More LOTOS Operators

TERMINATION IN LOTOS

UNSUCCESSFUL TERMINATION: STOP

off_hook ; tone ; dial ; connect ; breakdown ; stop

SUCCESSFUL TERMINATION: EXIT

off_hook ; tone ; dial   ; connect ; exit

Unsuccessful termination can be specified either explicitly by a* stop *or implicitly by the result of a deadlock.

ENABLE: ">>"

$$B_1 \;\; >> \;\; B_2$$

This is a sequential composition between two behavior expressions. Note difference w.r.t. the Action Prefix operator ";".

example:

```
off_hook ;
          (
           tone ; dial ;  exit
               [>   hang_up; stop
           )
              >>
                  connect;  talk; stop
```

The connect and talk actions will not occur if the user hangs up.

However if connect occurs, disable becomes impossible.

Exit Inference Axiom

$$\text{exit} \xrightarrow{\delta} \text{stop}$$

A process that exits stops,
after having produced action δ,
which can be picked up by other
processes, as we shall see.

Enable Inference Rules: $B_1 >> B_2$

$$\frac{B_1 - a \rightarrow B_1'}{B_1 >> B_2 - a \rightarrow B_1' >> B_2}$$

(continuation of B_1)

$$\frac{B_1 - \delta \rightarrow B_1'}{B_1 >> B_2 - i \rightarrow B_2}$$

(B_1 stops and B_2 takes over)

Note that by execution of enable δ action is *internalized*

The δ action has effects
on the parallel composition:

gate δ is always included
in the synchronization set.

Therefore ...

EXIT AND PARALLEL COMPOSITION

An exit of a process that is composed in parallel with other processes can be executed only if all processes exit

example:

(a ; b ; c; exit

|||

d ; e ; f ; stop) >> B

will not go to* B*, because the second parallel process cannot exit.

However the following example will exit:

a ; b ; c; exit

|||

(d ; e ; f ; stop   [> exit)

because this time, the second behavior can synchronize at any time on exit

exit ***is triggered by the end of the first parallel process***

any part of the second parallel process will be executed.

GENERAL PARALLELISM
INFERENCE RULES WITH EXIT

$$\frac{B_1 \ \text{-}a\text{->}\ B'_1, \qquad B_2 \ \text{-}a\text{->}\ B'_2}{B_1 \ |[g_1,\ldots,g_n]|\ B_2 \ \text{-}a\text{->}\ B'_1 \ |[g_1,\ldots,g_n]|\ B_2'} \ \text{if } a \in \{g_1,\ldots,g_n,\delta\}$$

$$\frac{B_1 \ \text{-}a_1\text{->}\ B'_1}{B_1 \ |[g_1,\ldots,g_n]|\ B_2 \ \text{-}a_1\text{->}\ B'_1 \ |[g_1,\ldots,g_n]|\ B_2} \ \text{if } a_1 \notin \{g_1,\ldots,g_n,\delta\}$$

$$\frac{B_2 \ \text{-}\ a_2\ \text{->}\ B'_2}{B_1 \ |[g_1,\ldots,g_n]|\ B_2 \ \text{-}\ a_2\ \text{->}\ B_1 \ |[g_1,\ldots,g_n]|\ B'_2} \ \text{if } a_2 \notin \{g_1,\ldots,g_n,\delta\}$$

δ is always included in every synchronization set
i cannot be included, must interleave

Normally all processes combined together by a |[...]| operator will synchro on δ.

When δ is used by >>, it becomes internal.

It can then interleave with actions of other processes involved in more external parallel composition ops.

The δ action also has effects on the disable operator

Additional disable rule:

$$\frac{B_1 - \delta \rightarrow B_1'}{B_1 [> B_2 - \delta \rightarrow B_1'}$$

i.e. if B_1 offers δ then
B_1 [> B_2 offers δ and then stops
this δ can trigger >>, [> disappears

specification ENABDISAB [a,b,c,d,e] : noexit
behavior
(a; b; exit [> c; d; stop) >> e; stop
endspec

Behavior tree:

```
1 a
| 1 b
| | 1 i (enable: exit)
| | | 1 e  DEADLOCK
| | 2 c
| | | 1 d  DEADLOCK
| 2 c
| | 1 d  DEADLOCK
2 c
| 1 d  DEADLOCK
```

Example with enable and disable

```
specification ENABDISAB [a,b,c,d,e] : noexit

behavior

   (a; b; exit  >>  c; d; stop)  [>  e; stop

endspec
```

```
* 1 a [5]
* | 1 b [5]
* | | 1 i (enable: exit) [5]
* | | | 1 c [5]
* | | | | 1 d [5]
* | | | | | 1 e [5] DEADLOCK
* | | | | 2 e [5] DEADLOCK
* | | | 2 e [5] DEADLOCK
* | | 2 e [5] DEADLOCK
* | 2 e [5] DEADLOCK
* 2 e [5] DEADLOCK
```

ENABLE EXPANSION

```
off_hook ;
        (
         tone ;
              (
                dial ;
                    (
                      i (*exit*);  connect ;  talk ; stop
                     []
                     hang_up ; stop
                     )
                 []
                hang_up ; stop
               )
          []
         hang_up ; stop
        )
```

Note:

when exit has been executed, **hang_up** ***can no longer take place in this example.***

Some interesting examples:

```
    a; b; c; exit
  ||
    (a; b; d; stop
     [> c; exit)
```

***this* exit *is triggered by the availability of a* c *after* a b.**

```
    a; b; exit
  ||
    (a; b; d; stop
     [> i; exit)
```

i ***can occur at any time (possibly yielding deadlock). However if* a b *occurs then it is forced to occur, and process will exit.***

```
    a; b; c; exit
  ||
    (a; b; d; stop
     [> i; c; exit)
```

similar.

```
    a; b; exit
  |||
    exit
```

***this will exit after* a b**

Some syntax

Basic Syntax (so far...):

behexpr = ’stop’ | ’exit’ | processname
| action ’;’ behexpr | behexpr op behexpr
op = ’[]’ | ’|[]|’ | ’[>’ | ’>>’

Examples:

stop; exit	***all syntactically invalid***
stop; a; exit	***because act. pref. joins***
exit; a	***an action and a behav. expr.***
stop >> stop	***all syntactically valid,***
stop >> exit	***although not***
exit >> exit	***semantically***
stop >> a; stop	***meaningful***
a; b [] c; d	***invalid: [] is between behav. expr.***
a; b; exit [] c; d; exit	***OK***
a; b; exit [] c; d; stop	***OK***
(a; b; stop [] c; d; exit); b; c; stop	***incorrect***
(a; b; stop [] c; d; exit) >> b; c; stop	***OK***

If P is a process name (ignoring params), the following are legal
a; b; P [] c; d; P
(a; b; stop [] c; d; exit) >> P

Operator priorities: ’;’ > ’[]’ > ’|[]|’ > ’[>’ > ’>>’
right-to-left otherwise

Multiway Synchro and Hiding

Following CSP (and unlike CCS) LOTOS adopts a **multi-way synchronization** *concept.*

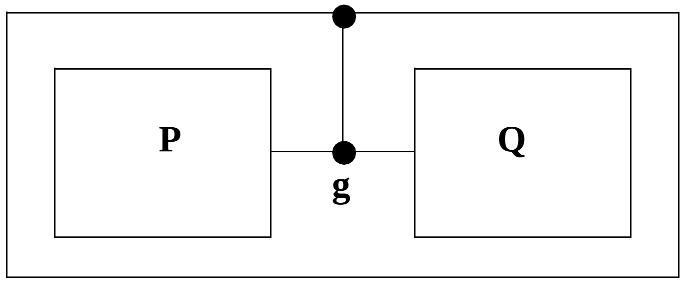
In order for an action to be executed, all behaviors that share that action by virtue of the parallel composition operator (and for which the action is not hidden, see later) must simultaneously participate in the action.

In CCS, an action becomes internal after it has been used for a two-way synchronization. In LOTOS, the action remains and can cause synchronization of a number of processes. It must be "consumed" in order to disappear. This can be achieved in one of two ways:

1. The action reaches a level where it is 'hidden'

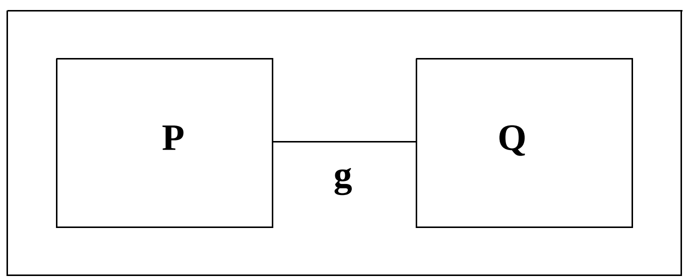
2. If 1 does not happen, the environment must participate in the action

HIDING GATES



P [g] || Q [g]

requires cooperation of environment on gate g



hide g **in** (P[g] || Q[g])

P and Q can synchronize on g without having to synchronize with the environment

The HIDE operator

hide <gatename> in B

In order for an action to be executed, all processes that synchronize on the gate must participate, unless the gate is hidden.

1) internalize actions that should not be observable

```
hide  ntwk in
  (
   phone_one[...]
     |||
   phone_two[...]
  )
    |[ntwk]|
  network[ntwk]
```

2)

```
( hide b in

     a  ;  b  ; c ; stop )

       ||

    a  ;  c  ; stop
```

is equivalent to a; i; c; stop

3)

hide c in

(c; d; stop

$||$

c; d; stop)

is equivalent to   i; d; stop

hidden actions are transformed in internal action* i *that does not need to synchronize.

HIDE ENABLES US TO ABSTRACT
or* INTERNALIZE *GATES

Hide Inference Rules

$$\frac{B \ \text{-}\ a\text{->}\ B'}{\text{hide } g_1,\ldots,g_n \text{ in } B \ \text{-}\ a\ \text{->}\ \text{hide } g_1,\ldots,g_n \text{ in } B'} \quad a \notin \{g_1,\ldots,g_n\}$$

$$\frac{B \ \text{-}\ a\text{->}\ B'}{\text{hide } g_1,\ldots,g_n \text{ in } B \ \text{-}\ i\ \text{->}\ \text{hide } g_1,\ldots,g_n \text{ in } B'} \quad a \in \{g_1,\ldots,g_n\}$$

Examples:

(hide a in a; stop |[a]| a; stop) |[a]| a; stop

the first two* a *synchronize, however the resulting* a *is transformed in internal action -> the third* a *causes deadlock

hide b in (a; stop |[a]| a; stop) |[a]| a; stop

all three* a *synchronize

PROCESS DEFINITION

PROCESS <name>[gate_list]: functionality :=

...... behavior expression

endproc

example:

```
process phone[offhook, tone, dial, talk]: exit:=
    offhook ; tone ; dial ; talk ; exit
endproc
```

A process has functionality *exit* if it can exit, it has functionality *noexit* otherwise (the precise definition is a bit complicated)..

All gates that appear in a process and are not hidden must appear in the process's parm list.

PROCESS INSTANTIATION
(inference rules for relabelling)

A sequence of action prefixes can be terminated not only by a **stop** or **exit**, but also by a *process instantiation*.

Suppose that we have process $P[g_1, \ldots, g_n]$
and an instantiation $P[h_1, \ldots, h_n]$

if process $P[g_1, \ldots, g_n]$ can execute action $\mathbf{g_i}$
then $P[h_1, \ldots, h_n]$ can execute action $\mathbf{h_i}$.

Note:

This means replacement of gate names at execution time and not at instantiation time.

The shape of the behavior tree cannot be changed by instantiation/relabelling.

RECURSION

A sequence of action prefixes can be terminated by a **stop** *or an* **exit***, or also by a* **process instantiation.** *Thus,* **recursion** *is possible.*

direct recursion:

```
process  P1 [a, b, c]: noexit:=
      a  ;   b  ;   c  ;   P1[a, b, c]
endproc
```

will generate the following sequence:

a ; b ; c ; a ; b ; c ; ....

indirect or cross recursion:

```
process  P1 [a, b, c]:noexit:=
      a  ;  b  ;  c  ;  P2[a, b, c]
endproc

  process  P2 [x, y, z]:noexit:=
      x  ;  y  ;  z  ; P1[x, y, z]
  endproc
```

Instantiating **P1[a, b, c]** *will produce the following sequence:*

a ; b ; c ; a ; b ; c ; a ; b ; c ; .....

This is because of the relabelling **[a/x, b/y, c/z]**

Note also: **a; P1[a,b,c]; P2[a, b, c]** *SYNTAX ERROR*

Although rarely done in practice,
it is possible to exchange gate parameters

```
process buffer [ in, out ];
        in;
        buffer [ out, in ]
endproc
```

in; out; in; out; ...

Interestingly, such "exchanges" can always be removed

```
process buffer [ in, out ];
        in; out;
        buffer [in, out];
endproc
```

Recursion makes it possible to define processes that cannot be described as Finite State Machines, e.g.

P := a; P [> b;stop

One way to understand such processes is to apply replacement:

P:= a; P [> b;stop
a; (a; P [> b; stop) [> b; stop
a; (a; (a; P [> b; stop) [> b; stop) [> b;stop
a; (a; (a; (a; P [> b; stop) [> b; stop) [> b; stop) [> b; stop
. . .
which implies that P can synchronize with processes offering the following sequences of actions:
b
a b
a a b
a a a b
a a a a b
... (these are caused by the last disable)

a b b
a a b b
... (these are caused by the 2nd from last disable)

in terms of formal language theory, the *language* of P can be described as

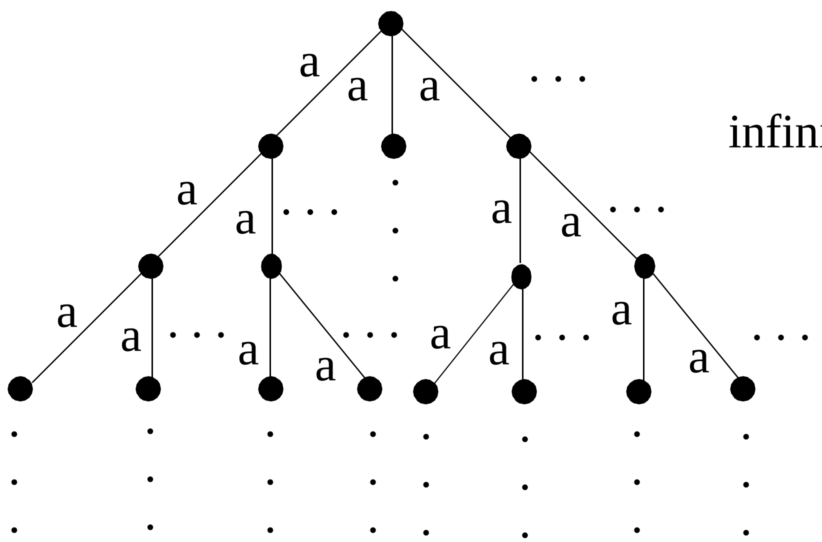
$$a^n b^m \text{ where } n+1 \geq m$$

Note also that this example shows that it may not be possible to redefine a process defined using [> by using only []

Unfortunately, with recursion, it is possible that inference rrules will not terminate on legal processes

P := a ; **stop** | | | P

= P := a; **stop** ||| a; **stop** ||| **. . .**

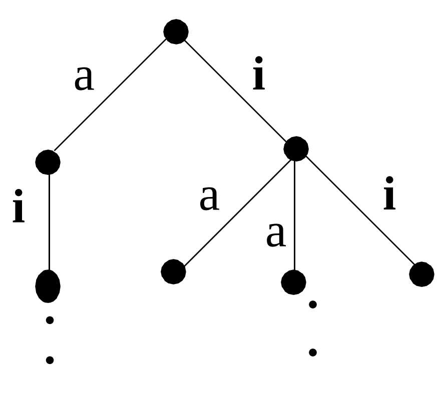


infinitely branching

This specific pitfall can be avoided by insisting that recursion be always *guarded*

e.g.

P := a ; **stop** | | | **i** ; P



finitely branching
(but still infinite)

This example shows a common pitfall in LOTOS: since process instantiation is not an action, nor a state in the LTS, in the case of ‘unguarded recursion’, finding the next action may require an infinite number of recursive instantiations, hence the derivation process becomes infinite.

Another similar example is:

P := a; **stop** [] P

Internal actions stop the derivation process:

P := a; **stop** [] i; P

Time-bomb process!

Keep_out := Time [> Bomb

Time := tick; Time

Bomb := boom; **stop** ||| Bomb

If you want to include an escape...

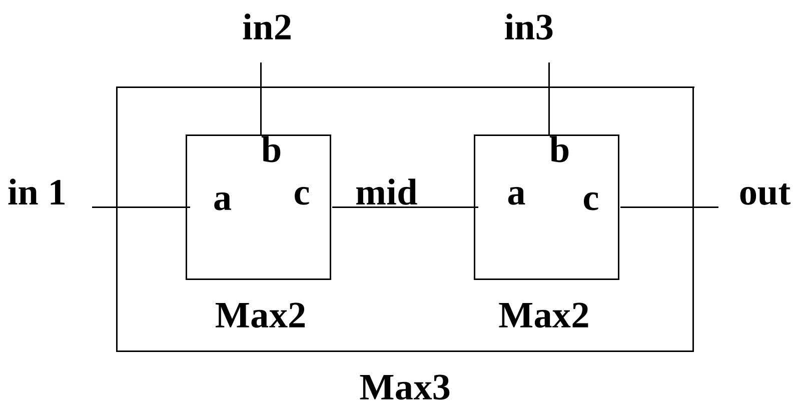
Keep_out := (Time [> Bomb) >> Run

Time := tick;Time [] **exit**

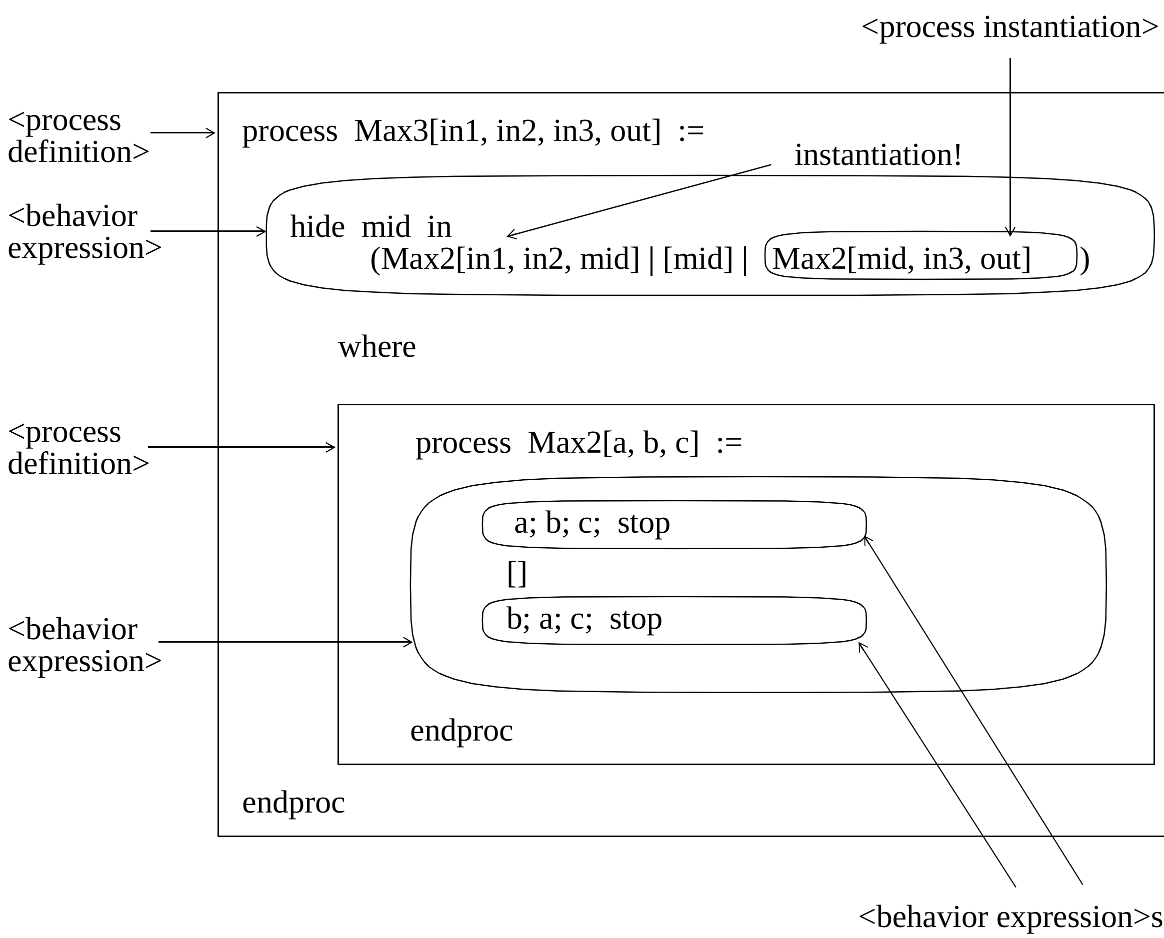
Bomb := boom; **stop** ||| Bomb

Run := run; Run [] **stop**

Infinite recursion impedes finite evaluation and can be prevented by always *guarding* (by any action, even internal) every process instantiation.



Spatial representation of process Max3 (Bolognesi + Brinksma)

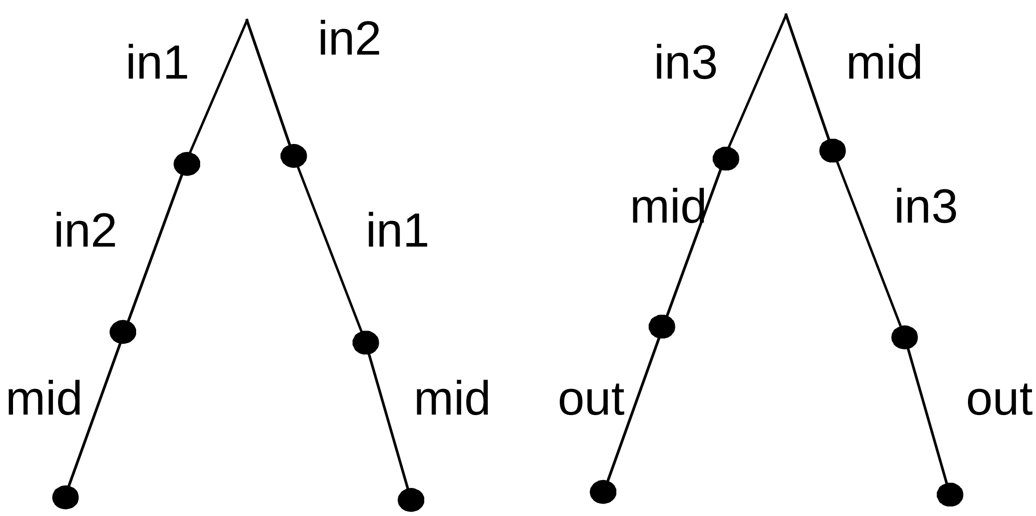


Definition of process Max3

```
process  Max2[a, b, c]  :=

        a; b; c;  stop
        []
        b; a; c;  stop

endproc
```

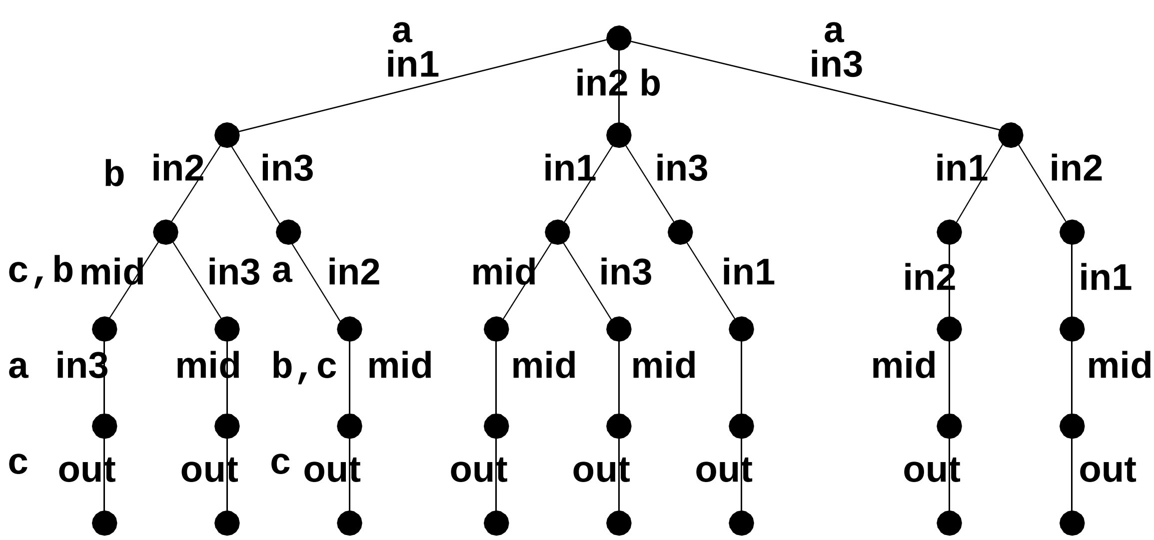
instantiations:

Max2[in1, in2, mid]   Max2[in3, mid, out]



The two behavior trees

```
process  Max3[in1, in2, in3, out]  :=

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

       where

          process  Max2[a, b, c]  :=

                a; b; c;  stop
                []
                b; a; c;  stop

          endproc

endproc
```



Ken Turner’s LIFE Specification

```
(* Ken Turner’s "Life" specification *)

specification LIFE
  [birth, puberty, death, marriage, children]: exit

behavior

   BIOLOGY [birth, puberty, death]
  |[puberty, death]|
   FAMILY [puberty, marriage, children, death]

   where

     process BIOLOGY [birth, puberty, death]: exit :=
       birth; ((puberty; stop) [> (death; exit))
     endproc

     process FAMILY
       [puberty, marriage, children, death] : exit :=

       (
         puberty;  ((marriage; exit) [] exit)
        |[puberty]|
         puberty;  ((children; exit) [] exit)
       )
      |[marriage]|
       (( marriage; stop) [> (death; exit ))
      endproc

endspec
```

Behavior tree of LIFE specification

```
1 birth [14]
| 1 puberty [14,20,22]
| | 1 children [22]
| | | 1 marriage [20,25]
| | | | 1 death [14,25]
| | | | | 1 exit ** EXIT SUCCEED ** [14,20,22,25]
| | | 2 death [14,25]
| | | | 1 exit ** EXIT SUCCEED ** [14,20,22,25]
| | 2 marriage [20,25]
| | | 1 children [22]
| | | | 1 death [14,25]
| | | | | 1 exit ** EXIT SUCCEED ** [14,20,22,25]
| | | 2 death [14,25]
| | | | 1 children [22]
| | | | | 1 exit ** EXIT SUCCEED ** [14,20,22,25]
| | | | 2 exit ** EXIT SUCCEED ** [14,20,22,25]
| | 3 death [14,25]
| | | 1 children [22]
| | | | 1 exit ** EXIT SUCCEED ** [14,20,22,25]
| | | 2 exit ** EXIT SUCCEED ** [14,20,22,25]
| 2 death [14,25] DEADLOCK
```

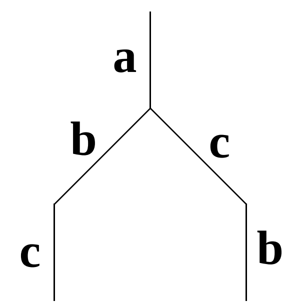
RELABELLING

difference with substitution

example:

```
process P[a,b,c] :=
   a ; b ; stop
   |[a]|
   a ; c ; stop
endproc
```

synchronization tree

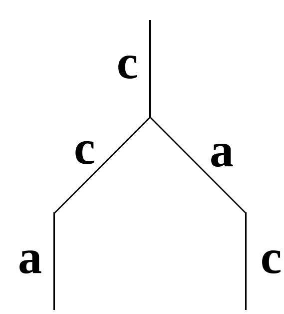


The principle is: create the synchronization tree first and then relabel its actions.
After instantiating with P[c,c,a]

substitution tree

wrong !

relabelling tree



correct !

note the significant differences in resulting trees

Note however:

This difference between substitution and relabelling is observable only for gate mappings (formal <-> actual) that are not one-to-one.

Most mappings used in practice are one-to-one, so the difference cannot be observed.

```
 1 (* Example to show effect of different gate parameterization
*)
 2
 3
 4 specification gates [a,b,c]: noexit
 5
 6 behavior
 7
 8 p[a,b,c]    (* p[a,b,a] *)
 9
10 where
11
12      process p[a,b,c]: noexit:=
13         q[a,b] |[a]| r[a,c]
14      endproc
15
16      process q[a,b]: noexit:=
17         a; b; stop
18      endproc
19
20      process r[a,c]: noexit:=
21         c; a; stop
22      endproc
23
24 endspec
25
```

```
(* as written *)

bh0 * 1 c line(s): [21]
bh1 * | 1 a line(s): [17,21]
bh2 * | | 1 b line(s): [17]  DEADLOCK

(* changing line 8 to p[a,b,a] *)

bh0 * 1 a line(s): [21]
bh1 * | 1 a line(s): [17,21]
bh2 * | | 1 b line(s): [17]  DEADLOCK
~
```

Example with gate parms and hiding

process vending_machine [coin, candy1, candy2] :=
    coin;
    (candy1; vending_machine [coin, candy1, candy2]
        [ ]
    candy2; vending_machine [coin, candy1, candy2])
endproc

process devil [candy] :=
    candy;
    devil [candy]
endproc

process system [coin, candy] :=
    hide candy_bar in
        (vending_machine [coin, candy, *candy_bar*]
        | [candy_bar] |
        devil [*candy_bar*])
endproc

$\approx$

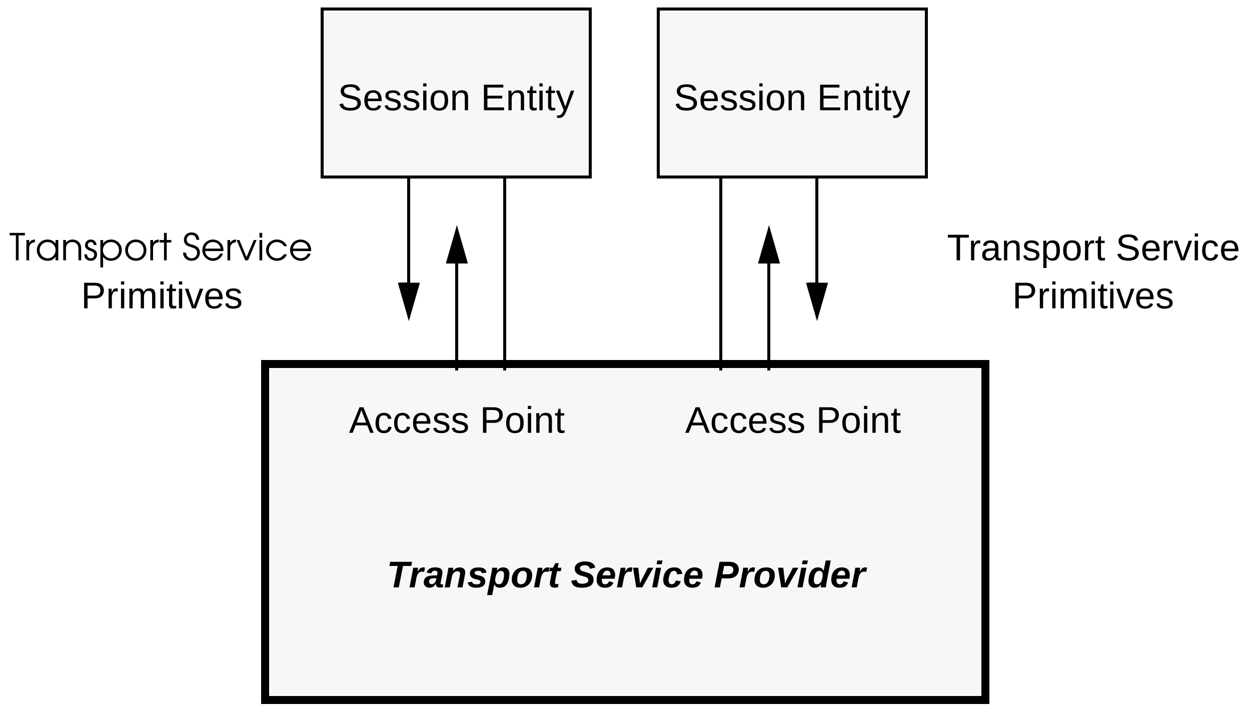
process system [coin, candy] :=
    coin; (candy; system [coin, candy]
        [ ]
    i ; system [coin, candy]
)
endproc

The exact meaning of ≈ is to be discussed.

An almost realistic example in Basic LOTOS:

OSI Transport Service Provider

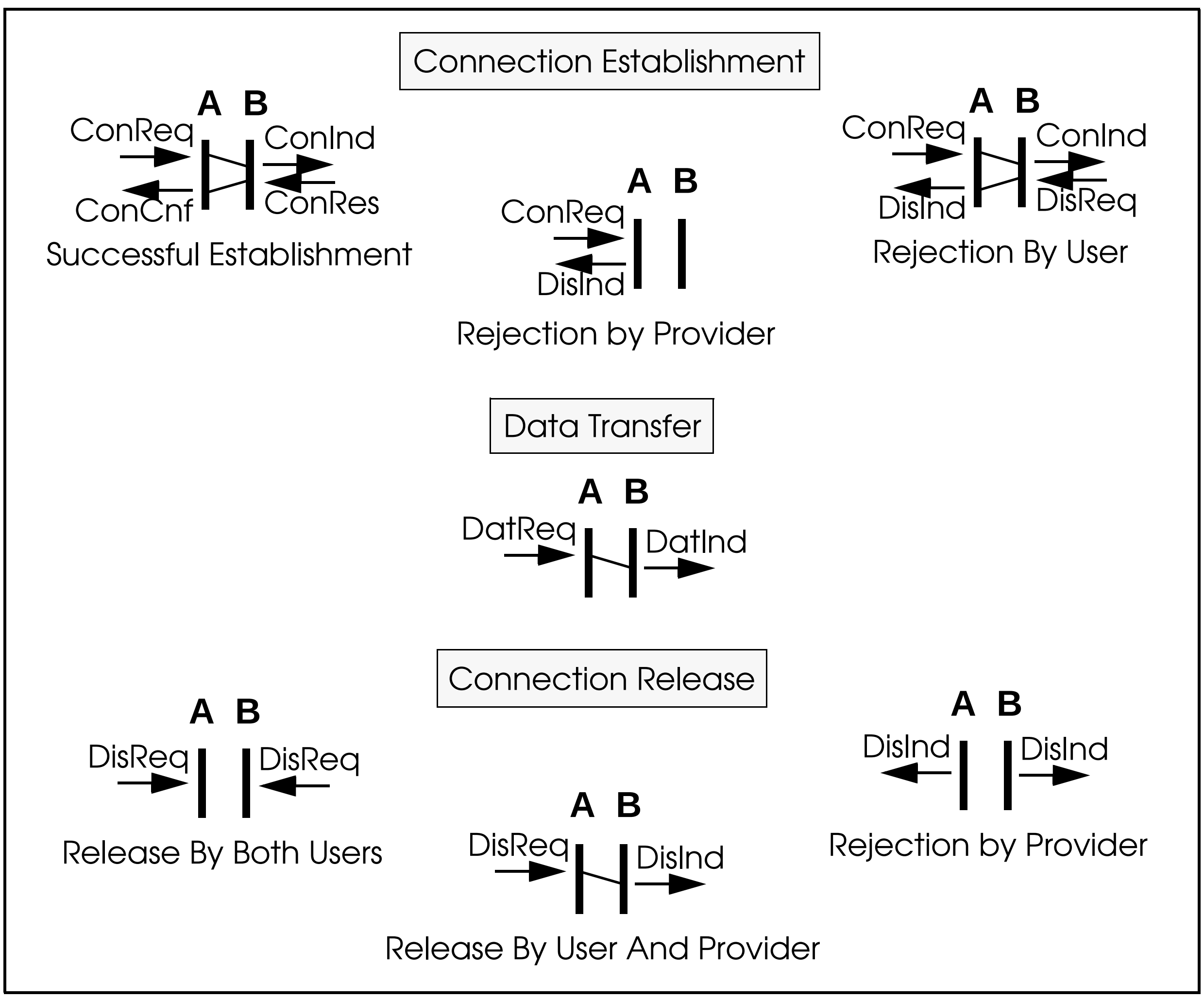
(from paper by Bolognesi and Brinksma)


Session Entity
Session Entity
Transport Service Primitives
Transport Service Primitives
Access Point
Access Point
Transport Service Provider

The following table describes the service primitives considered and their significance:

Primitives	Significance
ConReq	Connection Request
ConInd	Connection Indication
ConRes	Connection Response
ConCnf	Connection Confirmation
DisReq	Disconnection Request
DisInd	Disconnection Indication
DatReq	Data Request
DatInd	Data Indication

The service primitive sequences are expressed by Message Sequence Charts:



Process	Phase	Primitives involved
Connection-Phase	**Connection establishment**	***ConReq, ConInd, ConRes, ConCnf, DisReq, DisInd***
Data_phase	**Data Transfer**	***DatReq, DatInd***
Termination_ phase	**Connection release**	***DisReq, DisInd***

The main behaviour of the specification can then be written as a call to process *Handler*:

Handler[*ConReq, ConInd, ConRes, ConCnf, DatReq, DatInd, DisReq, DisInd*]

where process *Handler* is defined as follows:

Connection_Phase[ConReq, ConInd, ConRes, ConCnf, DisReq, DisInd]
>>
(Data_phase[DatReq, DatInd]
[>
Termination_phase[DisReq, DisInd])
>>
Handler[ConReq, ConInd, ConRes, ConCnf, DatReq, DatInd, DisReq, DisInd]

Data_phase is only enabled when process *Connection_Phase* terminates successfully. On the other hand, process *Termination_phase* can disrupt process *Data_phase* at any time before it terminates. Once process *Termination_phase* exits, it enables recursively the whole service again by calling process *Handler.*

```
process Handler[ConReq, ConInd, ConRes, ConCnf, DatReq, DatInd, DisReq, DisInd]:=

    Connection_Phase[ConReq, ConInd, ConRes, ConCnf, DisReq, DisInd]
    >> (        Data_phase[DatReq, DatInd]
        [>      Termination_phase[DisReq, DisInd]
        )
        >>      Handler[ConReq, ConInd, ConRes, ConCnf, DatReq, DatInd, DisReq, DisInd]

where
        process Connection_Phase[CRq, CI, CR, CC, DR, DI] :=

                (   i; Calling[Rq, CI, CR, CC, DR, DI]
                []  Called[Rq, CI, CR, CC, DR, DI]
                )
        where
                process Calling[Rq, CI, CR, CC, DR, DI]:=

                    CRq;    (   CC; exit
                            []  DI; Connection_Phase[CRq, CI, CR, CC, DR, DI]
                            )
                endproc (* Calling *)

                process Called[Rq, CI, CR, CC, DR, DI]:=

                    CI;     (   i; CR; exit
                            []  i; DR; Connection_Phase[CRq, CI, CR, CC, DR, DI]
                            )
                endproc (* Called *)

        endproc (* Connection_Phase *)

        process Data_phase[DtR, DtI]

                i; DtR; Data_phase[DtR, DtI]
            []  DtI; Data_phase[DtR, DtI]

        endproc (* Data_phase *)

        process Termination_phase[DR, DI] :=

                i; DR; exit
            []  DI; exit

        endproc (* Termination_phase *)

endproc (* Handler *)
```

Concepts discussed in Class 3:

- termination: exit and stop
- exit and enable
- exit, enable, disable, and par. composition
- hiding and abstraction
- process definition and instantiation
- gate relabeling
- recursion
- pitfalls of unguarded process instantiation
- instantiation,relabeling and parallelism
- the transport service provider in basic LOTOS