Semantics of
LOTOS operators

Main LOTOS operators

(operators are used to combine actions and behavior expressions into more complex b.ex.)
(let $\mathbf{a}, \mathbf{a_1}, \mathbf{a_2}$,.. be actions, and $\mathbf{B}, \mathbf{B_1}, \mathbf{B_2}$... be behavior expressions).

a; B: the *action prefix* operator “**;**” means that action **a** is offered and then actions from behavior **B** are offered.

$\mathbf{B}_1$ **[]** $\mathbf{B}_2$: the *choice* operator means that the next action can be obtained either from $\mathbf{B}_1$ or from $\mathbf{B}_2$. The other behavior is discarded.

$\mathbf{B}_1$ ||| $\mathbf{B}_2$: the *interleaving* operator means that at any point actions from $\mathbf{B}_1$ or from $\mathbf{B}_2$ can be offered.

$\mathbf{B}_1$ || $\mathbf{B}_2$: the *full synchronization* parallel operator means that every action must be a common action from $\mathbf{B}_1$ and $\mathbf{B}_2$. At each step, if such an action exists, it is offered (synchronization) and then the next common action must be obtained similalry and so on.

$\mathbf{B}_1$ **|[**$\mathbf{a}_1$**,** $\mathbf{a}_2$**,...** $\mathbf{a}_n$**]|** $\mathbf{B}_2$: the *general parallel* operator is a generalization of the full synchronization and interleave operators. It means that on actions $\mathbf{a_1}$**,...,**$\mathbf{a_n}$, $\mathbf{B}_1$ and $\mathbf{B}_2$ must synchronize; on other actions $\mathbf{B}_1$ and $\mathbf{B}_2$ interleave.

$\mathbf{B}_1$ **[>** $\mathbf{B}_2$: the *disable* operator means that at any time during the execution of $\mathbf{B}_1$, $\mathbf{B}_2$ can take over, thus terminating $\mathbf{B}_1$.

$\mathbf{B}_1$ >> $\mathbf{B}_2$: the *enable* operator means that provided that $\mathbf{B}_1$ has completed successfully (**exit**), $\mathbf{B}_2$ can start offering actions.

hide a in B: it internalizes actions **a** occurring in **B**.

INFERENCE RULES

LOTOS operational (=dynamic) semantics is expressed in terms of inference rules of the general form

$$\frac{B_1 \text{ -}a_1\text{-> } B_1' \qquad B_2 \text{ -}a_2\text{-> } B_2' \ \ldots}{C \text{ -}b\text{-> } C'}$$

Meaning:

If ***behavior expression*** B_1 ***can transform to*** B_1' ***by execution of action*** a_1 **and** B_2 ***can transform to*** B_2' ***by action*** a_2 ***etc. ...***

Then ***behavior expression C can transform to C' by execution of action*** **b**

There are also **inference axioms*****, meaning that certain actions can occur unconditionally in certain situations (empty premiss).***

Note: for non-internal actions, execution ***is synonim with*** synchronization with environment ***on the action.***

Inference rules are becoming one of the standard forms of expressing the **operational semantics** ***of computer languages.***

The idea of using inference rules in this way is due to Plotkin. It was applied to process algebras by Milner, and Hennessy wrote a whole book about it.

The Inference Rule for the ; (action prefix) operator is

a simple Inference Axiom:

$$a \; ; B \xrightarrow{\;a\;} B$$

Meaning:

when action a is executed in behavior a ; B
then what remains to be done is B.

Note: in order for action a to be executed, the environment must *participate* in it (= *synchronize* with it)

E.G. : given the behavior expression: coin; coffee; **stop**

$$
\begin{aligned}
&\text{coin} \; ; \text{coffee} \; ; \mathbf{stop} \\
&\xrightarrow{\;\text{coin}\;} \text{coffee} \; ; \mathbf{stop} \\
&\qquad \xrightarrow{\;\text{coffee}\;} \mathbf{stop}
\end{aligned}
$$

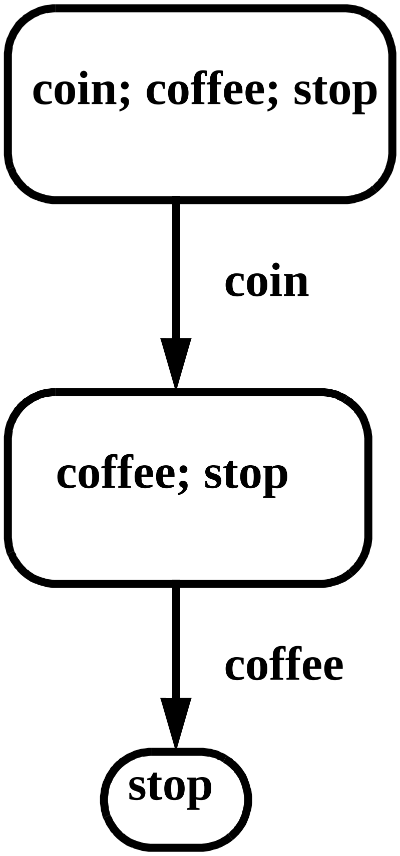
(we call these
<u>derivations</u>)

There is no inference rule for **stop** so execution ends.

The behavior expression has resulted in the *trace*

coin coffee

by applying twice the inference axiom for the action prefix operator



There is a direct relationship between inference rules and behavior trees.

One can think that the arrow shown in the inference rule is also the arrow shown in the behavior tree.

CHOICE: B_1 [] B_2

where B_1 and B_2 are behavior expressions.

Example:

A phone station can be represented as a choice between two behaviors:

off_hook ; tone ; dial ; talk ; stop

[]

ring ; answer ; talk ; stop

The first behavior is a call initiator.
The second behavior is a call responder.

The environment and the station together determine the behavior of the phone station, by cooperating on the first action of one of the two choices.

In order to obtain the set of all possible initial actions ***from this behavior expression, we have to apply the inference rules on both behaviors B_1 and B_2 individually.***

We then find action prefix operators yielding:

off_hook ***and*** **ring** ***as the two possible first actions.***

Different behaviors will result from each of these choices.

Simplified Inference Rule:

$$\frac{B_1 \quad -a_1\text{->} \quad B'_1}{B_1 \,[]\, B_2 \quad -a_1\text{->} \quad B'_1}$$

$$\frac{B_2 \quad -a_2\text{->} \quad B'_2}{B_1 \,[]\, B_2 \quad -a_2\text{->} \quad B'_2}$$

When it has been determined which first action to execute, the remaining next possible actions can only be the actions of the resulting behavior.

Suppose $B_1 = a_1; B'_1$

***If action* a_1 *is selected, the next action will have to come from behavior expression* B'_1.**

***Selecting the first action of* B_2 *(suppose it is* a_2*) will result in next actions coming from* B'_2 .**

If neither B_1 nor B_2 is **stop**, *then they must be of the form a_i ; B'_i and we say that*

B_1 [] B_2 *can be* **expanded** *as* **a_1 ; B'_1 [] a_2 ; B'_2**

and also that the following derivations are possible:

$$a_1 \ ; \ B'_1 \ [] \ a_2 \ ; \ B'_2 \ \text{-} \ a_1 \text{->} \ B'_1$$

or $$a_1 \ ; \ B'_1 \ [] \ a_2 \ ; \ B'_2 \ \text{-} \ a_2 \text{->} \ B'_2$$

Obviously, if either **B_1** *or* **B_2** *is* **stop** *(i.e. it has no action), the action will have to come from the other behavior:*

$$B_1 \ [] \ \text{stop} = B_1$$
$$\text{stop} \ [] \ B_2 = B_2$$

so only one of the two derivations above is possible.

For example, in

off_hook ; tone ; dial ; talk ; stop **(B_1)**

[]

ring ; answer ; talk ; stop **(B_2)**

it is possible to use either inference rule

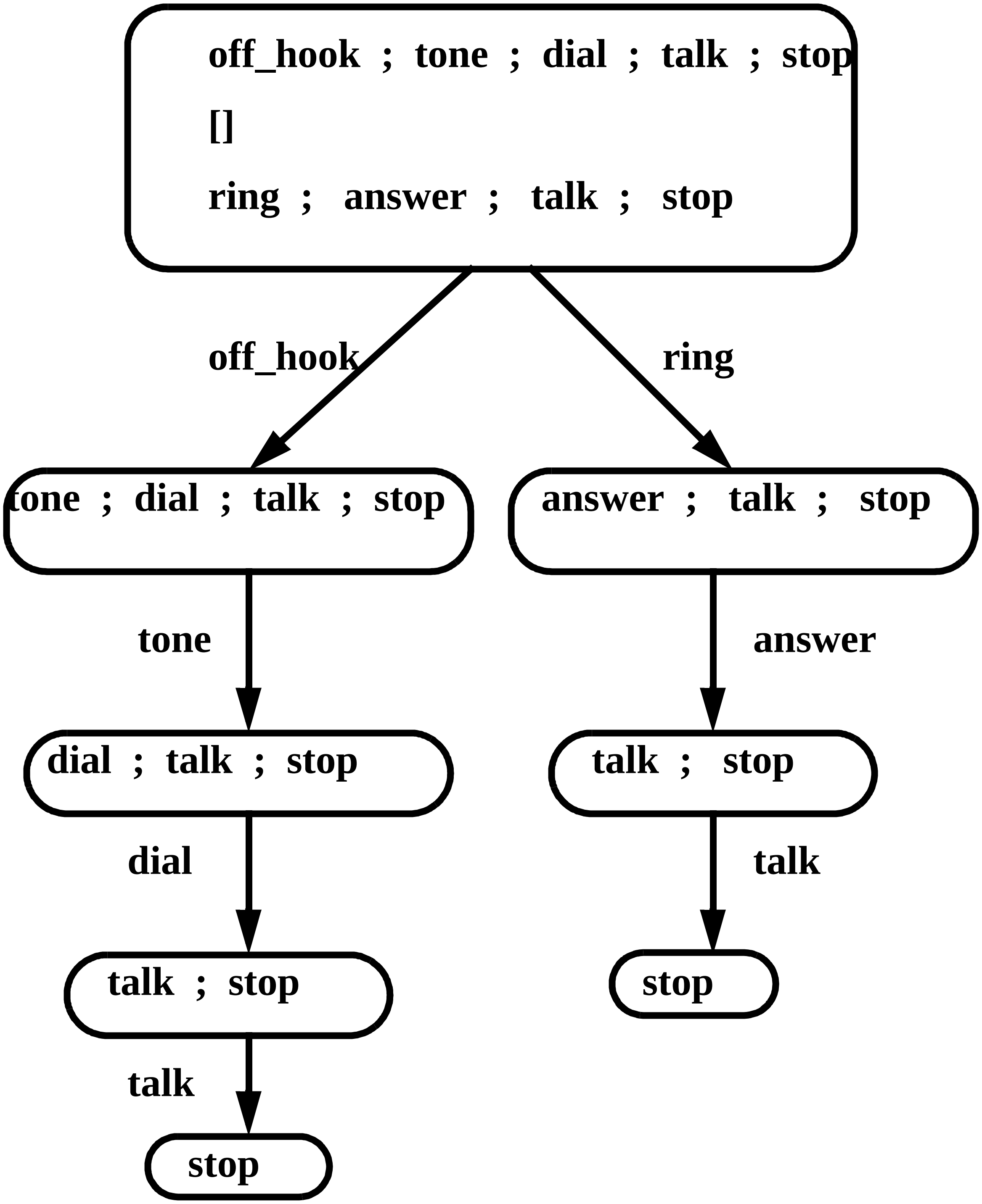
$$\frac{B_1 \text{ - off\_hook-> (tone ; dial ; talk ; stop)}}{B_1 \text{ [] } B_2 \text{ -off\_hook -> (tone ; dial ; talk ; stop)}}$$

or

$$\frac{B_2 \text{ - ring -> (answer ; talk ; stop)}}{B_1 \text{ [] } B_2 \text{ - ring -> (answer ; talk ; stop)}}$$

The environment and the process decide together which inference rule is executed by synchronizing on **off_hook** *or* **ring.**

In terms of behavior trees...



NONDETERMINISM

coin ; chocolate ; **stop**

[ ]

coin ; candy ; **stop**

If the environment offers coin, either inference rule can be applied.

The process must decide which way to go.

How?

> The specification does not say

Nondeterminism is a powerful specification concept:

> It means that at the specification level,
> we don’t care how the choice is handled.

i ***can be used to model nondeterminism:***

```
off_hook ;
          ( tone ; ...

            []

            i ; line_down ; stop
         )
```

user can get either tone or silence depending on an internal event.

One can say that line_down ***has priority, in the sense that if the env't proposes it, it will occur.***

But if the env't proposes tone ***after the process has decided to do the internal action,*** tone ***will be refused.***

NONDETERMINISM IN LOTOS

Nondeterminism occurs when at some point, considering what the environment and the process offer, several inference rules are applicable.

coin; coffee; **stop** [] coin; choc; **stop**	will always accept *coin* but subsequently will accept only one of *choc* or *coffee,* dep. on nondetermin. choice
coin; (**i**; coffee; **stop** **[]** choc; **stop)**	after *coin* will always accept *coffee,* may or may not accept *choc*, depending on whether it decides to do **i**
coin; (**i**; coffee; **stop** [] **i**; choc; **stop**}	after *coin* may or may not accept *coffee* or *choc,* dep. on nondet. choice

i can be executed without participation of the env't and will be executed eventually if it is the only possible choice (the inference rules will stop only when none is applicable).

Note that **i** *can either be specified explicitly, or arise from the execution of* hidden *actions.*

Can we consider that the first and last case are equivalent?
We shall see...

INTERLEAVED PARALLELISM: |||

$$B_1 \;|||\; B_2$$

example:

the behavior of two phone stations placing a call

```
phone_one_off_hook ;
phone_one_tone ;
phone_one_dial ;
phone_one_talk ;
stop

|||

phone_two_off_hook ;
phone_two_tone ;
phone_two_dial ;
phone_two_talk ;
stop
```

Each phone behaves totally independently from the other. This means the actions of the two phones can mutually interleave in all possible ways.

INTERLEAVE EXPANSION

This is a partial expansion of the interleaved expression.

```
phone_one_off_hook ;
                                 ( phone_one_tone ;
                                                            (phone_one_dial ; ....
                                                                [] 
                                                             phone_two_off_hook ; ....
                                                            )
                                     []
                                    phone_two_off_hook ;
                                                                    ( phone_one_tone ; ....
                                                                       []
                                                                      phone_two_tone ; ....
                                                                     )
                                 )
[]

phone_two_off_hook ;
                                 ( phone_two_tone ;
                                                            (phone_two_dial ; ....
                                                                []
                                                             phone_one_off_hook ; ....
                                                            )
                                     []
                                    phone_one_off_hook ;
                                                                    ( phone_two_tone ; ....
                                                                       []
                                                                      phone_one_tone ; ....
                                                                     )
                                 )
```

INTERLEAVE OPERATOR
SIMPLIFIED INFERENCE RULES

$$\frac{B_1 \;\; - a_1\text{->} \;\; B'_1}{B_1 \;\; ||| \;\; B_2 \;\; - a_1 \text{->} \;\; B'_1 \;\; ||| \; B_2}$$

$$\frac{B_2 \;\; - a_2 \text{->} \;\; B'_2}{B_1 \;\; ||| \;\; B_2 \;\; - a_2 \text{->} \;\; B_1 \;\; ||| \; B'_2}$$

In short: when one selects an action from one behavior the whole other behavior is still active.

Note the difference w.r.t. the choice operator **[] .**

The choice operator **[]** ***implies comitment to the chosen branch.***

The interleave operator **|||** ***implies that it is always possible to take an action from any of the participating processes.***

(We are ignoring synchro. on exit: see later)

DEPENDENT PARALLELISM: ||

$B_1 \;\|\; B_2$

Every visible action of B_1 *has to* synchronize *with a visible action in* B_2. Each of B_1 and B_2 acts as the env’t of the other.

Example: a phone station and its controller.

```
off_hook ;
         (
           tone  ;   dial   ;  talk  ;  stop
          []
           dial   ;   tone  ;  talk  ;  stop
         )


||

offhook  ;   tone  ;   dial  ;   talk  ;   stop
```

No call can be placed before the controller has received
an off_hook *signal from the station. Then the user has*
to wait for a dial tone before dialing.

The second choice of user behavior will not synchronize with what the controller is expecting.
The result of this behavior expression is:

```
offhook  ;  tone  ;  dial  ;  talk  ;  stop
```

Note how the choice has been resolved by the second process.

INTERNAL ACTION i

Internal actions denote internal events of the system.
Are not **visible.**

a ; i ; b ; stop
||
a ; b ; stop

a and b will synchronize because i, being an internal action,does not need to synchronize

DEPENDENT PARALLELISM
SIMPLIFIED INFERENCE RULES

$$\frac{B_1 \ \text{- a ->}\ B'_1 \qquad B_2 \ \text{- a ->}\ B'_2}{B_1 \ \|\ B_2 \ \text{- a ->}\ B'_1 \ \|\ B'_2} \quad a \neq i$$

Each action of one process must match (= synchronize with) a corresponding action in the other process.

After matching, both processes go to their next behavior expression.

(for simplification, we are ignoring synchronization on exit, also there is nothing that says that a process can independently offer internal actions - see later)

DEADLOCK

Deadlock is expressed in LOTOS as **stop**

It corresponds to the case when no "next action" is possible.

i.e., no inference rule can be applied.

e.g.

```
off_hook ;
          (
            tone ; dial ; talk ; stop
           []
           dial; tone ; talk ; stop
         )
 ||
   off_hook; tone; onhook ; offhook ; tone ; dial ; talk;
   stop
```

is equivalent to:

```
off_hook ; tone ; stop      (= DEADLOCK)
```

a simpler example:

```
a; b; stop || c; b; stop  =  stop
```

If there are inference rules that can be executed, there is no deadlock

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

||

c; d; stop

this is equivalent to c;d; stop

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

||

c; d; stop [] d; f; stop

this is also equivalent to c; d; stop

However there is no "look-ahead' and premature deadlock can result later:

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

||

a; b; stop

this is equivalent to a; b; stop [] a; stop

In other words, the environment can interact successfully with P||Q iff the sequence of events it provides satisfies **both** ***P and Q***

This is the main idea of **constraint-oriented specification** ***in LOTOS.***

It comes from Hoare's CSP.

Examples involving nondeterminism

a; stop || (i; a; stop [] i; b; stop) = i; a; stop [] i; stop

(a; stop [] i; b; stop) || (a; stop [] i; b; stop) =
a; stop [] i; i; b; stop [] i; i; b; stop

as we’ll see later, this can be simplified to
a; stop [] i; i; b; stop
and further to
a; stop [] i; b; stop

AND UNFORTUNATELY (perhaps)

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

expands to

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

which can be simplified to

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

meaning that, in the presence of certain kinds of nondeterminism, A||A ≠ A (|| is not like a logical AND operator)

GENERAL PARALLEL COMPOSITION: $|[g_1, g_2]|$

In

$$P|[g_1, g_2]|\ Q$$

P and Q must synchronize on gates $g_1$and g_2, interleave on all others.

Example:

(a; b; stop [] c; d; stop) |[a,b]| (a; b; stop [] d; f; stop)

=

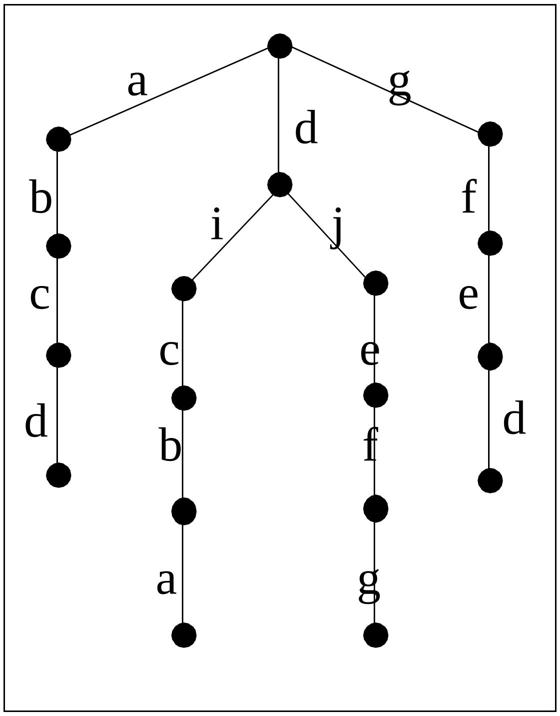
a; b; stop [] (c; d; stop ||| d; f; stop)

=

a; b; stop []
(c;(d;d;f;stop [] d;(d;f;stop [] f;d;stop)) []
d;(c;(d;f;stop [] f;d;stop)[] f;c;d;stop))

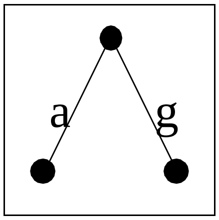
PARTIAL SYNCHRONIZATION

$$| [g_1, . . . , g_n] |$$

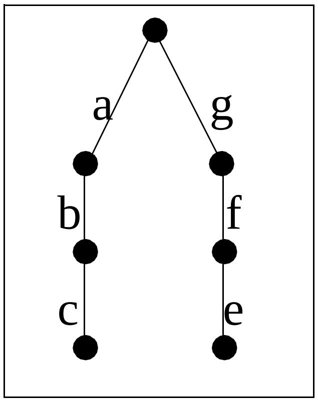


| [a, d, g] |

(synchronize on a,d,g
interleave on all others)



⊜



NOTE: IF THERE IS CHOICE, POSSIBILITIES RESULTING
IN "STOP" ARE DISCARDED.

GENERALIZED PARALLEL COMPOSITION: $|[g_1,...,g_n]|$

$$B_1 \quad |[g_1,..,g_n]| \quad B_2$$

Processes B1 *and* B2 *must synchronize on actions that appear in the action list* [g1,...,gn] *but interleave on the remaining actions.*

For example a controller process is in parallel with a call initiator phone and a call responder phone. Normally it will interact with both phones in various sequences but not at the same time.

This is the high level behavior of a primitive phone system:

call_initiator_phone[...]

|[***conreq, conconf,connect_initiator***]|

controller[...]

|[*ring,connect,connect_responder*]|

call_responder_phone[...]

The controller will "talk" and agree with both phones on different actions

The controller expects to agree with the call initiator on a connection request action (conreq)

it will then turn to the call responder side and expects to agree on a ring

it will then eventually agree on connecting the responder and confirming the connection to the initiator.

```
        call_initiator[...]  |[conreq, conconf]|
        controller[...]  |[ring,connect]|
        call_responder [...]
where
  process  call_initiator [...]:noexit:=
    off_hook ;  tone  ; dial   ; conreq ;  conconf ; talk1 ; stop
  endproc
  process  controller[...]:noexit:=
    conreq  ;  ring  ;  connect  ;  conconf  ;  stop
   endproc
   process    call_responder [...]:noexit:=
     ring  ;  answer  ;  connect  ;  talk2  ; stop
   endproc
```

call_initiator[...] **|[*conreq, conconf*]|**
controller[...] **|[*ring,connect*]|**
call_responder [...]

where

process call_initiator [...]:noexit:=

off_hook ; tone ; dial ; conreq ; conconf ; talk1 ; stop

endproc

process controller[...]:noexit:=

conreq ; ring ; connect ; conconf ; stop

endproc

process call_responder [...]:noexit:=

ring ; answer ; connect ; talk2 ; stop

endproc

Applying the inference rules will result in the following sequence of actions:

off_hook, tone, dial : by the call initiator because they are the only initially independent actions
conreq: is the first action on which both the call initiator and the controller have to agree on in order to mutually proceed.
ring: is the next common action controller / responder
answer: the only action that can be executed independently after ring
connect follows for similar reasons
talk 2 or conconf are equally possible afterwards

The ordering of actions is given explicitly by this expansion*:*

```
off_hook  ;  tone  ;  dial  ;
conreq  ;  ring  ;  answer  ;  connect  ;
( conconf   ;
   (
     talk2  ;  talk1  ;  stop
      []
     talk1  ;  talk2  ;  stop
   )
  []
   talk2  ;  conconf  ;  talk1  ; stop
)
```

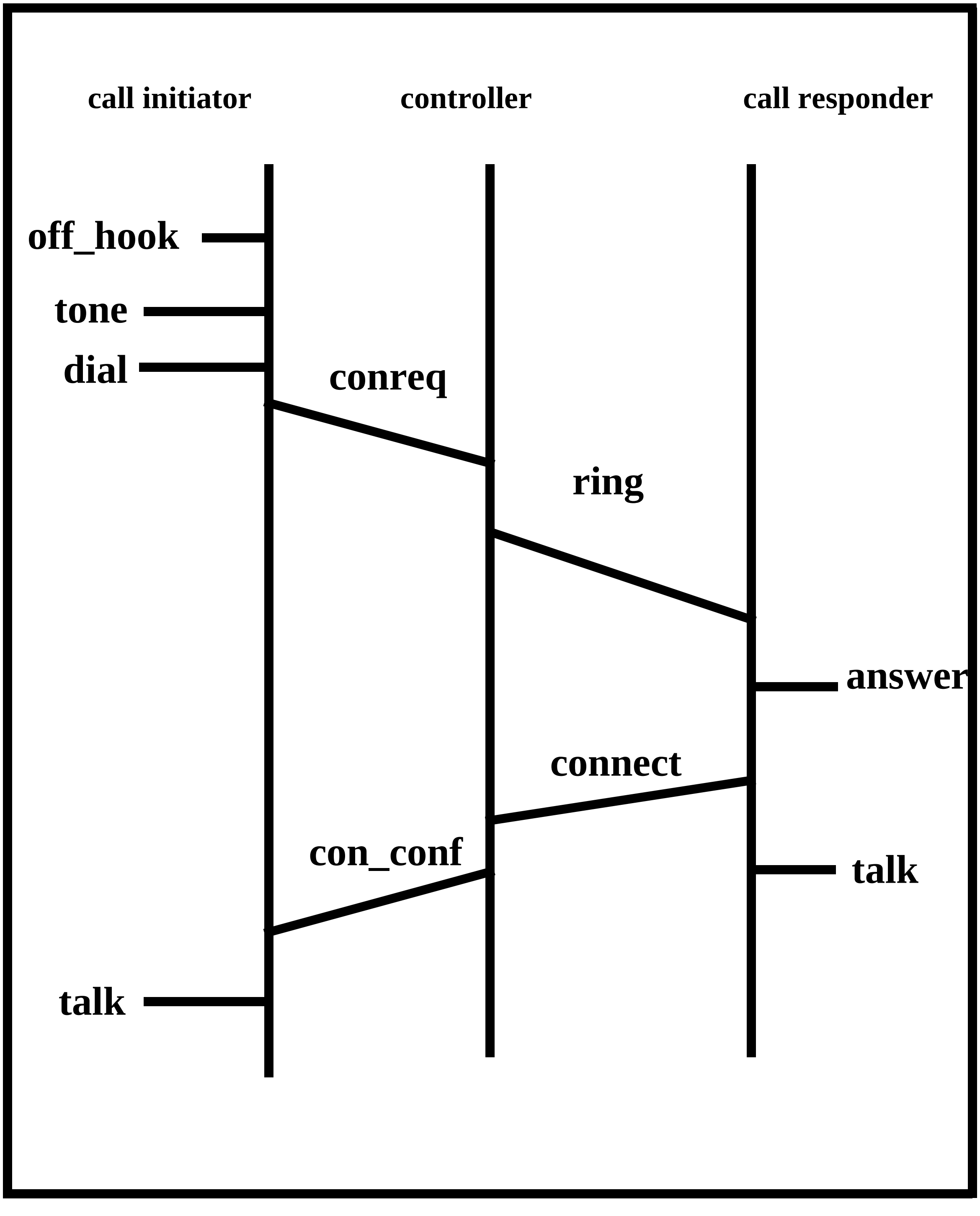
NOTE: the design error, which is hidden in the parallel composition, shows up explicitly in the expansion.

An expanded specification such as this one is said to be in ’monolithic’ style, while a specificaton such as the one presented earlier, showing a set of communicating components, is said to be in ’resource-oriented’ style.

Resource-oriented specs have architectural meaning, while monolithic ones emphasize action sequences.

MESSAGE SEQUENCE CHART

diagram showing independent actions and synchronizations



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.

(a major consequence is the possibility of the* constraint-oriented style *in LOTOS)

SPECIFYING PARTIAL ORDERING BETWEEN EVENTS

a > d   and b > d and c > d   (> : precedes)

(NOTE: NO ORDER SPECIFIED BETWEEN a, b, c)

WE CAN SPECIFY THIS BY:

```
     a;  (b;c;d;stop [] c;b;d;stop)
  [] b;  (a;c;d;stop [] c;a;d;stop)
  [] c;  (a;b;d;stop [] b;a;d;stop)
```

OR, EQUIVALENTLY, BY THREE PARALLEL PROCESSES SYNCHRONIZING ON d

```
( a; d; stop ) | [d] | ( b; d; stop ) | [d] | (c; d; stop )
```

(PROCESSES:
<u>INTERLEAVE</u>   W.R.T. a, b, c
<u>SYNCHRONIZE</u>   W.R.T. d   )

e.g.   a d b   impossible   WHY?

NOTE THE MULTI-WAY SYNCHRONIZATION

Note that identical actions in parallel processes which are not in the synchronization set interleave:

```
    a; b; c; stop |[b]| a; b; d; stop
```

expansion:

```
    a; a; b; (c; d; stop [] d; c; stop)
   []
    a; a; b; (c; d; stop [] d; c; stop)
```

or

```
    a; a; b; (c; d; stop [] d; c; stop)
```

SIMPLIFIED GENERAL PARALLELISM INFERENCE RULES

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

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

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

i cannot be included in the synch set, so any process can execute i independently.

Note that synchro. is binary, but after each binary synchronization the action is still available for further synchronization up to a possible *hide (see later)*.

In other words, although the env't sees the system as offering all synchronizing actions together, the inference rules derive them in pairs, e.g.

(a; d; stop) |[d]| ((b; d; stop) |[d]| (c; d; stop))

After a, b, and c are executed independently the second and third d synchronize and the resulting d is propagated and synchronizes with the first d. If the env't also offers d, the action (a single d) occurs.

DISABLE: "[>"

B_1 [> B_2

Models interruption

example:

```
off_hook ;
        (
         tone  ;
         dial  ;
         stop
             [>   hang_up  ;   stop
         )
```

The user can hang up the phone anywhere after the off_hook action and once hung up the user can no longer execute the normal sequence of actions.

Expansion of the above behavior expression:

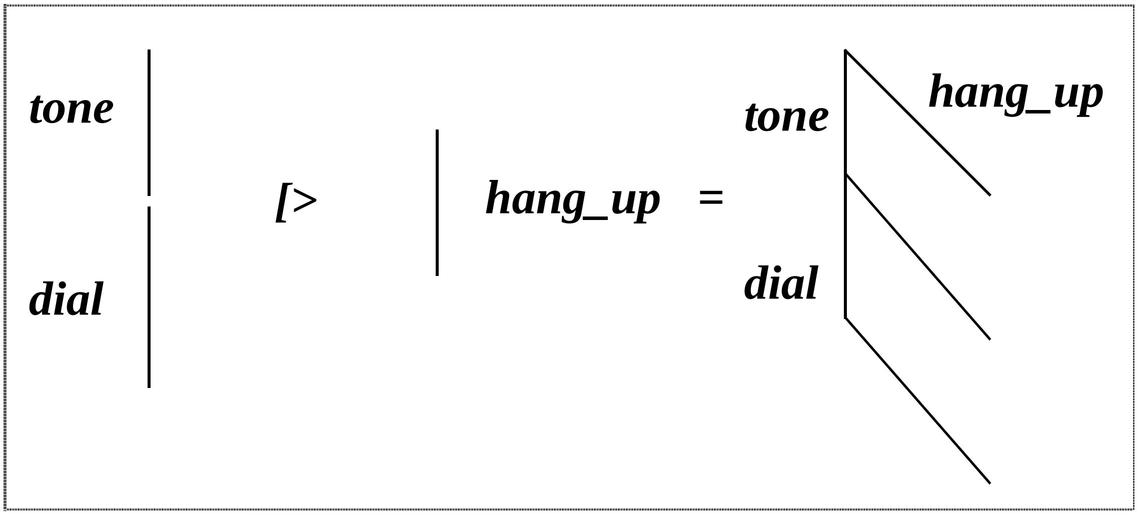
```
off_hook  ;
       ( tone  ;
             (dial  ;   hang_up   ;   stop
               []
              hang_up  ;   stop
             )
         []
        hang_up  ;   stop
       )
```

Behavior trees

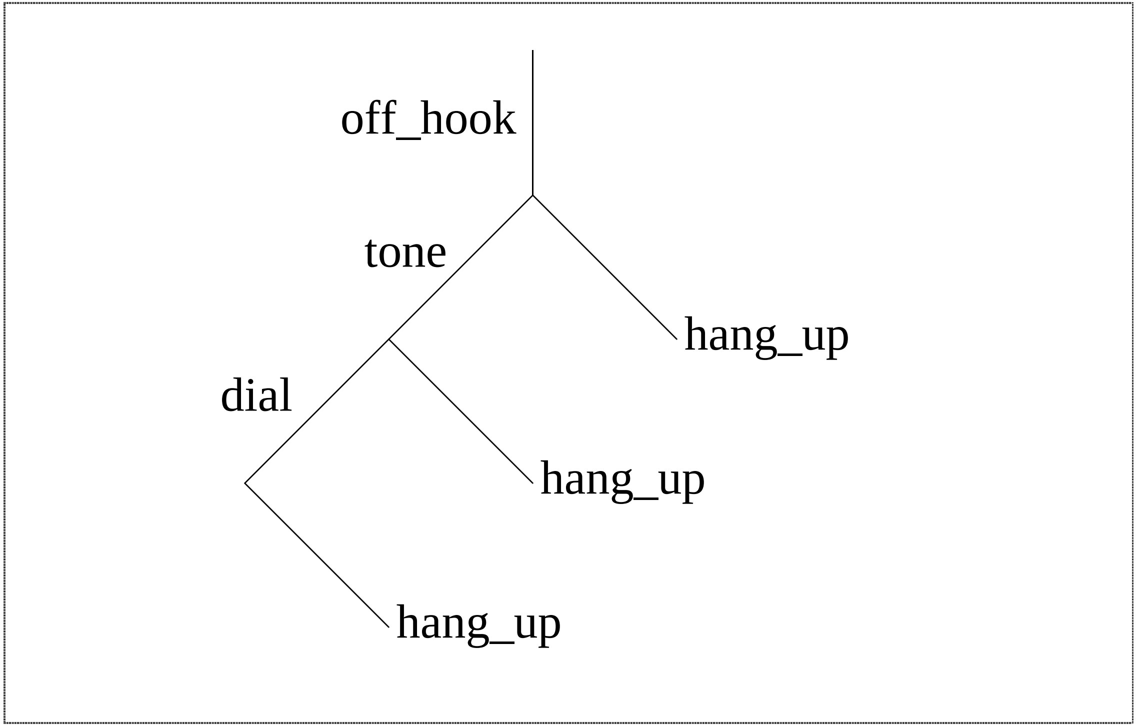
In general, disabling an expression is equivalent to attaching the subtree of the disabling expression to every node of the disabled expression:

hang_up

e.g.



So the behavior of the whole expression will be:



DISABLE OPERATOR
SIMPLIFIED INFERENCE RULES

B_1 - a_1-> B'_1
---------------------------------- (no disable, ctn. B_1)
B_1 [> B_2 - a_1-> B'_1 [> B_2

B_2 -a_2 -> B'_2
--------------------------- (disable occurs, go to B_2)
B_1 [> B_2 - a_2-> B'_2

For each action of **B_1** ***there is the alternative to enter in behavior*** **B_2*****. And once entered in behavior*** **B_2** ***only remaining actions of*** **B_2** ***can be performed.***

Relation with exit behavior ignored for now.

Note important differences:

```
a; b; c; . . .
      [> d; . . .
```

Disable is triggered by environment offering action d

```
a; b; c; . . .
      [> i; d; . . .
```

disable is triggered by internal event, thus is nondeterministic and can occur anytime

```
a; b; c; d; stop
      [> b; . . .
```

if after a the environment offers b, the disable may or may not occur.
However, if the first b from the env't does not cause a disable, the second one will.

EXECUTABILITY OF LOTOS

LOTOS is a specification language, not an implementation language.

However executability at the specification stage is useful, because the specification becomes a *prototype* of the system.

By execution of inference rules it is possible to simulate a specification step by step

Simulation enables the designer to see the system functioning before it is implemented.

Design flaws can be detected, and test data can be generated.

Note: it is possible to write in LOTOS specifications that cannot be executed, because evaluation of certain conditions may not terminate (see later). Usually this is avoided.

USING INFERENCE RULES

(the operation of the interpreter)

```
(
  a   ;   b   ;   c   ;   stop
   []
  c   ;   a   ;   b   ;   stop
  )

      |[a]|

  (
    a   ;   c   ;    stop
     [>
          b   ;   c   ;   stop
   )
```

Decomposing the behavior expression:

The first operator encountered is **"|[a]|"**

```
     B1 |[a]| B2
```

where

```
          B1 is :   a   ;   b   ;   c   ;   stop
                     []
                    c   ;   a   ;   b   ;   stop
```

and

```
          B2 is :    a   ;   c  ;   stop
                     [>
                         b   ;   c   ;   stop
```

We have to apply inference rules on each expression* B1 *and* B2 *separately:

B1 ***is an expression of the form*** **B11 [] B12**

where **B11 is "a ; b ; c ; stop"**
and **B12 is "c ; a ; b ; stop"**

B11 ***and*** **B12** ***are action prefix expressions.***

inferring **B11** ***will produce action*** **"a".**
inferring **B12** ***will produce action*** **"c"**

We perform the same operation on **B2*:***

B2 ***is a disable expression of the form*** **B21 [> B22**

where **B21 is "a ; c ; stop"**
and **B22 is "b ; c ; stop"**

B21 ***and*** **B22** ***are action prefix expressions.***

inferring **B21** ***will produce action*** **"a".**
inferring **B22** ***will produce action "b"***

Applying inference rules is a recursive process:

The inference rules require to go down the syntactic tree of the behavior expression looking for actions, i.e. inference axioms.

Once found, inference axioms make it possible to derive actions, which then can be used by the inference rules.

It is then possible to perform the chain of recursive returns for the inference rules.

***We have reached the lowest level of inference, we may return to the higher level where we had a parallel operator* | [a] | .**

Applying this operator on the resulting actions derived above will give us three possible actions:

***action "*a*" result of the synchronization of action "*a*" that occured in* B1 *and action "*a*" that occured in* B2.**

an action "*c*" coming from* B1 *that is the result of interleaving.

an action "b" coming from* B2 *that is the result of disabling

TRANSITION TO THE NEXT BEHAVIOR

If we choose action "a" *the next behavior expression to infer on will be:*

```
 (
   b   ;   c   ;   stop
  )

       |[a]|

   (
    c  ;  stop
                [>  b  ;   c   ;   stop
    )
```

Explanation:

Choosing to execute action "a" *corresponds to having selected the first action of* **B11** *and* **B21.** *When doing so, this means that we are abandoning the branch corresponding to behavior* **B12.**

The "disable" B22 however remains possible

Choosing action "c" in the initial behavior will result in:

```
a; b; stop

   |[a]|

 (
   a   ;    c    ;    stop
                          [>
                              b   ;    c    ;    stop
 )
```

Some syntax:

Basic Syntax (so far...):

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

Examples:

stop; stop	*all syntactically invalid*
stop; a; stop	*because act. pref. joins*
stop; a	*an action and a behav. expr.*
a; stop	*OK*
a; b [] c; d	*invalid: [] is between behav. expr.*
a; b ||| c; d	*invalid: ||| is betw. b. exp.*
a; b; stop [] c; d; stop	*OK*
a; b; stop || c; d; stop	*OK*
(a; b; stop [] c; d; exit); b; c; stop	*invalid*

Concepts discussed in Class 2:

- Inference axioms and inference rules
- Operators:
  - action prefix
  - choice
  - parallel composition (multi-way synchro.)
  - disable
- Nondeterminism
- Deadlock
- Executability by inference rules