gate b)

hide a **in**

a ?x:nat; b !x; **exit**
|[a]| =   **hide** a **in** a ?x:nat; b !x; **exit**
a ?y:nat; **exit**

Full LOTOS: uses ADT notation to define data structures and operations on them.

synchronization with value exchange



Several processes get together and come up with a value agreeable to all of them (if this is not possible -> deadlock). The value is then "used" by all processes

"shriek" a ! 0

Process proposes value 0 in interaction

(if other processes cannot agree on 0 -> deadlock)

"query" a ? x: nat

Process requests a value for x of sort "nat"

(if other processes cannot agree on sort -> deadlock)

NOTE:

a ? x: nat =

a! 0 [] a! succ(0) [] a! succ(succ(0)) [] . . .

ADTs in LOTOS

The "data" part of LOTOS is a fairly "standard" ADT formalism ("Act One")

```
type   integer is
       sorts int
       opns                                        (*signature*)
         zero      :            -> int
         succ      :       int  -> int
         largest   :  int, int  -> int


      eqns forall X, Y: int ofsort int             (*equations*)
        largest ( zero , X ) = X;
        largest ( X , zero ) = X;
        largest ( succ(X) , succ(Y) ) = succ ( largest (X, Y) );
 endtype
```

Equations can be conditional:

$$Y = 0 \quad \Rightarrow \quad largest(Y , X) = X$$

COMPLETENESS

A confluent, terminating system is said to be *canonical*, or *complete*.

In such a system, we can decide equality by reducing terms to normal form.

So it is important to be able to transform an incomplete system into a complete one.

A partial procedure to do this was invented by Knuth and Bendix.

It amounts to adding rules to a system so that all proofs can be done by straightforward rewriting

(the procedure may succeed or loop forever)

In order to guarantee termination, it is necessary:

- to show that the ordering exists

- that the axioms be *oriented* according to the ordering.

That's why we write 0+x =>x rather than the converse: we are assuming an ordering where 0+x >> x.

FINITE TERMINATION

A rewriting system is *finitely terminating* when the evaluation of every term always terminates in a normal form.

e.g. obviously a system containing a rule such as:

$$x+y \Rightarrow y+x$$

is *not* finitely terminating.

Finite termination can be proven by showing that there is a well-founded ordering >> defined on the set of terms, such that:

if t is any term and $\sigma(t)$ is the result of applying any rewriting rule to t, then $t >> \sigma(t)$.

In fact, one usually proves that for every rewrite rule $A \Rightarrow B$, $A>>B$. If the ordering satisfies certain conditions (see handout), this is enough.

Orienting Axioms into Rewriting Rules

$0+x$	$\Rightarrow$	x	(R1)
$x+0$	$\Rightarrow$	x	(R2)
$(-x)+x$	$\Rightarrow$	0	(R3)
$(x+y)+z$	$\Rightarrow$	$x+(y+z)$	(R4)

Desirable properties:

- Completeness (w.r.t. given unoriented rules)
- Finite Termination
- Unique termination, or confluence

Any strategy chosen should be:

- *sound*, i.e. should not lead to wrong solutions
- as much as possible *complete*, i.e. not ignore solutions, if they are reachable with the available means

Proof Methods

- Unification: finding variable bindings that will unify an expression with one side of an axiom
  E.g. (-(-a)) is unified with x in i) above. .

- Rewriting: replacing one side of an axiom by the other side within an expression, possibly after unification.

- Strategy: Finding the sequence of axiom applications that leads to equal terms on both sides. This is difficult.

In general, this requires search and trial of all possible rewritings at each step. A vast amount of nondeterministic searching is likely to be req'd. This area has been the subject of research for many years in AI (theorem proving). In general, extensive computations are necessary and 'difficult' proof are still out of reach.

Some proof heuristics are:

- try to avoid repetition by recording the paths followed through the search space

- adopt a 'breadth first' strategy to try and find 'simpler solutions' before going to 'more complex' ones.

An Example

Given the equations:

$$
\begin{array}{rcll}
0+x & = & x & (1) \\
x+0 & = & x & (2) \\
(-x)+x & = & 0 & (3) \\
(x+y)+z & = & x+(y+z) & (4)
\end{array}
$$

Prove that $(-(-a)) = a$ for any a.

We may use the ability to freely interchange equal expressions in whatever context they may appear, to explore the properties and implications of the given equations, and prove the desired result. E.g.:

$$
\begin{array}{llll}
\text{i)} & (-(-a)) = & (-(-a))+0 & \text{by } (2) \leftarrow \\
\text{ii)} & = & (-(-a))+((-a)+a) & \text{by } (3) \leftarrow \\
\text{iii)} & = & ((-(-a))+(-a))+a & \text{by } (4) \leftarrow \\
\text{iv)} & = & 0+a & \text{by } (3) \rightarrow \\
\text{v)} & = & a & \text{by } (1) \rightarrow
\end{array}
$$

How can we automate this reasoning?

How to compute with equational axioms.

The typical problem when computing with equational axiom is determining if two terms are equivalent, by applying the axioms in any order until they are rewritten to equal terms. There are two main ways of doing this:

1. Use general theorem-proving methods. This is the most general method. However it is complex to program, computationally inefficient, and unsure to terminate.

2. Treat the axioms as oriented ***rewriting rules*** and apply them on the terms 'as far as possible'. If the 'normal forms' of the terms turn out to be identical, then we know that they are equivalent, otherwise 'we don't know'. This approach is much simpler, however by orienting the equations, certain equalities might be 'lost'.

The axioms could be written from the beginning in such a way that methods 1 and 2 lead to the same results.

If not, in certain cases, for given sets of axioms, it is possible to find a set of rewriting rules such that the two methods lead to the same results.

Computing with Equational Axioms

**(again, this class includes many slides that are not here...
will be distributed in class)**