Dpto. Ingeniería de Sistemas Telemáticos
Universidad Politécnica de Madrid

An Introduction to LOTOS

Arturo Azcorra Saloña
Juan Quemada Vives
Santiago Pavón Gómez

Contents

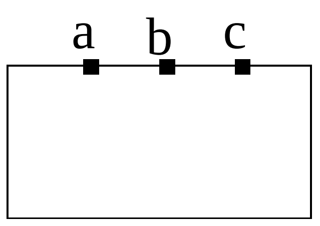
INTRODUCTION

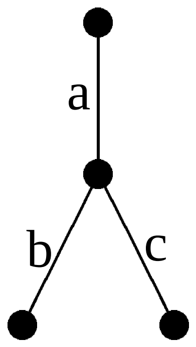
System Specification vs. Implementation

- A specification is a MODEL of a system at a given level of abstraction
- An implementation is the system itself
- Specifying means modeling, i.e., abstracting away from some aspects
- The specification may be tested, validated, used for performance analysis, ...
- The implementation may be obtained by adding to the specification the aspects that were abstracted away

Abstract Modeling and Behavior

- Dynamic description of systems by "event ordering".
- An event is an instance of communication.
- Events are atomic, instantaneous and sequential (never simultaneous).
- A system is specified by defining all the possible event orderings that an external observer may detect.
- Event ordering is structured as a recursive tree.
- Graphical Representation:





Abstract Modeling and Behavior

The LOTOS model of a system is a black box with a number of *gates* that can be seen from its environment.

The first step when specifying a system is the selection of the relevant aspects of the system and decide how can these aspects be mapped to gates. The events that are abstracted to describe a system define the granularity of the behavior of the system.

For example, to model a traffic light we could select three gates (R, G, Y), one for each of the colored lights. These abstraction has been done under the point of view of a typical driver. Notice that we have abstracted away from many aspects, such as the timing mechanism, the power source, etc.

The system behaves by activating its gates. An observable event or action corresponds to the activation of a gate.

The behavior of the system is specified by describing all the possible sequences of events that the system may offer to the environment. Using again the semaphore example, its behavior could be specified as the following sequence of events:

G Y R G Y R G Y R G Y R G Y R ...

Events always occur sequentially. This is, events never occur simultaneously. This restriction is not so important as it may appear in a first moment. In informatics simultaneity is not so frequent (two characters cannot appear simultaneously on the screen, two keys may not be simultaneously pressed, nor can two telephone calls arrive simultaneously to our telephone).

CHAPTER 1: SEQUENTIAL BEHAVIOUR

Input/Output

- An instance of input/output is modeled with an *event* or *action*
- Input/output of data takes place through *gates*
- The system has an interface with a fixed number of gates
- Example of gate identifiers: `Chan_in`, `BusVME`

Input

- Input is modeled as *accepting a value* from the environment at a gate.
- The received value is stored in a *variable*
- The variables is locally *declared* in the value acceptance
- The declaration has a variable *identifier* and a variable *sort*
- Generic input (value acceptance):

```
<gate_name> ? <variable_name> : <sort_name>
```

- Example of input:

```
keyboard_in ? square_side : nat
```

Output

- Output is modeled as *offering a value* to the environment at a gate.
- The offered value is an *expression*
- An expression is formed by *operators* and variables
- Generic output (value offering):

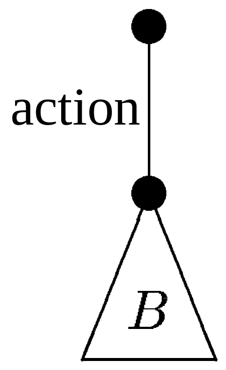
  ```
  <gate_name> ! <expression>
  ```
- Example of output:

  ```
  keyboard_out ! (square_side * square_side)
  ```

Action prefix: “`;`”

- Action prefix denotes sequentiality
- Action prefix composes an event and a behaviour description
- Example:
  `action ; B` means that the system executes `action` and then behaves as `B`

Graphical Representation:



Action prefix: “;”

The sequentiality between events is denoted by operator “;”, called action prefix. This operator composes an action “a” with a behavior expression “B”. The composition is another behavior expression from which it is possible to initially observe event “a”, and afterwards those events belonging to behavior “B”.

The intuitive meaning is that the system will initially accept event “a” behaving afterwards as “B”.

It it important to remember that operator action prefix does not take two behaviors as arguments, as most other operators do. Its arguments are an event denotation and a behavior expression.

Events are usually represented as a line labeled with the event name. Behavior expression are usually represented as a triangle (sequential representation) or as a box (parallel composition).

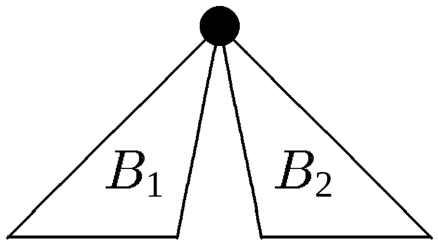
Example of action prefix

- An IP router receives datagrams and forwards them
- The IP router may be forced to segment a datagram
- Its behaviour could be abstracted as:

```
  Net1_in ? in_datagram : ip_dtgrm
; Net2_out ! First_Segment(in_datagram)
; Net2_out ! Second_Segment(in_datagram)
; Net2_out ! Third_Segment(in_datagram)
; . . . . .
```

Choice: “`[]`”

- The system offers to the environment two (or more) alternatives. This is, the system may behave in severl ways.
- Choice composes two alternative behaviour descriptions. The environment will select among both behaviour descriptions.
- Example:
  `<Beh_1> [] <Beh_2>` means that the system behaves as either of both behaviours
- Graphical Representation of choice:



Choice: “[ ]”

Operator choice composes two behavior expressions to form another one that will behave as either of them. The selection between B_1 and B_2 depends on the first event that occurs. Once that an event belonging to B_1 or B_2 occurs, the behavior of the expression will be that of the behavior to which the event belonged. The other behavior is discarded.

Intuitively it may be seen as two two state machines in which their initial states are joined. Thus, once that a transition of the state machines is fired the system will behave as that particular state machine.

The usual way to represent choice graphically is by joining the root of the behavior trees of both behaviors or the top vertex of the triangles representing them.

Notice that using action prefix and choice it is possible to describe the behavior of a system as a tree of events. Operator “;” makes the tree deeper (more levels) and operator [] allows the addition of branches to the nodes of the trees.

Operators “;” and “[]” are the basic operators of LOTOS. The semantics of all the other operators may be represented in terms of an equivalent tree using only “;” and “[]”.

Example of Choice

- An IP router may receive datagrams on net 1 or net 2

```
  (   Net1_in ? in_datagram : ip_dtgrm
    ; Net2_out ! First_Segment(in_datagram)
    ; . . . .  )
[]
  (   Net2_in ? in_datagram : ip_dtgrm
    ; Net1_out ! First_Segment(in_datagram)
    ; . . . .  )
```

- Initially, events `Net1_in...` and `Net2_in...` are offered to the environment.
- Once that the environment selects one, it is only offered the next sequential event.

Guard: “`[<exp1> = <exp2>] ->`”

- A guard is a predicate over values that prefixes a behaviour
- The typical use is to select internally between actions in a choice
- Example of a Teller Machine:

```
  (    [ cash_in_account = true ] ->
     ; Money_dispenser ! requested_money
     ; . . . .  )
[]
  (    [ cash_in_account = false ] ->
     ; Teller_Machine_screen ! no_money_text_message
     ; . . . .  )
```

Example of Guard

- An IP router connected to three subnetworks

```
    Net1_in ? datagram : ip_dtgrm
 ; (    (  [ route(datagram) = 2 ] ->
         ; Net2_out ! datagram
         ; . . . .  )
     []
        (  [ route(datagram) = 3 ] ->
         ; Net3_out ! datagram
         ; . . . .  )
    )
```

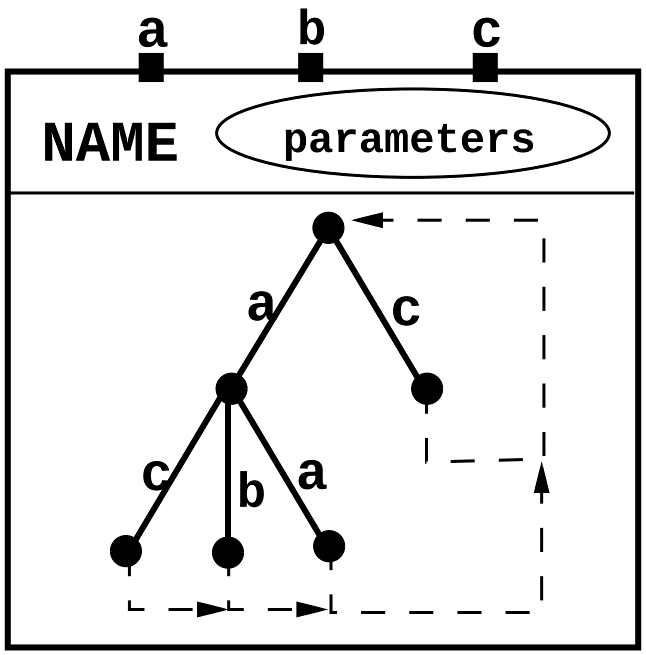
- Depending on the value of `route(datagram)` the datagram will be routed to network 2 or network 3.

Processes

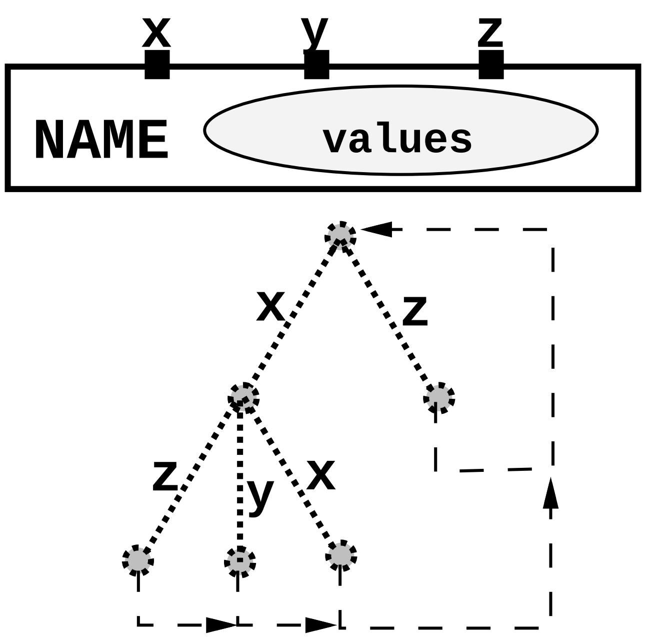
- A process instantiation is an executing instance of a *process definition*
- A process definition is a description of the behaviour of a subsystem
- The process construction in LOTOS serves three main purposes.
  - Representation of recursivity (loops and infinite behaviors).
  - Create behavior abstractions and hierarchies (top-down).
  - Gate relabeling.

Process Definition and Instantiation (I)

PROCESS DEFINITION



PROCESS INSTANTIATION



Process Definition and Instantiation (II)

Process Definition:

```
PROCESS
<proc_name> [ <formal_gates> ] ( <params> ) : <funct> :=
   <behaviour>
WHERE
   <local definitions>
ENDPROC
```

Process Instantiation:

```
<proc_name> [ <actual_gates> ] ( <parameter_values> )
```

Processes

Behavior abstraction is very similar to procedures (more precisely to coroutines) in conventional programming languages. It is possible to define a behavior, assign a name to it and later on perform multiple instantiations of it.

A process instantiation is an executing instance of a *process definition*. A process definition is a description of the behaviour of a subsystem. The process construction in LOTOS serves three main purposes.

- Representation of recursivity (loops and infinite behaviors).
- Gate relabeling.
- Create behavior abstractions and hierarchies (top-down).

Process definition is the only way in LOTOS to specify recursive or iterative behaviors (there are no such statements as `while`, `for` or `goto`). Inasmuch as processes may be instantiated recursively, it is also possible to represent infinite behaviors.

In the process definition it is necessary to declare the (formal) gates through which it will interact with the environment. When the process is instantiated, it is possible to substitute the formal gate list by an actual gate list. This mechanism is called “gate relabeling” and it is very similar to formal parameters and actual parameters in conventional procedures.

The syntax of the process definition consists in a header, a body and a set of local definitions.

```
PROCESS <process_name> [ <formal_gates> ] ( <parameters> ) : <functionality> :=
   <behaviour>
WHERE
   <local definitions>
ENDPROC
```

Header It begins with keyword “`process`” followed by the process identifier, the formal gate list (enclosed in square brackets), the formal parameters (enclosed in parenthesis) and a functionality indication followed by the reserved symbol “`:=`”. The functionality will be seen later and is approximately similar to the returned value of a function.

Body The body of the process definition is the behavior of the process. It may only contain those gates that have been declared in the process definition header. It may contain other process instantiations, provided that the scope rules are preserved. The body of the process ends at keyword “`where`” if there are local definitions, else ending at keyword “`endproc`”.

Local Definitions Local definitions of data types (explained in following sections) and other process are optional. They begin with keyword “`where`” and end at keyword “`endproc`”. The rules for local process definitions are exactly the same as for the definitions of processes at the specification level.

The syntax of a process instantiation consists in the process identifier followed by the actual gate list (enclosed in square brackets) and the actual values for the parameters (enclosed in parenthesis).

```
<process_name> [ <actual_gates> ] ( <parameter_values> )
```

Example of Recursivity (I)

```
PROCESS
IP_ROUTER[Net1_in,Net1_out,Net2_in,Net2_out]:NOEXIT:=
  (   Net1_in ? datagram : ip_dtgrm
    ; Net2_out ! datagram
    ; IP_ROUTER [Net1_in,Net1_out,Net2_in,Net2_out]
  )
[]
  (   Net2_in ? datagram : ip_dtgrm
    ; Net1_out ! datagram
    ; IP_ROUTER [Net1_in,Net1_out,Net2_in,Net2_out]
  )
ENDPROC
```

Example of Recursivity (II)

```
PROCESS
TM [Keypad,Money] (assets,secretNum:nat) : NOEXIT :=
    Keypad ? Number : nat
  ; [ Number = secret_num ] ->
    keypad ? withdraw : nat
  ; (    [ assets ge withdraw = true ] ->
         Money ! withdraw
       ; TM [Keypad,Money] (assets-withdraw,secretNum)
     []
         [ assets ge withdraw = false ] ->
         Money ! assets
       ; TM [Keypad,Money] (0,secret_num)
    )
ENDPROC
```

Recursivity

The example shows a recursive process which instantiates itself in its process definition.

It is possible to define very complex recursive behaviors by cross-instantiating ancestors, brothers, etc.

It is important to remark that LOTOS does not impose a limitation on the number of successive instantiations that can be performed, as is the case with conventional programming languages that may run into stack overflow.

As a general rule, any time that a loop or recursive behavior is needed, it is necessary to define a process.

Process: Example of Gate Relabeling

```
process
IP_ROUTER[Net1_in,Net1_out,Net2_in,Net2_out]:noexit :=
    Net1_in ? datagram : ip_dtgrm
  ; Net2_out ! datagram
  ; IP_ROUTER [Net2_in,Net2_out,Net1_in,Net1_out]
endproc
```

Gate Relabeling

Gate relabeling is equivalent to parameter passing in PASCAL. When the process is defined, a list of formal gates is declared, exactly as the declaration of formal parameters performed in PASCAL. When the process is instantiated, a set of actual gates is provided, exactly as the provision of values for the parameters when calling a PASCAL procedure.

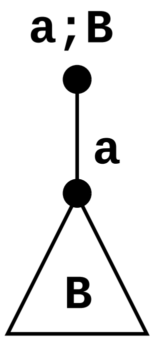
There are no restrictions to the actual gate list provided (e.g. the gate list could be formed by repeating a single actual gate), but of course that they should have been declared within the scope of the process instantiation.

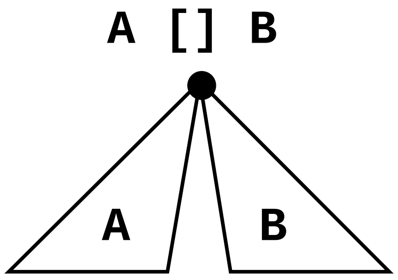
It is very frequent to instantiate a process with different gate lists, specially when they are composed with a parallel operator.

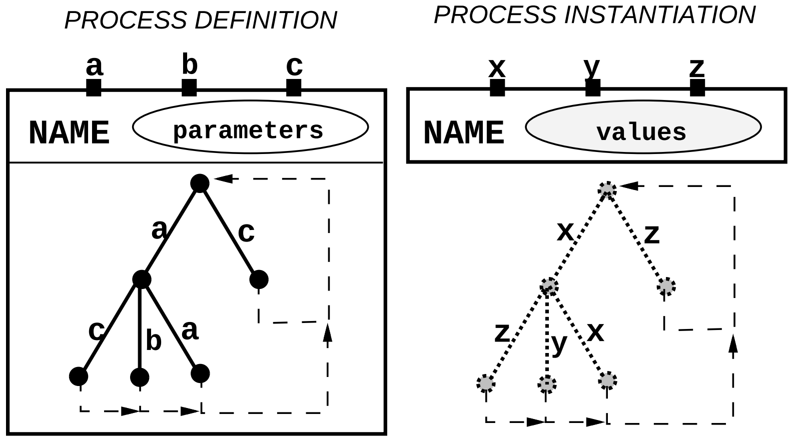
In the example, it may be seen a specification of the IP router in which the actual gate list is formed by swapping the gates corresponding to network 1 and network 2. The effect is that the router will forward from net 1 to net 2 and afterwards from net 2 to net 1 (and so on).

Summary of Basic Operators

- Events and behavior expressions are composed using *operators* (statements).
- Basic operators:
  - Action prefix –“`;`”– models sequentiality.
  - Choice –“`[]`”– models alternative.
  - Guard –“`[<e1> = <e2>] ->`”– models conditions.
  - Process – models recursivity (and also other concepts).



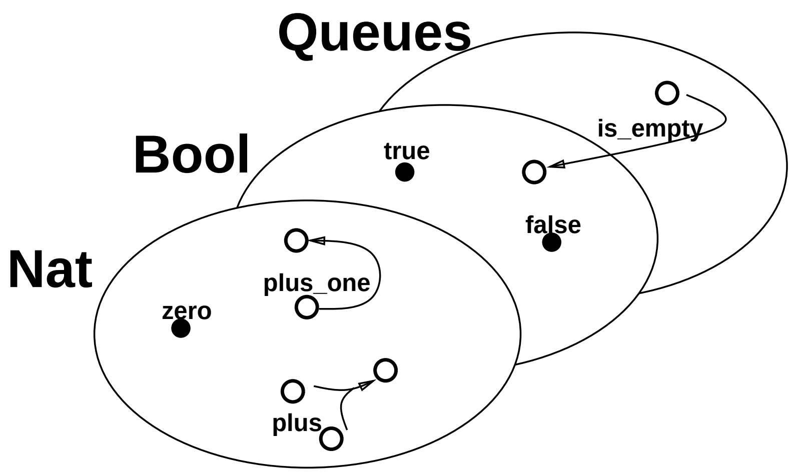




Overview of Data Types

Abstract data types with equational semantics.

- *types*: constructs for the encapsulation of declarations and definitions.
- *sorts*: disjoint sets of values.
- *operations*: declarations of functions (constants are a particular case).
- *equations*: semantics of operations (purely functional, i.e. no memory).



Data Type Library (I)

- It contains the most usual data types and contructions.
- Example of a type:

```
TYPE Boolean IS
  SORTS
    bool
  OPNS
    true, false                 :                -> bool
    not                         : bool           -> bool
    _and_, _or_, _xor_, _iff_ : bool, bool -> bool
    _equal_, _ne_               : bool, bool -> bool
  EQNS
    . . . . . . . . .
ENDTYPE
```

Data Type Library (II)

```
TYPE NaturalNumber IS Boolean
  SORTS
    nat
  OPNS
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9        :           -> nat

    _._, _+_, _-_, _*_, _**_, _/_, _%_ : nat, nat -> nat

    _gt_, _lt_, _ge_, _le_, _eq_, _ne_ : nat, nat -> bool
  EQNS
    . . . . . . . . .
ENDTYPE
```

Examples of Data Type Expressions

```
true                             (* = true  *)
1.0.2                            (* = 1.0.2 *)
not(false)                       (* = true  *)
false and true                   (* = false *)
((1.0) gt (2.0)) or not(false)   (* = true  *)
(1.1) * (1.7)                    (* = 1.8.7 *)
```

Structure of a Specification

```
SPECIFICATION
<name> [ <gates> ] ( <params> ) : <functionality>
  <data types>
BEHAVIOUR
  <Behaviour>
WHERE
  <local definitions>
ENDSPEC
```

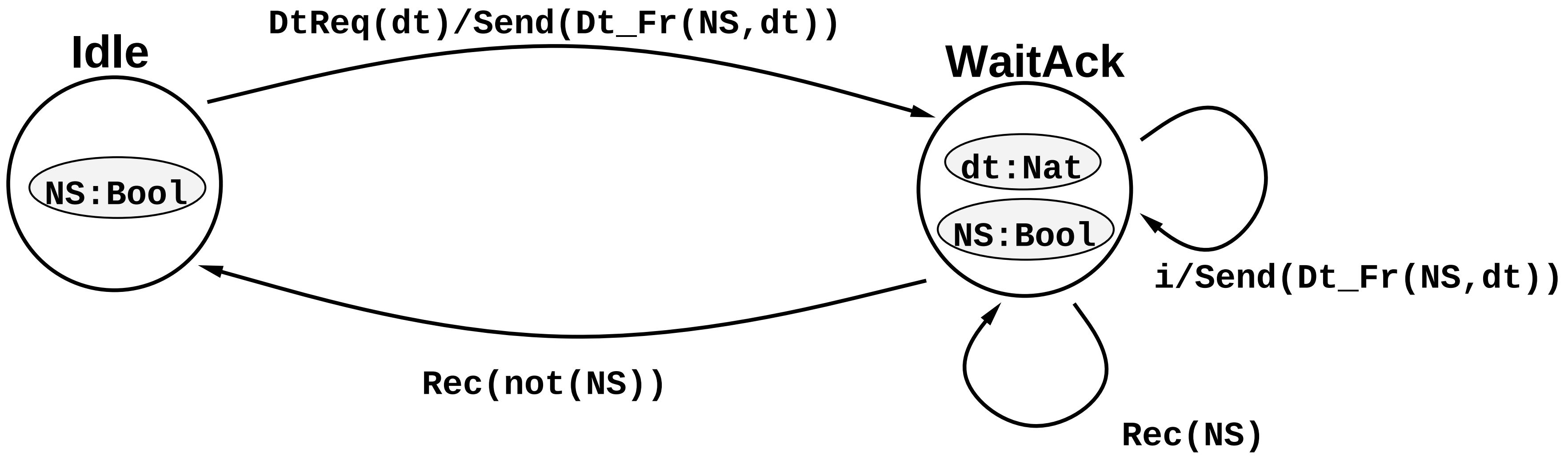
CHAPTER 2: EXTENDED FINITE STATE MACHINES

Extended Finite State Machines (EFSM)

- Finite state machine with auxiliar variables for data.
- It is very frequently used to specify and implement protocols.
- A state machine is defined as:
  - A set of *states*.
  - A set of state *variables*.
  - A set of *inputs* to the automata.
  - A set of *outputs* from the automata.
  - A set of *extended transitions*.
- An extended transition is defined by the initial state, the input that fires the transition, an enabling predicate, the output, a set of actions and the new state.

The State Diagram

- EFSM are usually represented using state diagrams:



Extended Finite State Machine in LOTOS

- One process definition is performed for each state.
- The variables of each state are mapped to parameters in each process.
- Transitions from a state are mapped to a choice of events followed by process instantiations:
  - Each input firing a transition is mapped to an event.
  - Predicates over variables that must hold to fire the transition are mapped to a guard preceding the event.
  - The output associated to a transition is mapped to a sequential event after the firing event.
  - The new state is mapped to the instantiation of the corresponding process.
  - Actualization of variables is mapped to paramaters of the process being instantiated.
- Other constructions may be used if neccessary.

Template of EFSM in LOTOS

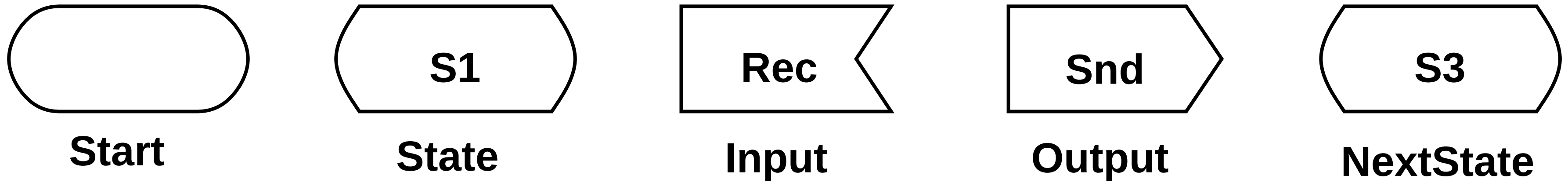
```
process InitialState [<gates>]    (* Initial State *)
             (Var1:T1,..,Varn:Tn) (* context variables *)
:=
                              (* First transition *)
   (   [<Predicate>]->                (* Enabling predicate *)
          <events> ;                  (* Interaction *)
          FinalState1 [<gates>]   (* Final state 1 *)
                 (Expr1,..,Exprn) (* Actualized variables *)

   []  .....                      (* Next transition *)
       .....                          ......
```

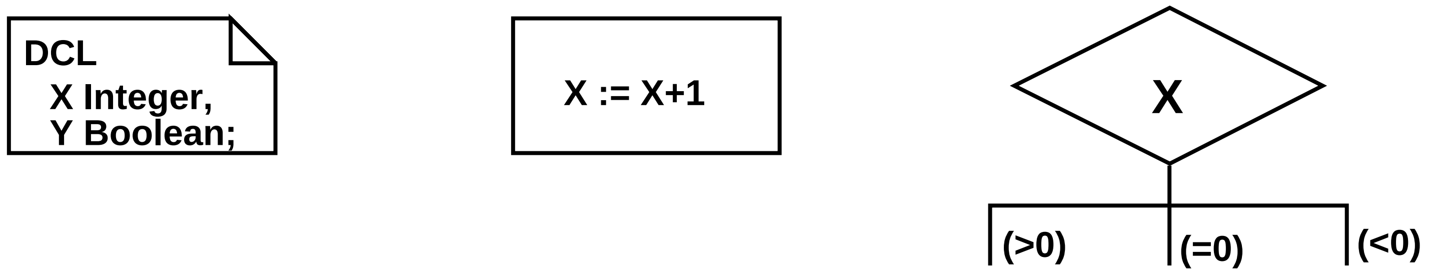
Example of EFSM in LOTOS

```
SPECIFICATION Bit_alt_sender [usr,snd,rcv] : NOEXIT
  LIBRARY Boolean, NaturalNumber ENDLIB
BEHAVIOUR    Idle [usr,snd,rcv](true)    WHERE
process Idle [usr,snd,rcv](NS:bool):noexit:=
   usr ? data:nat ; snd ! Dt_Fr(data,NS)
 ; WaitAck [usr,snd,rcv](data,NS)
endproc
process WaitAck[usr,snd,rcv](dt:nat,NS:bool):noexit:=
     rcv ? NR:bool
   ; (   [NR ne NS] -> Idle[usr,snd,rcv](not(NS))
      [] [NR eq NS] -> Wait_Ack[usr,snd,rcv](dt,NS) )
 []  i ; snd ! Dt_Fr(dt,NS)
   ; WaitAck [usr,snd,rcv](dt,NS)
endproc  ENDSPEC
```

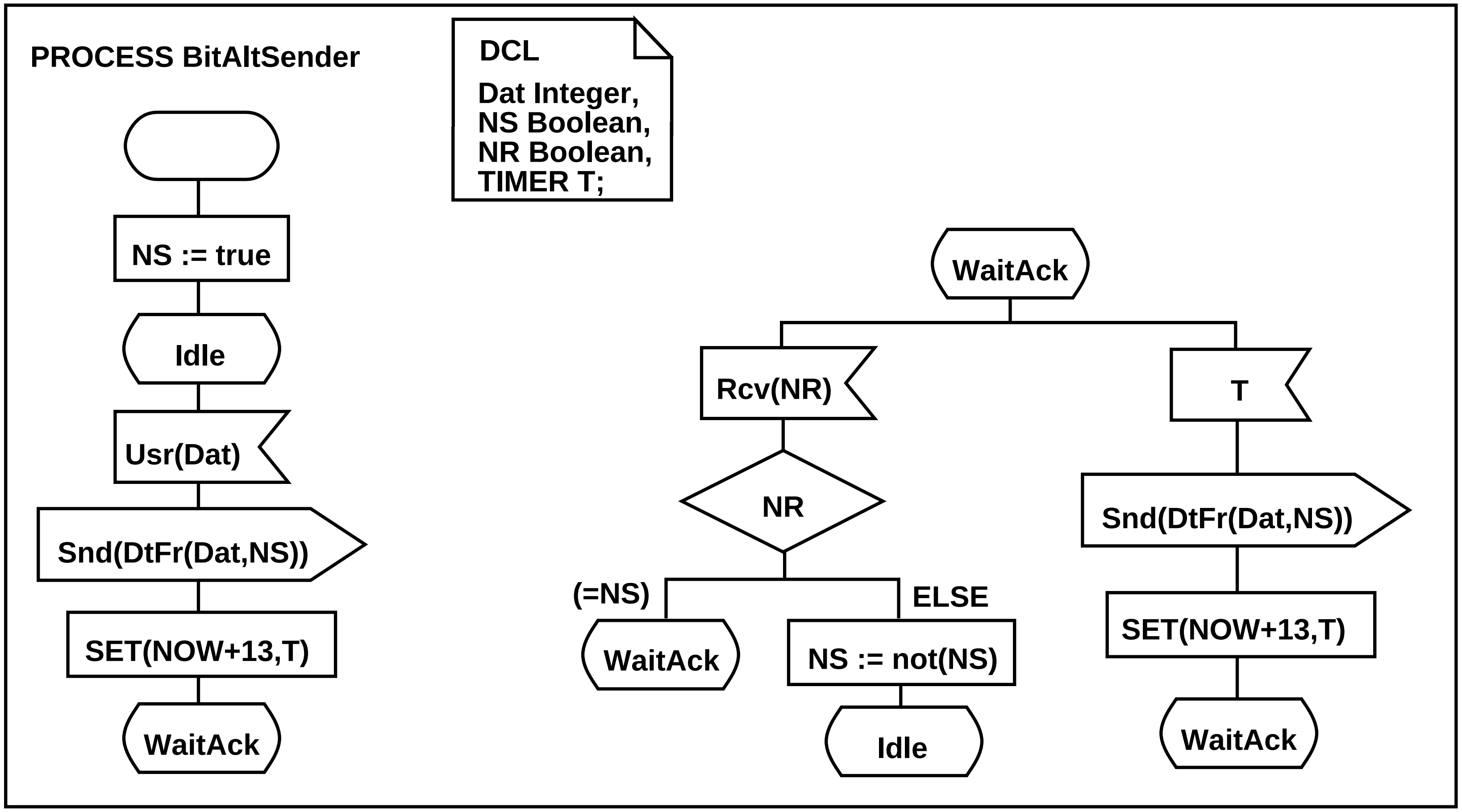
Extended Finite State Machine in SDL

- SDL: Specification and Description Language (Rec. Z.100)
- Graphical Syntax oriented to description of EFSM.





Example of EFSM in SDL

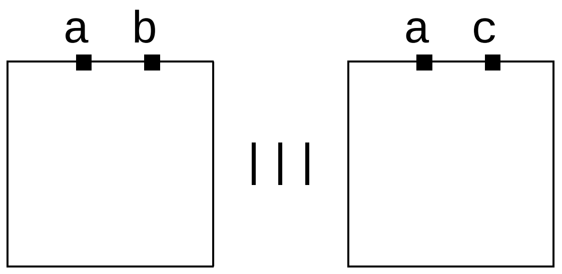


CHAPTER 3: CONCURRENCY

Interleaving: "|||"

- Interleaving is a *parallel* operator.
- Parallel operators model concurrency.
- Interleaving represents concurrent composition without interaction.
- In LOTOS there is no true concurrency, but *interleaved* one.
- The two concurrent subsystems *interleave* their events.

Graphical Representation:



Interleaving: “|||”

When two behaviors are composed in interleaving, both of them evolve in an independent manner. Informally, this means that if one of the behaviors accepts an event and changes it state, the composition will behave in the same way.

If B_1 accepts event a and changes its state to B_1'. Then, if B_1 is interleaved with another system B_2 the resulting behavior will also accept event a and will change its state to another one equivalent to B_1' interleaved with B_2.

The graphical representation is that of two isolated systems with nothing in common: two boxes with their gates. Notice that in this case behaviors are being depicted as boxes instead of triangles (used for trees) because it is usually more clear for illustrating concurrency.

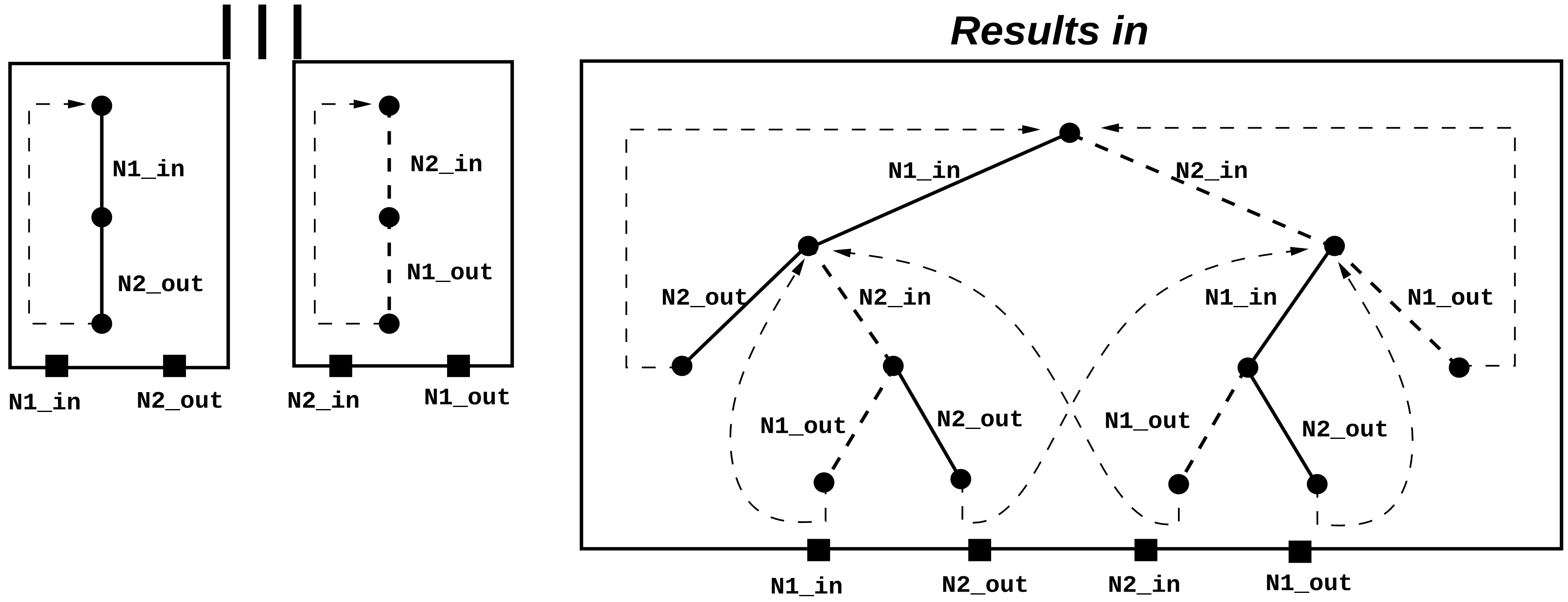
Example of Interleaving (I)

An IP router as multiple forwarding subsystems:

```
process
IP_ROUTER[Net1_in,Net1_out,Net2_in,Net2_out]:noexit :=
   FORWARD[Net1_in,Net2_out]
|||
   FORWARD[Net2_in,Net1_out]
where
 process FORWARD[Net_in,Net_out]: noexit :=
    Net_in ? datagram : ip_dtgrm
  ; Net_out ! datagram
  ; FORWARD[Net_in,Net_out]
 endproc
endproc
```

Example of Interleaving (II)

Equivalent behaviour of the IP router example



Synchronization: Value Passing

- Two processes may communicate by *synchronizing* at given gates.
- Synchronize is equivalent to *share* a common gate between two processes.
- Two events may synchronize only if they occur at a gate in which two behaviours are synchronizing.
- A value offering synchronizes with a value acceptance if they are of the same sort.
- *Value passing*: the value is communicated from the offering process to the accepting process.
- The environment perceives the occurrence of only *one* event.
- It is equivalent to *simultaneously* firing both transitions.
- One such event comes from the *simultaneous* ocurrence of one event at each behaviour.
- For an event to occur at a synchronizing gate, it must occur in both processes.

Partial Synchronization: “`|[ <gates> ]|`”

- Partial Synchronization is another *parallel* operator
- Partial Synchronization represents concurrent composition with interaction.
- The concurrent behaviours synchronize in the gates listed in the operator:
  - Events ocurring at gates in the list will synchronize.
  - Events ocurring at gates not in the list will interleave.

  Interleaving is Partial Synchronization in no gates.

Example:

```
  (keyboard ? num : nat ; line ! num ; ...)
|[ line ]|
  (line ? var : nat ; screen ! var ; ...)
```

Partial Synchronization: “`|[ <gates> ]|`”

Partial synchronization is the general case of the parallel operator. When using partial synchronization it is possible to select the gates at which the two behaviors must synchronize, leaving the remaining gates evolve in interleaving.

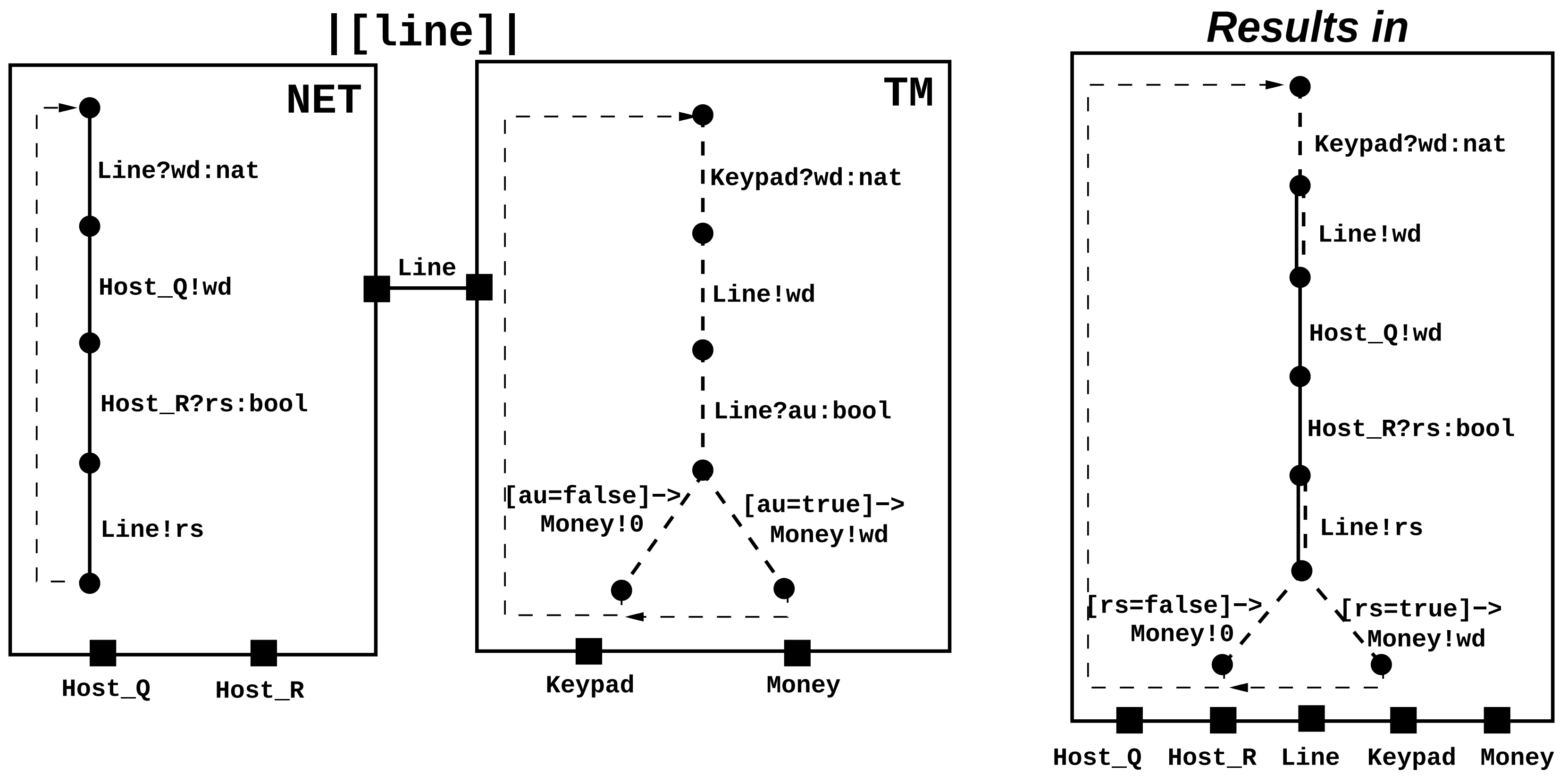
Example: Suppose that we have defined processes `ProcA[a,b]` and `ProcB[a,c]`. Then, we instantiate the processes composing them with partial synchronization:

```
  ProcA[a,b]
|[a]|
  ProcB[a,c]
```

Those events that take place at gate `a` must occur simultaneously in both processes because gate `a` belongs to the gate set of the partial synchronization operator. Those events that take place at gates `b` or `c` may occur independently in `ProcA` or `ProcB` (respectively).

Example of Partial Synchronization (I)

A Teller Machine connected through a network to request authorisation from a Host.



Example of Partial Synchronization (II)

The LOTOS text of the example.

```
process AT_NET[Keypad,Money,Line,Host_Q,Host_R]:noexit:=
  NET[Host_Q,Host_R,Line]
|[ Line ]|
  TM[Keypad,Money,Line]
where
  process NET[Host_Q,Host_R,Line]:noexit:=
     Line ? wd:nat
   ; Host_Q ! wd
   ; Host_R ? rs:bool
   ; Line ! rs
   ; NET[Host_Q,Host_R,Line]
  endproc
```

Example of Partial Synchronization (III)

```
   process TM[Keypad,Money,Line]:noexit:=
      Keypad ? wd:nat
    ; Line ! wd
    ; Line ? au:bool
    ; (    [ au = false ] ->
           Money ! 0
         ; TM[Keypad,Money,Line]
      []
           [ au = true ] ->
           Money ! wd
         ; TM[Keypad,Money,Line]   )
   endproc
endproc
```

Full Synchronization: “||”

- Full Synchronization is another *parallel* operator.
- Full Synchronization represents sharing *all* gates (common or not).
- Full Synchronization means synchronize in *all* vissible events.
- Full Synchronization is Partial Synchronization at *all* gates.

Full Synchronization: “||”

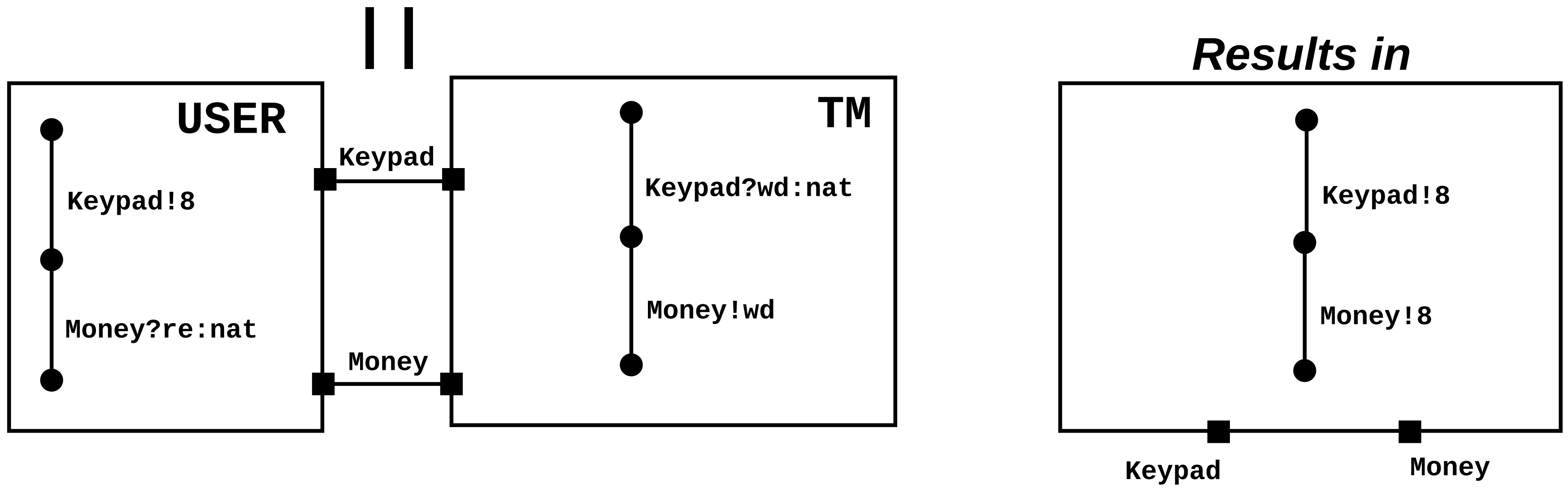
Full synchronization represents total agreement between both behavior in order to accept an event from the environment. If the events offered at visible gates by the behaviors are different they will not occur.

If both behaviors offer the same event, it may occur and both behaviors would then change their state. This is, if B_1 offers event a and changes its state to B_1', and if B_2 offers also event a and changes its state to B_2', then, the composition would offer event a and then would behave as B_1' in full synchronization with B_2'.

Full synchronization is equivalent to partial synchronization where the gate set is the union of gate sets of the parallel behaviors.

Example of Full Synchronization

A Teller Machine that cannot run out of money and a user asking for 8 units.



Summary of Parallel Operators

Syntax:

```
B_1 ||| B_2               (*  interleaving  *)

B_1 |[ <gates> ]| B_2  (*  partial synchronization *)

B_1 || B_2                (*  full synchronization  *)
```

- Interleaving gate: events from `B_1` and `B_2` occur independently.
- Synchronizing gate: events from `B_1` and `B_2` occur simultaneously.

Summary of Parallel Operators

The parallel composition operator is used to represent concurrent behaviors. In the current model of LOTOS there is no real parallelism in the sense that two event cannot occur at the same instant. If there are two behavior expressions in parallel, their events will occur in an interleaved manner, just as processes execute in a single processor machine. Inasmuch as events are assumed to be instantaneous in LOTOS, this restriction in the model does not reduce the expressive power of the language.

The interleaved semantics of the parallel composition allows to obtain the equivalent sequential behavior tree of a concurrent specification.

There are three different syntax for the parallel operator. The partial synchronization operator covers the general case, so the other ones are somehow redundant with it. The syntax of a partial synchronization expression is:

```
A |[G]| B
```

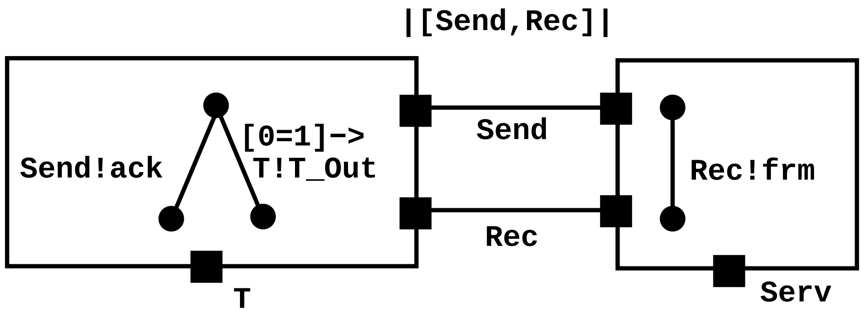
where `A` and `B` are behavior expressions and `G` is a gate set. The gates contained in set `G` are the ones that have to synchronize. The remaining gates of `A` and `B` are interleaved.

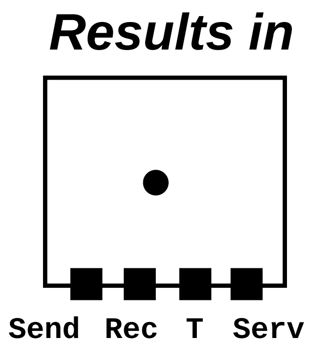
If two behavior expressions `A` and `B` synchronize in a gate `g1`, it is required that every event that occurs at gate `g1` occurs simultaneously in both `A` and `B`. This means that behavior `A` and `B` cannot evolve independently anymore. Informally, this is equivalent to simultaneous transitions in classical state machines,i.e., two transitions that are simultaneously fired by two concurrent state machines.

The interleaving operator `A ||| B` is equivalent to the partial synchronization operator with an empty gate set `G`. The full synchronization operator `A || B` is equivalent to the partial synchronization operator with a gate set `G` that contains both the gates of `A` and `B`, i.e., a parallel composition where all the events that occur must be synchronized.

Deadlock

- A behaviour is said to *deadlock* when no action can be observed from it.
- Deadlock occurs when the two following conditions hold:
  - No interleaving events are offered, and
  - The synchronizing events offered are incompatible.
- Most real systems should be designed deadlock-free.
- Example of deadlock:





Inaction: “`stop`”

- “`stop`” is the representation of a behavior such that no event will ever be observed.
- It can be used to express that a process finishes its execution.
- It can be used to express explicit deadlock situations.

Inaction: “`stop`”

Systems are represented in LOTOS as “behavior expressions” (usually called “behavior” for short). LOTOS provides operators that allow the combination of behaviors in order to build more complex behaviors.

“Stop” is a predefined basic behavior. It describes the system that cannot show any action. Nothing can be observed from “stop”.

It is used to represent deadlock situation or the termination of the activity of the system.

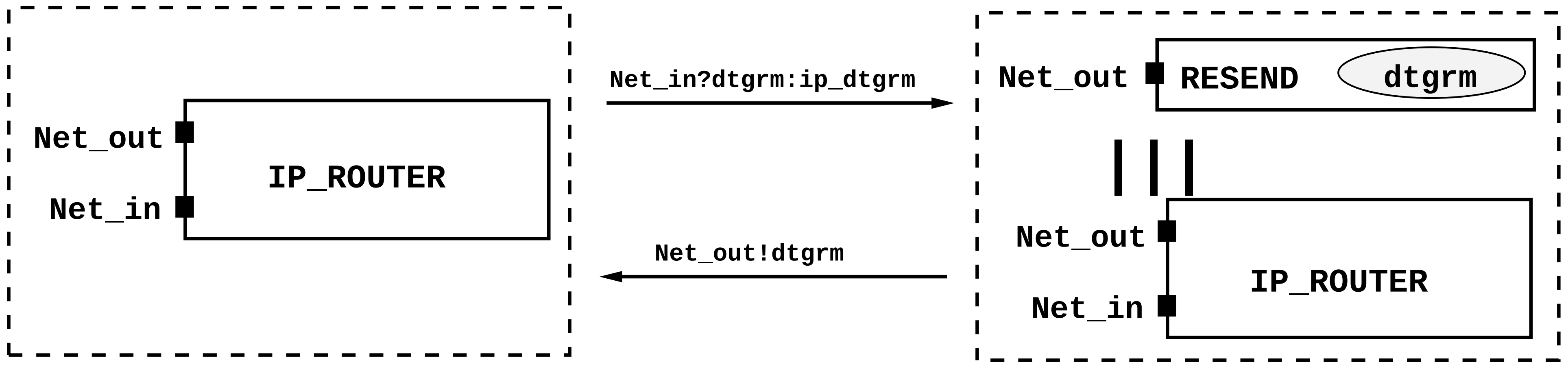
Example of Inaction (I)

An IP router as a process spawning system:

```
process
IP_ROUTER[Net_in,Net_out]:noexit:=
   Net_in ?dtgrm:ip_dtgrm
 ;(  RESEND[Net_out](dtgrm)
   |||
     IP_ROUTER[Net_in,Net_out] )
endproc
process
RESEND[Net_out](dtgrm:ip_dtgrm):noexit:=
   Net_out ! dtgrm
 ; stop
endproc
```

Example of Inaction (II)

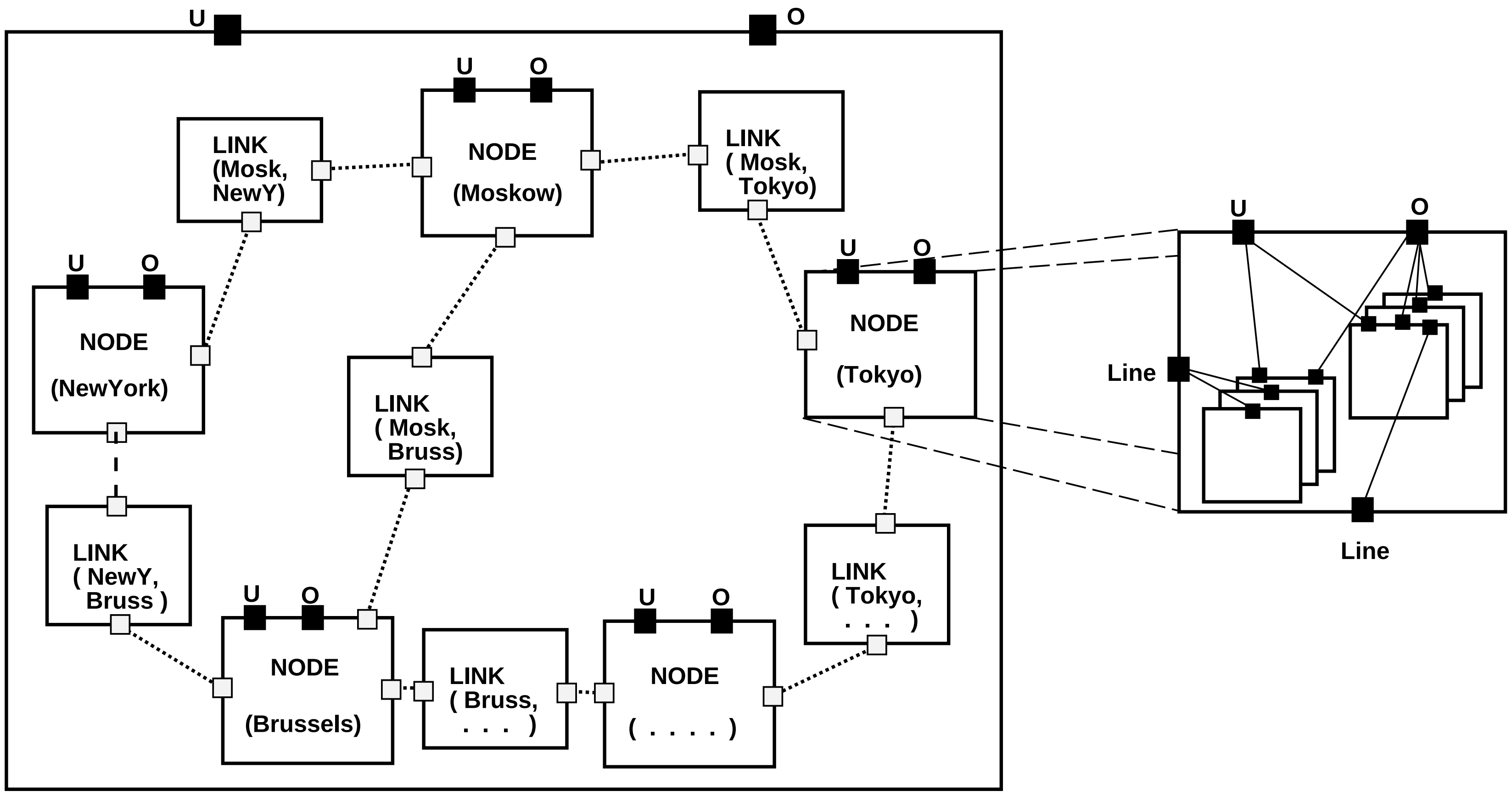
Dynamic creation and destruction of process instantiations:



Resource Oriented Specification (I)

- The system is designed as a collection of interacting *process instantiations*
- A process instantiation is an executing instance of a *process definition*
- It is possible to instantiate several times a single process definition
- A process definition is a description of the behaviour of a subsystem.
- Process instantiations interact by exchanging data through *gates*.

Example of Resource Oriented Specification



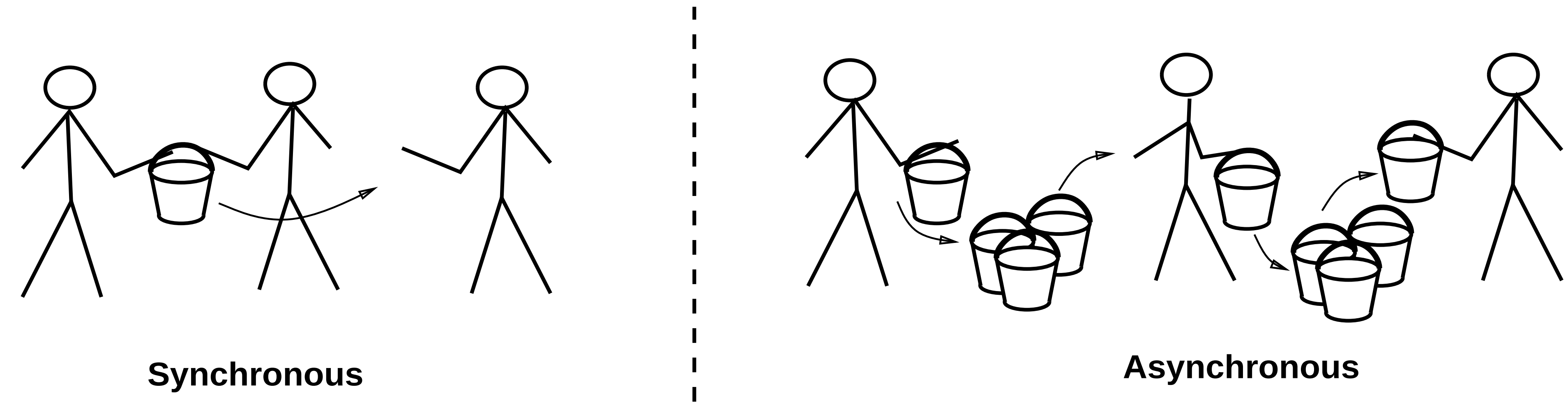
CHAPTER 4: COMMUNICATING EFSMs

Synchronous vs. Asynchronous Communication

- Synchronous communication:
  - Simultaneous transmission and reception.
  - Requires coordination between parties.
  - Example: system call.
- Asynchronous communication:
  - Requires buffering (queues) between parties.
  - Communication identified by location or by process.
  - Example: UNIX pipes.

Synchronous vs. Asynchronous Communication

The fire brigade simile:



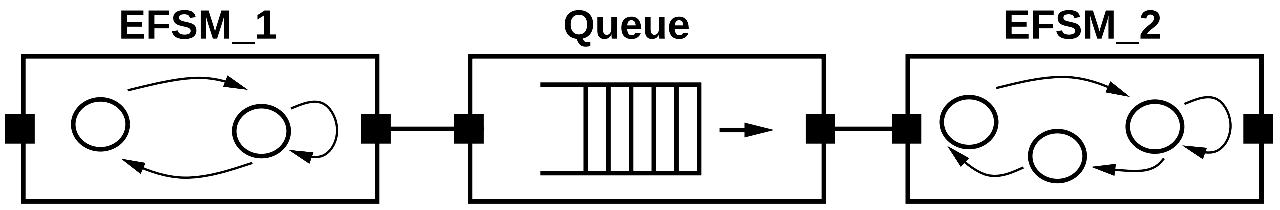
Synchronous Communication in LOTOS

- It is the basic communication mechanism of the language.
- LOTOS supports n-ary symmetric rendez-vous with choice.
- For communicating EFSMs only binary asymmetric rendez-vous required.

Asynchronous Communication in LOTOS

- Asynchronous communication is modeled placing queues between EFSMs.
- The queues are specified as processes.

```
PROCESS Queue [in_q,out_q](queue:string):noexit:=
     in_q ? next:element
   ; Queue [in_q,out_q](lpush(next,queue))
 []  [ queue ne <> ] ->
     out_q ! right(queue)
   ; Queue [in_q,out_q](rpop(queue))
ENDPROC
```



Asynchronous Communication in SDL

- It is the basic communication mechanism of the language.
- SDL supports asynchronous communication identified by agent.
- Every EFSM has an implicit input queue to buffer incomming events.
- Output events are place in the input queue of the destination EFSM.
- SDL 92 has introduced *remote procedure call*: a step to synchronous communication.

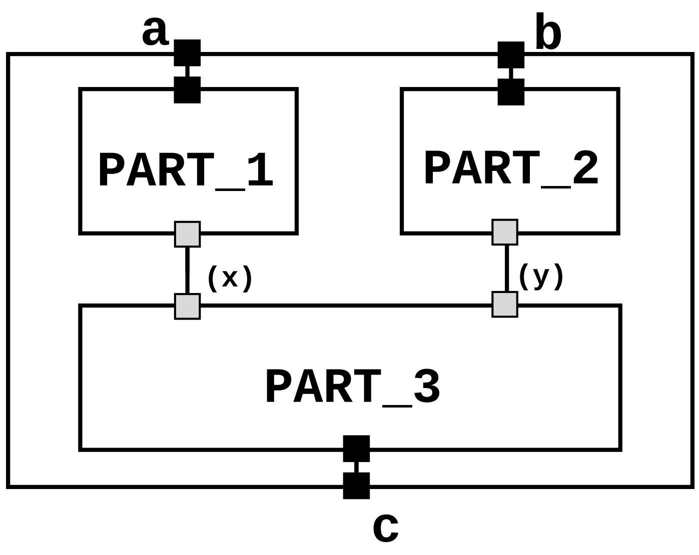
Characteristics of Asynchronous Communication

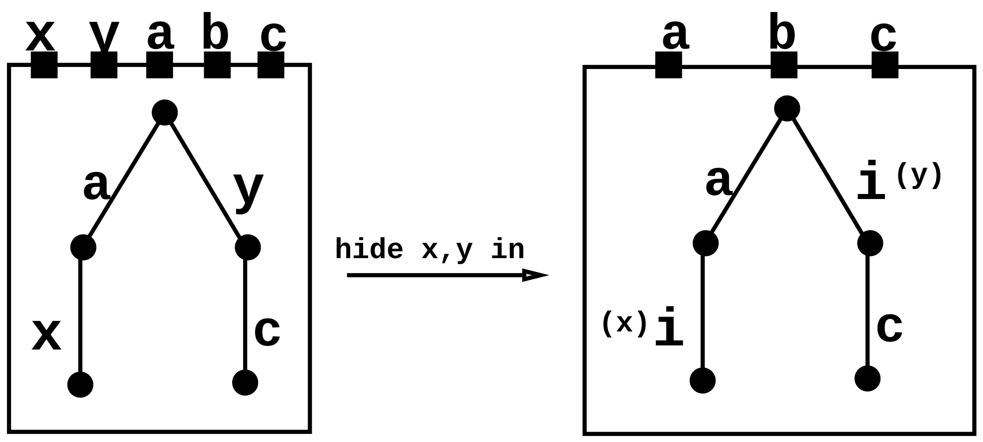
- Finite length queues may lead to deadlock situations.
- Event collision in distributed systems may lead to unconsistent states.
- Unexpected incomming messages must be removed to guarantee.
- It can not be used to model synchronous communication.
- Very frequently used as IPC mechanism in operating systems.

CHAPTER 5: HIDING, INTERNAL EVENT, NON DETERMINISM AND ABSTRACTION

Hiding

- It allows to control the environment’s view of the system.
- It is used to perform stepwise refinement design.
- Its effect is to transform visible events into internal events.
- The hidden gates dissapear from the system’s interface.





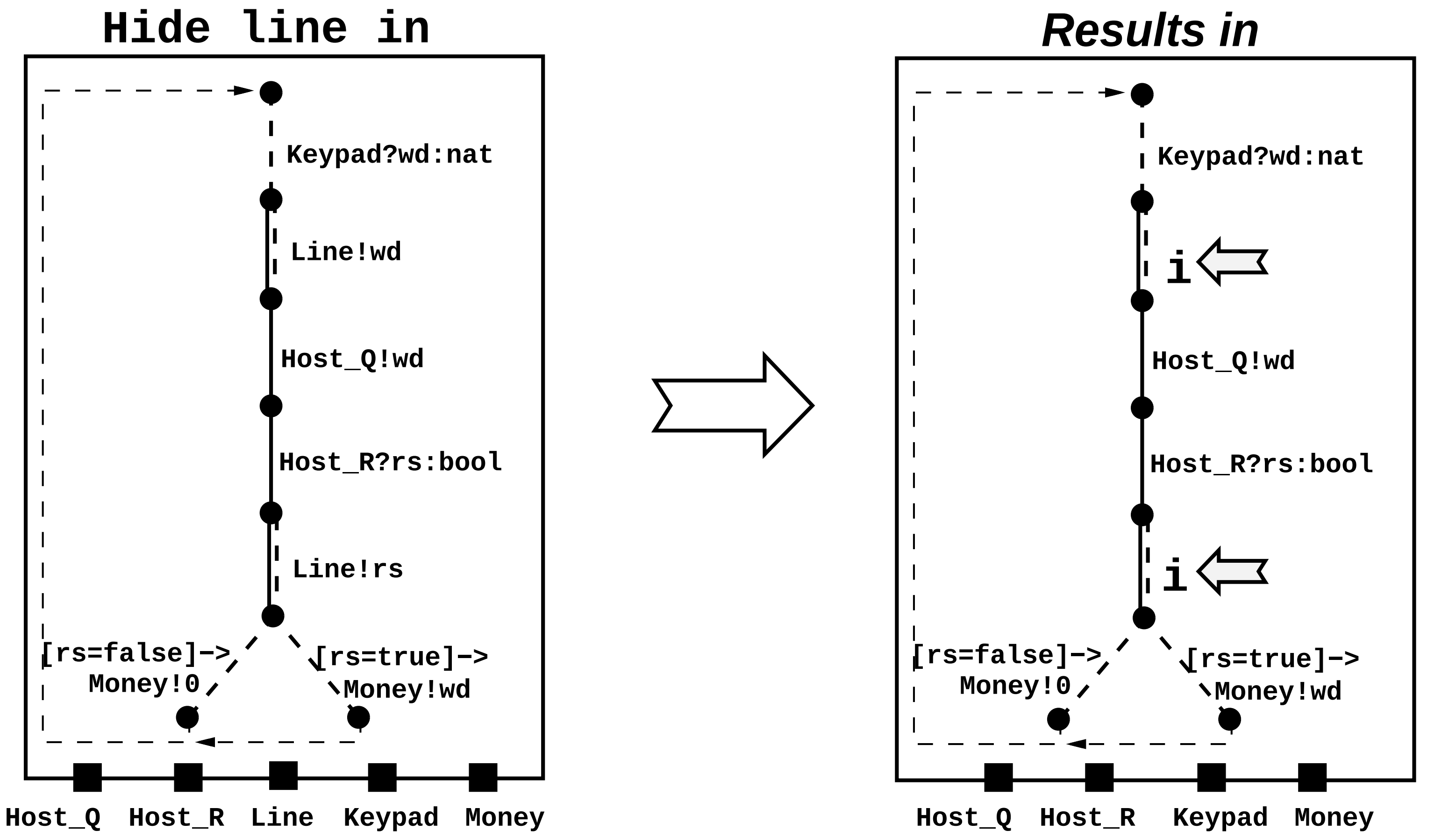
Hiding

Hiding is used to hide communication between subsystems (internal communication) to the environment. It allows to offer a simple interface to the user by hiding internal details.

The effect of hiding is equivalent to substitute the hidden events by internal events, at the global behavior level. It is important to remark that the hiding event is not distributive, i.e., it is not equivalent to hide the events before or after applying other operators.

Hiding is an extremely useful abstraction mechanism. It has the drawback (which is an advantage in some cases) that can introduce non-determinism in the behavior of the system.

Example of Hiding



Internal Event: “`i`”

- The internal event “`i`” is predefined in LOTOS.
- It is an internal transition.
  Consequently, the environment cannot synchronize in it.
- It changes the state of the system.
  Consequently, the environment could notice its effects.
- It appears from hiding a vissible event.
- It is used to express non-determinism.
- It is used to produce a partial specification of a system.

Internal Event: “`i`”

Event “`i`” is an special event that does not take place at a gate. The reason is that events occurring at gates are visible from the environment, while event “`i`” cannot be seen from the environment. It is an internal event, much like expontaneous transitions in state machines.

The syntax for the internal event is a reserved word in LOTOS (i.e. there can be no identifier called “`i`”).

Although the event itself cannot be seen, an external observer is able to notice its effects. Inasmuch as the system experiments a change in its state, the behavior of the system may not be the same before and after the occurrence of the internal event.

For example, let us compose any behavior and an internal event followed by stop with operator choice:

```
   B
[]
   i ; stop
```

An external observer would notice that in some executions the system deadlocks, while in other executions the system works properly. It is clear that despite not being visible, the internal event introduces modifications in the behavior of systems that can be noticed by the environment.

Example of Non-Determinism

An unreliable line may non-deterministically change bits

```
process UNRELIABLE_LINE [Data_in, Data_out] : noexit :=
      Data_in ? bit : bool
    ; (    i
         ; Data_out ! bit
         ; UNRELIABLE_LINE [Data_in, Data_out]
       []
           i
         ; Data_out ! not(bit)
         ; UNRELIABLE_LINE [Data_in, Data_out]
      )
endproc
```

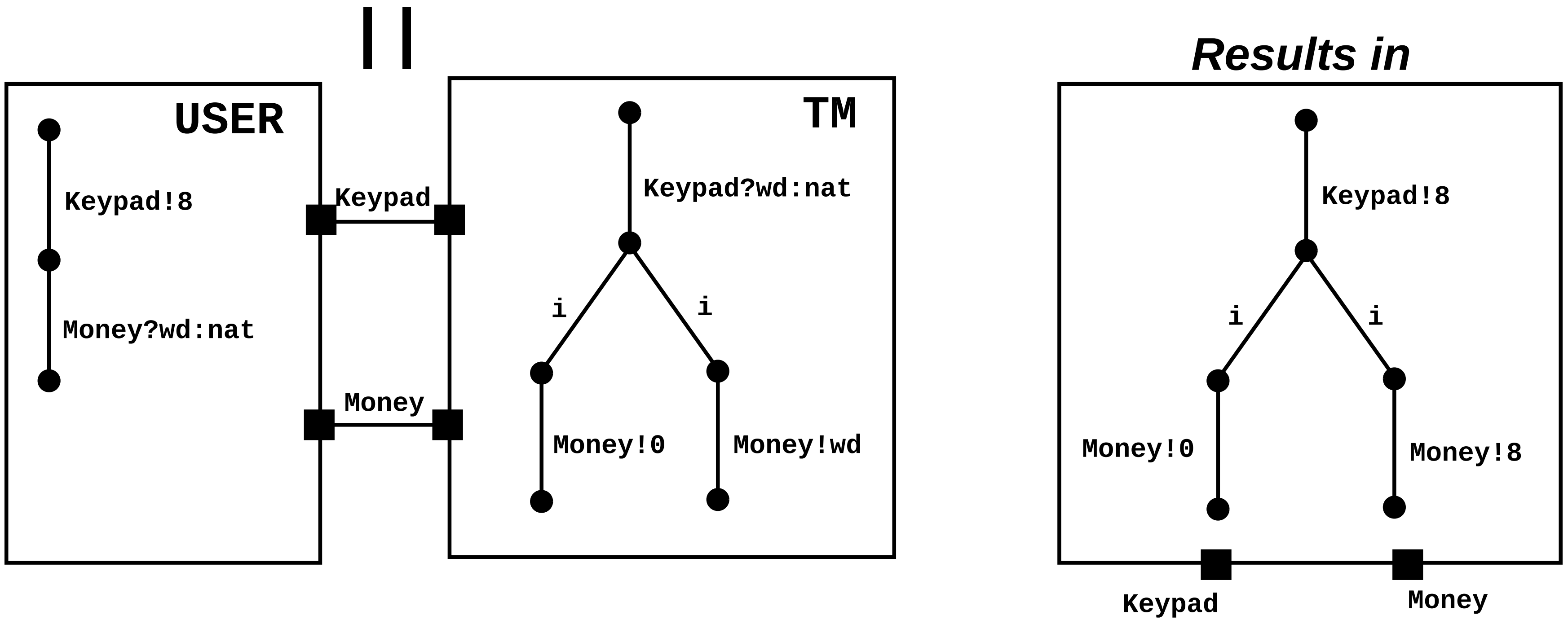
Example of Partial Specification (I)

An IP router *deterministically* discards datagrams in some cases.

```
process IP_ROUTER [Net_in, Net_out] : noexit :=
      Net_in ? datagram : ip_dtgrm
    ; (    i
         ; Net_out ! datagram
         ; IP_ROUTER [Net_in, Net_out]
       []
           i
         ; IP_ROUTER [Net_in, Net_out]
      )
endproc
```

Example of Partial Specification (II)

A Teller Machine that can run out of money and a user asking for 8 units.



CHAPTER 6: ENABLING AND DISABLING

Successful Termination: “`exit(<values>)`”

- `exit` represents the successful termination of a behavior.
- Termination, together with sequential composition, allows the decomposition of a behavior in phases.
- Termination is equivalent to a process instantiation, where the process to be instantiated is determined later on.
- The values placed in the exit statement are passed to the instantiated process.
- *Functionality*: is the list of sorts of values passed in exit

Example:

```
keyboard ? val:nat ; exit(val)
```

Successful Termination: “`exit(<values>)`”

Another basic behavior expression (in addition to `stop`) is successful termination: `exit`.

The informal semantics of successful termination is that the current behavior expression terminates, and the system behaves as the behavior expression in the next phase. Phases are delimited using the sequential composition operator.

The behaviour expression `exit` produces only the event “δ”, that means successful termination.

The semantics of “δ” is somehow similar to the internal event, in the sense that it cannot be seen from the environment, but it produces a change of state in the system.

Enabling: “`>> accept <variables> in`”

- Enabling allows the decomposition of a behavior in sequential phases (Sequential Composition).
- `Beh_1 >> Beh_2` means that behaviour `Beh_2` begins right after `Beh_1` terminates successfully.
- When the first behaviour executes `exit`, the compositiont will behave as the second behaviour.
- The functionality of the first behaviour must match the varible list of the enabling.

Enabling: “`>> accept <variables> in`”

Operator enabling is used to describe a system as a series of sequential phases. The operator composes two behavior expressions to produce another behavior expression. The second behavior expression begins when the first one terminates successfully. Remember that a behavior terminates successfully whenever it executes behavior expression `exit`, i.e., whenever it produces event δ.

The sequential composition `A >> B` is equivalent to replace the occurrences of behavior `exit` in `A` by behavior B.

Notice that enabling is quite different from action prefix: “`;`”. While action prefix composes an event with a behavior, operator enabling composes two behavior expression. The action prefix expression `a ; B` behaves as `B` after the occurrence of `a`. The sequential composition `A >> B` behaves as `B` after the occurrence of `exit` within `A`. In the meantime, the composition behaves as `A`.

The funtionality of the first behaviour must be the same as the varible list of the enabling. This is so beacuse the values provided in the `exit` of the first behaviour will be used as the values for the variables provided in the enabling operator.

The funtionality is declared in process definition and the correct composition of behaviours is checked when performing static semantics analysis.

Example of Enabling

```
   Connect_phase [sap]
 >> accept source:address , dest:address in
   Data_Transfer_phase [sap] (source,dest)
where
 process Connect_phase [sap] : exit(address,address) :=
    sap ? CReq : connreq
  ; sap ! ConnIndication
  ; sap ? CResp:connresp
  ; (   [ Reject(Cresp) = true ] ->
        Connect_phase [sap]
     [] [ Accept(Cresp) = true ] ->
        exit(source(Creq),destination(CReq)) )
 endproc
 . . . . . . . .
```

Disabling

- Allows the representation of interruptions.
- `Behavior_2` destroys (inhibits) `Behavior_1` when it begins.
- `Behavior_2` begins either because the environment accepts one of its initial events or because it contains an initial internal event.

Example:

```
  Data_Transfer_phase [sap] (source,dest)
[>
  Abort [sap]
```

Disabling

This operator is somehow similar to an interruption. The disabling composition `A [> B` behaves as `A` until one of the initial events of `B` occurs. After that, the composition will behave as `B`. Informally, we could say that `B` is interrupting `A`.

The initial events of `B` can only when the environment accepts them (unless they are internal events).

In case that `A` executes an `exit` the composition `A [> B` terminates successfully. This means that after `exit` has occurred, behavior `B` can not interrupt `A` because the disabling composition has terminated successfully.

It is quite frequent to use the following structure in a specification:

```
(A [> B) >> (C [> D) >> ...
```

The above behavior expression is equivalent to:

```
X >> Y >> ...
```

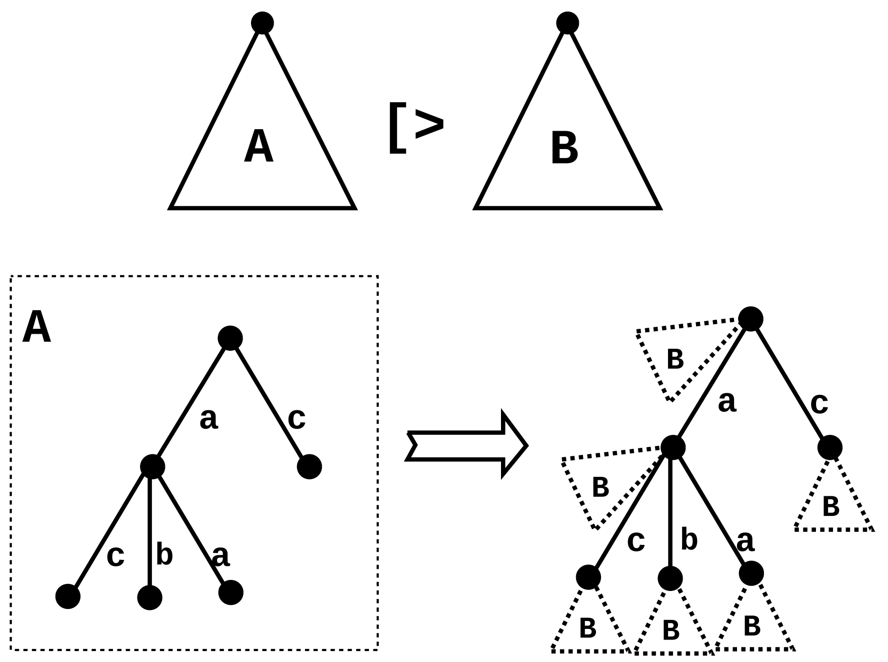
This means that the composition will behave as `X` until it terminates successfully. Behavior `X` is `(A [> B)`. Consequently, it may terminate because `A` terminates successfully or because `B` terminates successfully after having interrupted `A`.

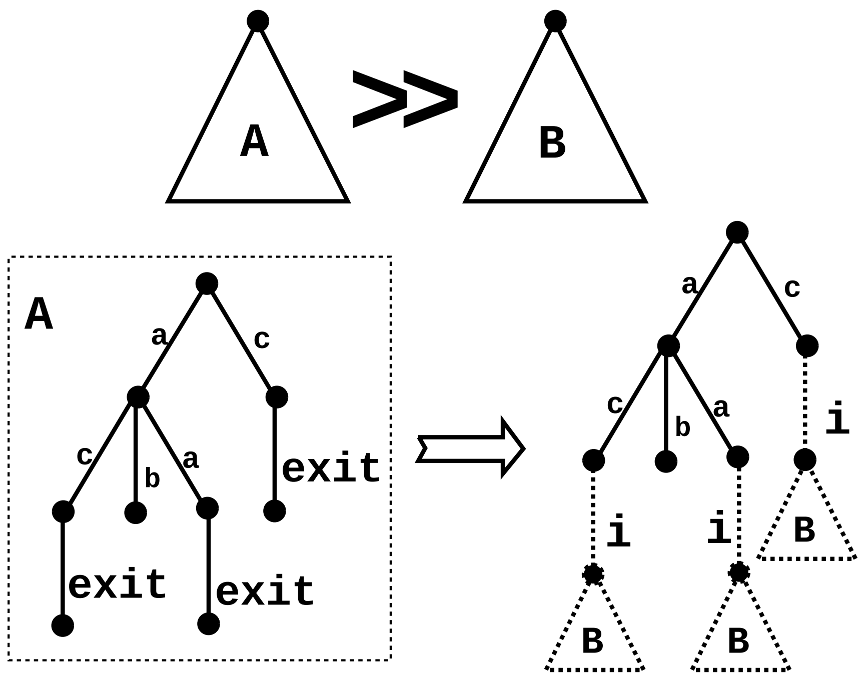
Disabling is equivalent to put the interrupting behavior in alternative at every node of the behavior tree of the interrupted behavior. That is why the initial events of the interrupting behavior may occur at any moment.

Example of Disabling

```
process INIT [start,kill,kbd](pr_id:nat):noexit:=
    start ? proc:proc_class
  ; (   [proc=mail_proc]->
        (     MAIL [kill,kbd] (pr_id)
         |||  INIT [start,kill,kbd](pr_id + 1) )
     [] [proc=other_proc]->
        (     OTHER [kill,kbd] (pr_id)
         |||  INIT [start,kill,kbd](pr_id + 1) )  )
  where
    process MAIL [kill,kbd] (pr_id:nat):noexit:=
         MAILING  [kbd]
     [>  Kill ! pr_id ; stop
    endproc
  . . . . . . . .
```

Summary of Enabling and Disabling





CHAPTER 7: DEFINING DATA TYPES

Writting Data Types

- A Data Type can be defined from scratch:

```
TYPE Convolutional_Code IS
  SORTS FEC_data, FEC_redundancy
  OPNS Encode : FEC_data -> FEC_redundancy ...
  EQNS . . . .
ENDTYPE
```

- A Data Type can be defined using sorts and operations from other types:

```
TYPE Convolutional_Code IS BitString, Boolean
  OPNS Encode : BitString -> BitString
       Check  : BitString, BitString -> bool ...
  EQNS . . . .
ENDTYPE
```

Defining Operations

- Operations are *defined* by writting *equations*.
- An equation is composed by two expressions and an equal sign:

```
  Inc_Mod_2(0) = 1 ;
  Inc_Mod_2(1) = 0 ;
```

- It is required to declare the sort of the equations:

```
EQNS
  OFSORT SeqNum
    Inc_Mod_2(0) = 1 ;
    Inc_Mod_2(1) = 0 ;  . . . .
  OFSORT Bool
    Is_Zero(0) = true ;
    Is_Zero(1) = false ;  . . . .
```

Using Variables in the Equations

- The equations may involve free variables.
- The free variables must have been previously declared.

```
EQNS
  FORALL x:Nat, y,z:Bool
    OFSORT nat
      x + 0 = x ; . . .
    OFSORT bool
      y and false = false ; . . .
    . . . . .
```

Writting Equations

- Equations define the semantics of operations.
- An expression is evaluated by repeatedly applying the appropriate equations.
- The tools apply the equations by *pattern matching*.
- The tools try to match the expression under evaluation to the *left* side of the equation, if it matches, the expression is replaced by the *right* side of the equation.
- For tools, the right side can only contain variables that appear in the left side.
- The equations must be written so the right side is *simpler* than the left side.
- The equations must be written so the result is independent of the order in which they are applied.
- The equations must be written so any expression can be evaluated.

Example on Writting Equations

```
TYPE Frames IS NaturalNumber, BitString
  SORTS Frame
  OPNS
    Mk_Frame       : Nat, Nat, BitString -> Frame
    SeqNum,FrType : Frame -> Nat
    Data           : Frame -> BitString
  EQNS
    FORALL typ,NS:Nat, data:BitString
      OFSORT Nat
        FrType(Mk_Frame(typ,NS,data)) = typ ;
        SeqNum(Mk_Frame(typ,NS,data)) = NS ;
      OFSORT BitString
        Data(Mk_Frame(typ,NS,data)) = data ;
ENDTYPE
```

Conditional Equations as Rewrite Rules

- A conditional equation may have several premises followed by an imply symbol:

```
x ge y = true =>  (x - y) ge 0 = true ;
```

- The tools try to match the expression under evaluation to the *left* side of the equation, if it matches, the premise is tested before performing the substitution.
- To generate a correct rewrite rule it is required that:
  The conditional premises only contain variables that appear in the left side of the equation.

Example of Conditional Equations

Combinations of “x” elements taken “y” by “y”.

```
x ge y = true, y gt 0 = true =>
  Combinat(x,y) = (x/y) * Combinat(x-1,y-1) ;
  Combinat(x,1) = x ;
  Combinat(x,0) = 1 ;
```

Correct Rewrite Systems

- Confluency:
  - The result must be independent of the order of application of the rules.
  - Example on non-confluency: `0 - x = minus(x) ;`
- Termination:
  - The rewrite rules should not form loops.
  - Example on non-termination: `x + y = y + x ;`

Hints for Writing Equations

- Chose the operations (constructors) that will represent basic values.
- Write the equations for each non-constructor operation.
- Rewrite expressions involving the non-constructor to expressions involving constructors.
- Non-constructor operations already defined can also be used in the right side.

Renaming of Types

- A Data Type can be obtained as a copy of an existing one.
- The new type is obtained by *renaming* the sorts and operations of the source type:

```
TYPE BitAltSeqNum IS Boolean renamedby
  SORTNAMES
    SeqNum for Bool
  OPNNAMES
    IncMod2 for Not
    . . . . . . .
ENDTYPE
```

- Renaming is used to make the specification more readable
- Renaming is used to define a new type by modifying the renamed type, without affecting the original type.

Example of Renaming

- Enumerated types are obtained by renaming library enumerated types:

```
TYPE FrameTypes IS Enumerated4 renamedby
  SORTNAMES
    Fr_Type for Enum4
  OPNNAMES
    SABM for Val1
    INFO for Val2
    RR   for Val3
    RNR  for Val4
ENDTYPE
```

- This allows to write the guard:

```
  [ Frame_typ = SABM ] -> ....
```

CHAPTER 8: MORE ON SYNCHRONIZATION

Event Structure

- An event may have more than one *value interaction*
- *Event structure*: ordered list of *sorts* of *value interactions* in an event.
- Most times there is a fixed event structure at gates.
- This is, all events at a gate have the same event structure (*typed gate*).

Examples of valid events and event structures:

Event	Event Structure
`Net ! NetNum ! mssg`	`Nat, datagram`
`Bus ! 0 ! 1 ! 1 ! 0 ! 1`	`Bit, Bit, Bit, Bit, Bit`
`Port ! NetNum ? msg:datagram`	`Nat, datagram`
`Keyb ? Id:Nat ? Ammount:Nat`	`Nat, Nat`

Selection Predicate: “`<event> [<exp1> = <exp2>] ;`”

- A selection predicate is associated to an event.
- A selection predicate imposses conditions on value acceptances.
- The expressions may include variables defined in the event.
- Example:

```
process
ETHERNET_MAC[Net,DL](myaddress:ph_addr):noexit :=
    Net ? add:ph_addr ? hd:fr_head ? dt:data
    [(add eq myaddress) and (length(dt) lt 4.0.0.0)]
  ; DL ! dt
  ; ETHERNET_MAC[Net,DL](myaddress)
endproc
```

Symmetric Rendez-vous

- LOTOS supports, in addition to asymmetric rendez-vous, *symmetric rendez-vous*.
- A value offering is more general than an output.
- A value acceptance is more general than an input.
- A value offering may synchronize with another value offering.
- A value acceptance may synchronize with another value acceptance.

Value Matching

- It is the synchronization of two value offerings.
- Both expressions must be of the same sort.
- The value resulting from each expression must be the same.

Example: Using a value to “split” a gate in an IP router.

```
process
IP_ROUTER[Net](routes:rout_table):noexit :=
    Net ? Net_id:Nat ? datagram : ip_dtgrm
  ; Net ! Next_hop(datagram,routes) ! datagram
  ; IP_ROUTER[Net](routes)
endproc
```

Value Negociation

- It is the synchronization of two value acceptances.
- Both variables must be of the same sort.
- The resulting value of both variables is the same.
- The resulting value is randomly selected from the sort.
- The random selection may be constrained using a selection predicate.

Example of Value Negociation

```
process INIT [start,kill,kbd,born]:noexit:=
    start ? proc:proc_class
  ; born ? new_id : Nat
  ; (   [proc=mail_proc]->
        (     MAIL [kill,kbd] (new_id)
         |||  INIT [start,kill,kbd,born] )
  . . . . . . . .
process ID_MANAGER [born,kill](free_ids:nat_set):noexit:=
     born ? new_proc : nat
     [ (new_proc IsIn free_ids) = true ]
   ; ID_MANAGER [born,die](free_ids Removing new_proc)
[]  . . . . . . . .
```

Summary of Interacion Types

```
process_A [g] |[g]| process_B [g]
```

LOTOS synchronization cases				
process A	process B	synch. condition	interaction type	effect
g !E	g ?x: Sx	$sort(E) = Sx$	value passing	synchro $x = E$
g !E1	g !E2	$E1 = E2$	value matching	synchro
g ?x: Sx	g ?y: Sy	$Sx = Sy$	value negotiation (generation)	synchro $x = y$

Synchronization Conditions

Two events synchronize if:

1. Their event structure is the same.
2. Their matching value offerings have the same value.
3. Their selection predicates hold for the value offerings.

Examples:

```
a ?x:bool !3    <->  a ?x:bool !(2+1) !true    (* NO  *)
a ?x:bool !3    <->  a ?x:bool !(2+2)          (* NO  *)
a ?x:nat  !3    /-\  a !(3+1)  ?y:nat          (* NO  *)
[x le 3 = true] \-/  [y gt 3 = true]
a ?x:bool !3    <->  a ?x:bool !(2+1)          (* YES *)
```

Multi-way Rendez-vous

- LOTOS supports, in addition to binary rendez-vous, full *n-ary rendez-vous*.
- Two synchronizing processes may be synchronized with a third one, and so on:

```
((A[a] || B[a]) || C[a]) || D[a]
```

- This means that *n* processes may synchronize in a given gate.
- For an event to occur at that gate, *all* the synchronizing processes must *agree* on that event.

Constraint Oriented Specification

- The system is designed as a collection of concurrent *process instantiations*.
- There are no internal gates, i.e. there is no structure in terms of subsystems.
- The process intantiations are composed with partial synchronization.
- Process instantiations interact by *constraining* the events observable from the environment.
- Process instantiations only interact at the interface of the system.

CHAPTER 9: MORE ON DATA TYPES

Defining Parameterized Data Types

- Parameterized types are similar to ADA's generics, to OO polymorphism or to the usage of pointers to functions and values in C.
- A parameterized type defines sorts constructed with elements of *formal* sorts.
- It is possible to define operations over the new sorts.
- It can be required that some operations (*formal* operations) exist in the formal sort.
- The semantics of the parameterized type is independent of the one of the formal sort.
- A parameterized type may only be used to produce an *actual* type by *actualizing* it.
- Actualization is performed by providing *actual* sorts, with actual operations, to fill the place of the formal ones.

Example of a Parameterized Type (I)

```
TYPE Parameterized_Stack IS
  FORMALSORTS    fbool, element
  FORMALOPNS
    FFalse :                  -> fbool
    _Feq_  : element, element -> fbool
    _Fand_ : fbool, fbool     -> fbool
  SORTS  stack
  OPNS
    Empty :                  -> Stack
    Push  : Element, Stack   -> Stack
    Top   : Stack            -> Element
    Pop   : Stack            -> Stack
    _eq_  : Stack, Stack     -> fbool
```

Example of a Parameterized Type (II)

```
  EQNS
    FORALL st1,st2:stack, el1,el2:element
    OFSORT Stack
      Pop(Push(el1,st1)) = st1 ;
    OFSORT Element
      Top(Push(el1,st1)) = el1 ;
    OFSORT Fbool
      Push(el1,st1) eq Empty = FFalse ;
      Empty eq Push(el1,st1) = FFalse ;
      Push(el1,st1) eq Push(el2,st2) =
                (el1 Feq el2) Fand (st1 eq st2) ;
ENDTYPE
```

Actualization of Parameterized Data Types

- The actualization of a parameterized type produces an actual type that can be used in the specification.
- A parameterized type is actualized using one (or more actual types).
- The actual sorts and operatiosn of the actual type are used to replace the formal sorts and operations of the parameterized type.

Example of Actualization

```
TYPE Stack_of_nats_par IS Parameterized_Stack
RENAMEDBY
  SORTNAMES Nat_stack for Stack
ENDTYPE
TYPE Stack_of_nats IS Stack_of_nats_par
ACTUALIZEDBY NaturalNumber, Boolean USING
  SORTNAMES
    Bool for Fbool
    Nat  for Element
  OPNNAMES
    False for Ffalse
    and   for Fand
    eq    for Feq
ENDTYPE
```

CHAPTER 10: SCOPE OF IDENTIFIERS

Overloading of Identifiers

- Two objects of different semantic classes may share the same identifier.
- This is, the same identifier can be used for a process, a type, a sort and an operation.
- Operations can have the same identifier if the sorts of their arguments/result are different.
- Variables can have the same identifier in the same scope if they have different sorts.
- To resolve expressions of ambiguous sort use the qualifier: “`of <sort>`”.
  Example:

```
        get ! 0 of nat ; send ! (0+1) of bit ; . . .
```

Scope of Variables

- Specification parameters can be used anywhere whithin the main behaviour definition.
- Process parameters can be used anywhere within the process behaviour.
- The variable from a value acceptance can be used:
  - In the selection predicate of that event.
  - Anywhere in the behaviour expression following, after action prefix, the event.
- The variable from an enabling can be used anywhere in the behaviour expression following the enabling.
- It is impossible to communicate through shared variables.

Scope of Process Definitions

In a proccess definition it is possible to instantiate:

- Any son (not grandson or deeper) process.
- Any ancestor (father, grandfather, and so on) process.
- Any brother process.
- Any brother process of any ancestor process.

Scope of Process Definitions

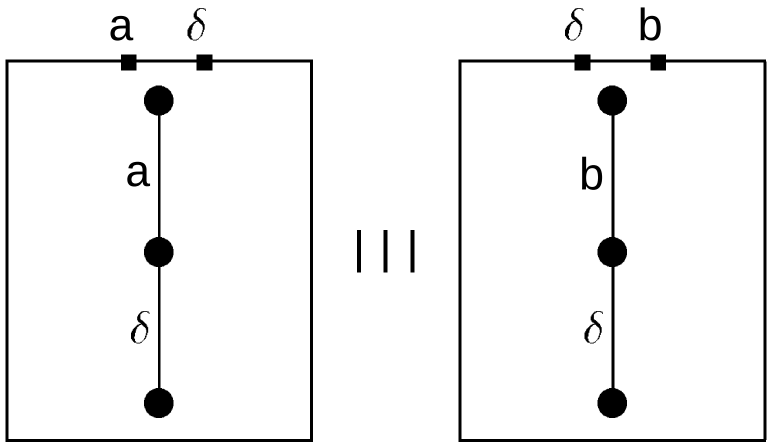
It is possible to define processes within processes. The scope rules for the process definitions in relation to process instantiation are quite similar to that of PASCAL: a process may be instantiated by its “brothers” any direct descendant (son, grandson,...) and also by its “cousins” (i.e. processes directly defined within the brothers of its parent process). The main difference is that PASCAL imposes the additional restriction that a procedure must be defined before it is used, while LOTOS does not impose it.

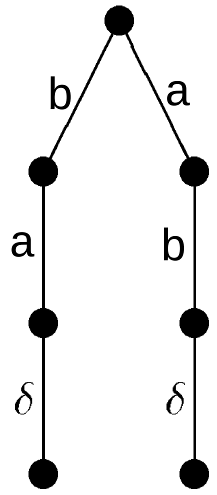
The hierarchy of process definition is a valuable tool to provide a suitable architectural decomposition of the system.

CHAPTER 11: OTHER CONCEPTS

Termination and Parallelism (I)

- Two behaviors composed in parallel must terminate successfully in a synchronous manner.
- Example: `(a ; exit) ||| (b ; exit)`





Termination and Parallelism (I)

Notice that the termination event δ must always occur synchronously (even if the two behaviors are composed with the interleaving operator). This means that if two behaviors are composed in parallel, neither of them can terminate successfully until the other one also terminates successfully.

Termination and Parallelism (II)

- All processes in parallel must have the same functionality:
  - Termination is synchronized.
  - The sorts of the termination must match the ones of the enabling.
- All the values offered in the termination must match.
- It is possible to offer a *neutral* value using "`any <sort>`":

```
(a ; exit(3)) ||| (b ; exit(any nat))
```

Let <var> = <expression> in

- Allows to define new symbolic values.
- Used to avoid repeating long expressions.
- Conceptually equivalent to value asignement.

Example:

```
gate ? x : nat ;
(   let ext_val = 3*((x+3)*x-2)**(x+5),
        int_val = (x*x) + 2              in
       gate ! ext_val ! int_val ; stop
    [] gate ? y:nat   ! int_val ; stop
    [] gate ! ext_val ? y:nat
)
```

Value and Gate Choice

- Value Choice: `choice <var1:sort1,var2:sort2,..> []`
  Composes with `[]` multiple behaviours altering the value of one, or several, variables.
- Gate Choice: `choice <gate_declarations> []`
  Composes with `[]` multiple behaviours altering the value of one, or several, gates.

Examples:

```
 gate ? x : nat ;
 ( choice x:nat,y:bool [] [x le 9]->
     SOME_PROC [gate] (x,y)           )
---------------------------------------------------
 gate ? x : nat ;
 ( choice gx in [chan1,chan2], gy in [k1,k2] []
     SOME_PROC [gx,gy] (9)                        )
```

`par <gateDeclarations> <parallelOp>`

- Used to compose in parallel multiple behaviours altering the value of one, or several, gates.

Example:

```
gate ? x : nat ;
( par gx in [chan1,chan2], gy in [k1,k2] |[gx]|
    SOME_PROC [gx,gy] (9)                        )
```