

Ingénierie du logiciel dans les réseaux informatiques

Chapter 2

The LOTOS Language

LOTOS Basic Principles

Language Of Temporal Ordering Specification

Process Algebra (CCS & CSP) + Abstract Data Types (ACT ONE)

ISO Standard: IS 8807

Based partially on the LOTOS tutorial by Bolognesi & Brinksma [BoB87]

In LOTOS, a distributed, concurrent system is represented by a **process**, possibly consisting of sub-processes.

Process = **black box** able to **interact** with other processes (its environment) and / or to perform internal actions.

Atomic interactions are called **events**

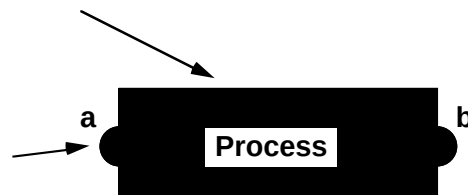
Atomic = instantaneous, one at a time

An event occurs at an **interaction point**, or **gate**

An event may involve the exchange of data

Events imply **process synchronization**

When an event occurs, at least two processes participate in, or experience, that event -> **rendezvous**



- [BoB87] T. Bolognesi and E. Brinksma.
Introduction to the ISO Specification Language LOTOS.
Computer Networks and ISDN Systems 14 (1) 25-59 (1987).

Basic LOTOS - Process Definition

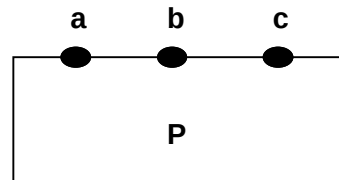
Basic LOTOS is a simplified version of LOTOS without data types.
In Basic LOTOS, an event is just a gate name (or the internal event τ)

Process $\langle \text{process_id} \rangle \langle \text{parameter_part} \rangle :=$

$\langle \text{behaviour_expression} \rangle$

endproc

Ex. : **process** P [a,b,c] := ... **endproc**



process_id : an identifier which designates a process, that is a **process name**

parameter_part : in Basic LOTOS, a list of interaction points or gates

behaviour_expression : an expression defining the behaviour of the process, i.e. the allowed orderings of events

Such expressions are built up from more elementary expressions by using LOTOS operators.

Inaction

A basic behaviour expression : **stop**

```
process inaction [a,b] :=  
    stop  
endproc
```



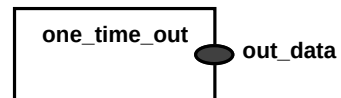
This process cannot perform any event

Action prefix

Prefixing a behaviour expression by an event **g ; B**

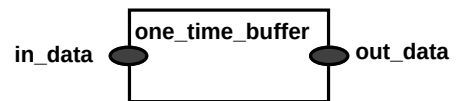
- B : a behaviour expression
- g : an event

```
process one_time_out [out_data] :=  
    out_data ; stop  
endproc
```



The event 'out_data' prefixes the behaviour expression **stop**

```
process one_time_buffer [in_data,out_data] :=  
    in_data ; out_data ; stop  
endproc
```



The event 'in_data' prefixes the behaviour expression (out_data ; **stop**)

Let B_1 and B_2 be two behaviour expressions

=> the behaviour expression $(B_1 \square B_2)$ defines a process that behaves either like B_1 or like B_2 . The choice offered is **resolved by the environment**.

If the environment offers an initial action of B_1 (which is not in B_2), then B_1 is selected.

If the environment offers an initial action of both B_1 and B_2 , then the selection is **non deterministic**.

process simple_duplex_buffer [in_a,in_b,out_a,out_b] :=

in_a; (in_b ; (out_a ; out_b ; **stop**

 [] out_b ; out_a ; **stop**)

 [] out_a; in_b ; out_b ; **stop**)

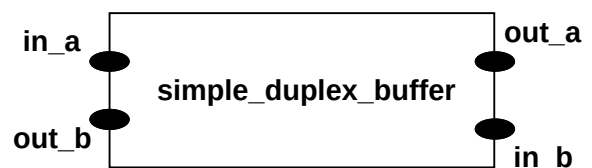
[]

in_b; (in_a; (out_b ; out_a ; **stop**

 [] out_a ; out_b ; **stop**)

 [] out_b ; in_a ; out_a ; **stop**)

endproc



Nondeterminism

$a ; (B_1 \sqcap B_2)$ and $a ; B_1 \sqcap a ; B_2$ model different behaviours

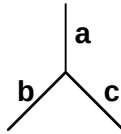
process one [a,b,c]

$a ; (b ; \text{stop}$

$\sqcap$

$c ; \text{stop})$

endproc



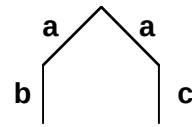
process two [a,b,c]

$a ; b ; \text{stop}$

$\sqcap$

$a ; c ; \text{stop}$

endproc



The choice of 'a' is not determined

Examples of nondeterministic behaviours:

$a ; B_1 \sqcap a ; B_2$

The internal action i is also a source of nondeterminism

$i ; B_1 \sqcap i ; B_2$

$a ; B_1 \sqcap i ; B_2$

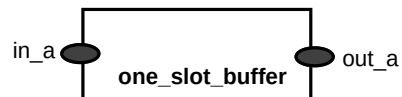
Recursion - Process instantiation

```
process generator [out_data] :=  
    out_data ;  
    generator [out_data]  
endproc
```



Process instantiation: process name + list of actual gates

```
process one_slot_buffer [in_a, out_a] :=  
    in_a ;  
    out_a ;  
    one_slot_buffer [in_a, out_a]  
endproc
```



```
process one_slot_buffer [in_a, out_a] :=  
    in_a; one_slot_buffer [out_a, in_a]  
endproc
```

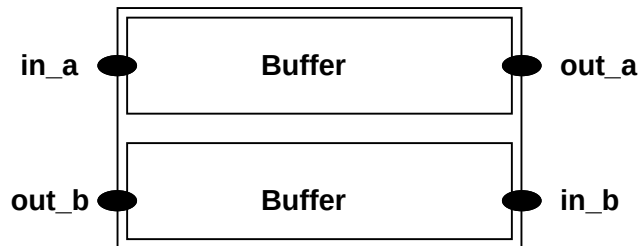
Instantiation with $\neq$ gates

Parallel composition without interaction (pure interleaving) $B_1 \parallel B_2$ **process** duplex_buffer [in_a,in_b,out_a,out_b] :=

buffer [in_a,out_a]

 $\parallel$

buffer [in_b,out_b]

where**process** buffer [in, out] :=in; out; **stop****endproc****endproc**No interaction between B_1 and B_2 is possible

Parallel composition with interaction

$B_1 \parallel [a_1, \dots, a_n] B_2$

process two_buffers [in_a, middle, out_b] :=

 buffer [in_a, middle]

 |[middle]|

 buffer [middle, out_b]

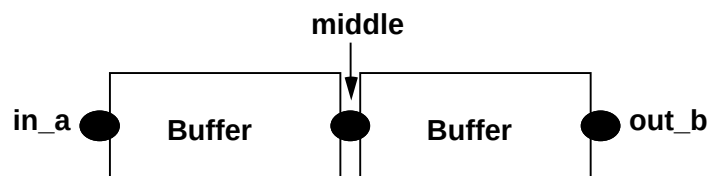
where

process buffer[in, out] :=

 in; out; **stop**

endproc

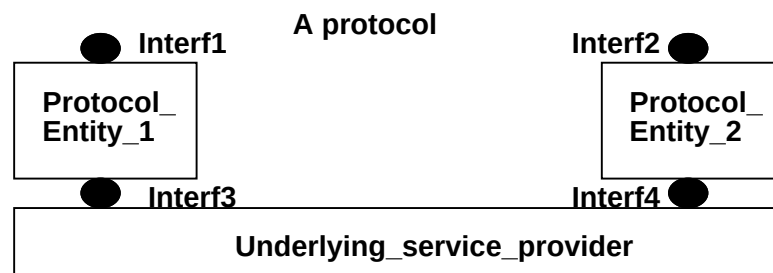
endproc



Every interaction a_i is performed **simultaneously** in both processes B_1 and B_2

Actions not in $a_1, \dots, a_n$ are not synchronized.

Examples of parallel architectures



2 solutions

Process

Protocol [interf1, interf2, interf3, interf4] :=

Protocol_entity_1 [interf1, interf3]

||[Interf3]

Underlying_service_provider [interf3, interf4]

||[Interf4]

Protocol_entity_2 [interf2, interf4]

endproc

Process

Protocol [interf1, interf2, interf3, interf4] :=

(Protocol_entity_1 [interf1, interf3]

|||

Protocol_entity_2 [interf2, interf4]

)

||[interf3, interf4]

Underlying_service_provider [interf3, interf4]

endproc

Multiway-rendezvous

It is easy to describe the synchronization of more than two processes

Example:

$B_1 [g] \parallel B_2 [g] \parallel B_3 [g]$ where $\parallel$ means the synchronization on all gates

Three processes synchronize on g:

g occurs if the three processes are ready to perform g

Hiding

hide $a_1, \dots, a_n$ in **B**

Hiding conceals the observable actions $a_1, \dots, a_n$ present in **B** from the environment.

These actions are thus made **unavailable for synchronization** with other processes.

process two_place_buffer [in_a, out_b] :=

hide middle in

(buffer [in_a, middle]

|[middle]|

buffer [middle, out_b])

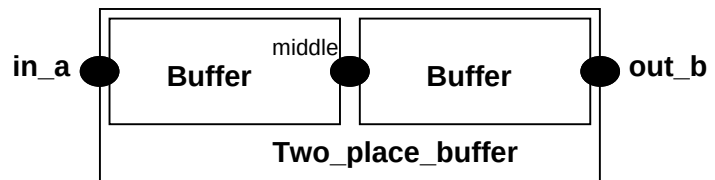
where

process buffer[in, out] :=

in; out; **stop**

endproc

endproc



Equivalent sequential process

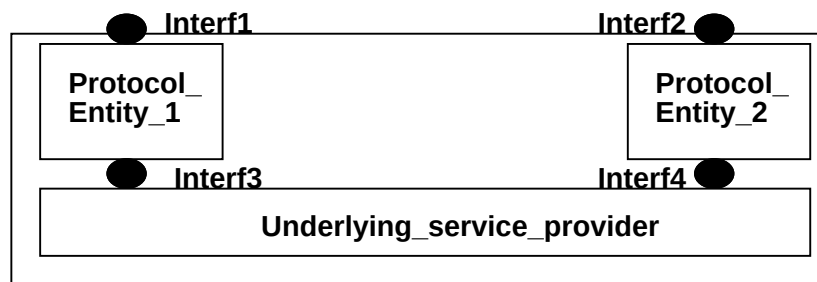
```
process two_place_buffer [in,out] :=  
  in; one_message_in_left [in,out]  
where  
  process one_message_in_left [in,out] :=  
    i; (in; two_messages_in [in,out]  
      []  
      out; two_place_buffer [in,out])  
  where  
    process two_messages_in [in,out] :=  
      out; one_message_in_left [in,out]  
    endproc  
  endproc  
endproc
```

Internal
action:

This is the
hidden
action
'middle'

Use of hiding in architectural design

A service provider



Process Service Provider [interf1, interf2] :=

```

hide interf3, interf4 in
  ((Protocol_entity_1 [interf1, interf3]
    |||
    Protocol_entity_2 [interf2, interf4]
  )
  [[interf3, interf4]]
  Underlying_service_provider [interf3, interf4]
)
endproc

```

This external
box represents
hiding
(or abstraction)

Successful termination of a process**No successful termination**

```
process one_slot_buffer [in_a,out_a] :=  
  in_a ; out_a ;  
  one_slot_buffer [in_a,out_a]  
endproc
```

Unsuccessful termination : stop

```
process one_time_buffer [in_a,out_a] :=  
  in_a ; out_a ; stop  
endproc
```

Successful termination : exit

```
process connection_phase [CR,CC] :=  
  CR ; CC ; exit  
endproc
```


Sequential composition **$B_1 \gg B_2$** B_2 is enabled iff B_1 terminates successfully (**exit** construct)**process** sender [ConReq, ConConf, Data] :=

Connection_phase [ConReq, ConConf]

» Data_phase [Data]

where**process** connection_phase [ConReq, ConConf] :=ConReq; ConConf ; **exit****endproc****process** Data_phase [Data] :=

Data; Data_phase [Data]

endproc**endproc**

When B_1 is composed of several parallel sub-processes,
 B_1 terminates successfully iff all parallel sub-processes
terminate successfully

Disabling - Disruption

$B_1 [> B_2$

B_1 can be disrupted at any time by B_2

process Data_transfer [DatReq, DatConf, DisReq] :=

normal_transfer [DatReq, DatConf]

[> disconnect_phase [DisReq]

where

process normal_transfer [DatReq, DatConf] :=

DatReq; DatConf; normal_transfer [DatReq, DatConf]

endproc

process disconnect_phase [DisReq] :=

DisReq; **stop**

endproc

endproc

Observable events have a finer structure

An observable event = a gate name + **a list of values (or value expressions)**

Examples: $g\langle 5 \rangle$ $g\langle \text{true} \rangle$ $g\langle 3, \text{false} \rangle$ $g\langle \rangle$

The representations of **data values** (e.g. 3) and **value expressions** (e.g. $3+5$) in LOTOS are derived from the specification language for **abstract data types** (ADT) **ACT ONE**.

Principles of ACT ONE will be presented later.

Extended action-prefix - Part 1: Value declarations **$g!E; B$** E = value expression = an expression describing a data value

Examples:

 $!(3+5)$, $!(x+1)$, $!true$, $!'toto'$, $!not(x)$, $!min(x,y)$ In Basic LOTOS, **$g; B$** is a process that offers **g** and then behaves like **B** :Formally: **$g; B \xrightarrow{g} B$** In (full) LOTOS, **$g!E; B$** is a process that offers **$g<value(E)>$** and then behaves like **B** :Formally: **$g!E; B \xrightarrow{g<value(E)>} B$** Ex. : $g!(3+5); B \xrightarrow{g<8>} B$

Extended action-prefix - Part 2: variable declaration **$g?x:t; B$** x is a variable name t is a sort identifier. It indicates the domain of values over which x rangesExamples: $?x:\text{integer}$ $?x:\text{string}$ $?x:\text{boolean}$ $g?x:t; B(x)$ is a process that offers **all** events $g\langle v \rangle$ where v is any value in the domain of sort t and then behaves like $B(v)$ **$g?x:\text{nat}; B(x)$** thus allows all events in the set $\{g\langle 0 \rangle, g\langle 1 \rangle, g\langle 2 \rangle, \dots\}$ Formally: **$g?x:t; B \rightarrow g\langle v \rangle \rightarrow [v/x] B$** for every v in the domain of sort t where **$[v/x] B$** is the result of the replacement by v of every free occurrence of x in B .Ex. : $g?x:\text{integer}; h!(x+1); \text{stop} \rightarrow g\langle 3 \rangle \rightarrow h!(3+1); \text{stop}$ $g?x:\text{integer}; h!(x+1); \text{stop} \rightarrow g\langle 5 \rangle \rightarrow h!(5+1); \text{stop}$

Scope of variables

(sap1?x:int; sap2!x; **stop**) || (sap3!x; **stop**)

binding
occurrence
of x

scope of
?x:int

free occurrence
of x

Usual rules for nested scopes apply

Synchronization between two processes
Interprocess communication
1. Value matching

process A	process B	Synchron. condition	Type of interaction	Effect
g!E1	g!E2	value(E1) = value(E2)	value matching	synchronization

Example:

g is a gate modelling a service interface

sap_id is a value representing the **s**ervice **a**ccess **p**oint identifier

g!sap_id; B models interaction at the SAP identified by sap_id at the interface g

Synchronization between two processes
Interprocess communication
2. Value passing
3. Value generation

process A	process B	Synchron. condition	Type of interaction	Effect after synchron.
g?x:t "input"	g!E "output"	value (E) is (in domain) of sort t	value passing	x = value(E)
g?x:t "negotiation"	g?y:u	t = u	value generation	x = y = v and v is some value (in domain) of sort t

Extended action-prefix: several attributes

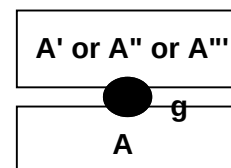
Example: **g!sap_id?x:primitive ; B**

Synchronization conditions in presence of several attributes

- Same number of attributes in both action offers
- Synchronization conditions for every pair of attributes

A := g!3?x:primitive; B

A' := g!3!connect_request; B' can synchronize with A
A'' := g!3!true; B'' cannot synchronize with A
A''' := g!4!connect_request; B''' cannot synchronize with A

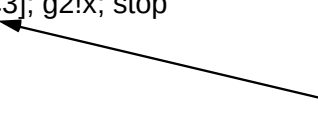


Selection Predicate

A **selection predicate** can be associated with an action denotation.

$g1?x:\text{nat } [x < 3]; g2!x; \text{stop}$

Predicate



This predicate may contain variables that occur in the **variable declarations** ($?x:t$)
It imposes restrictions on the values that may be bound to these variables.

$g1?x:\text{nat } [x < 3]; g2!x; \text{stop}$ has the following three possible transitions:

— $g1 < 0 > \rightarrow g2!0; \text{stop}$

— $g1 < 1 > \rightarrow g2!1; \text{stop}$

— $g1 < 2 > \rightarrow g2!2; \text{stop}$

A value negotiation between two processes
$$g?qos:nat [qos \leq \max]; B_1 (qos)$$
$$I[g]I$$
$$g?qos:nat [qos \geq \min]; B_2 (qos)$$
$$=$$
$$g?qos:nat [(qos \leq \max) \text{ and } (qos \geq \min)];$$
$$(B_1 (qos)$$
$$I[g]I$$
$$B_2 (qos))$$
Constraint-oriented style

Guarded Expressions

Any behaviour expression may be preceded by a guard (i.e. a predicate + an arrow).

Interpretation:

If the predicate holds, the behaviour expression that follows the guard is possible

Otherwise, the whole expression is equivalent to 'stop'

Example 1

$[x > 0] \rightarrow g!x; \text{next_process } [\dots] (x, \dots)$

$\square$

$[x < 0] \rightarrow g!-x; \text{next_process } [\dots] (x, \dots)$

si $x=1$: $g!1; \text{next_process } [\dots] (1, \dots)$

si $x=-2$: $g!2; \text{next_process } [\dots] (-2, \dots)$

si $x = 0$: stop

Example 2

$[x > 0] \rightarrow \text{processus1}$

$\square$

$[x = 5] \rightarrow \text{processus2}$

$\square$

$[x > 3] \rightarrow \text{processus3}$

Predicates need not be mutually exclusive

Parametric process and instantiation**Process declaration:****process** compare [in,out] (min,max:int) :=

in?x:int;

([min<x<max] -> out!x; compare [in,out] (min,max)

[] [x≤min] -> out!min; compare [in,out] (x,max)

[] [x≥max] -> out!max; compare [in,out] (min,x))

endproc**Process instantiation:** compare [one,two] (x,2*x)

Formal parameters are replaced by value expressions.

Name clashes are avoided by a suitable renaming.

compare [one,two] (x,2*x) has the same behaviour as:

one?y:int;

([x<y<2*x] -> two!; compare [one,two] (x,2*x)

[] [y≤x] -> two!x ; compare [one,two] (y,2*x)

[] [y≥2*x] -> two!2*x; compare [one,two] (x,y))

Sequential composition with value passing

```
Connection_phase [...](...)
>>
Data_phase [...](...)
```

One would like to express that the behaviour of the Data_phase depends on parameters that are established in the Connection-Phase.

- Data_phase will be specified as a parametric process
- We need a mechanism for instantiating these parameters when the first process terminates and enables the second.

Successful termination with value offers

The **exit** process can have a finite list of value expressions added to it.

```
gate1?x:int; gate2?y:int; gate3?z:int; exit (x,y,z)
tsap!cei ?qual:int ?exp_data: bool [qual>min]; exit (qual, exp_data)
```

Accepting values from the Enabling Process

Connection_Phase [...] (...)

>> **accept qos: qos_sort in** Data_Phase [...] (qos)

Requirement:

The sort **qos_sort** must match the sort of the value expression that Connection_Phase exits

This implies that in an expression like **B1 >> accept $x_1:t_1, \dots, x_n:t_n$ in B2**

B1 always exits the same number of value expressions of the same sorts.

This leads to the definition of the **functionality** of a process.

The functionality of a process

Functionality = the list of the **sorts** of the values offered at successful termination

```
Process connect1 [gate1, gate2, gate3] : exit (int, int, int) :=  
gate1?x:int; gate2?y:int; gate3?z:int; exit (x,y,z)  
endproc
```

Functionality

```
Processconnect2 [tsap] (tcei: tcei_sort, min: int) : exit (int, bool) :=  
tsap!cei ?qual:int ?exp_data: bool [qual > min]; exit (qual, exp_data)  
endproc
```

The construct **exit (any sort_id)** is used when

- a value must be passed on successful termination because the next process is expecting one,

BUT

- the process does not care about the value to be passed on successful termination

Rules on the functionality Examples
--

funct (stop) =	noexit
funct (exit) =	exit
funct (g; B) =	funct (B)
funct (B ₁ [] B ₂) =	funct (B ₁) if funct (B ₁) = funct (B ₂) or funct (B ₂) = noexit funct (B ₂) if funct (B ₁) = noexit undefined otherwise (static semantic error)
funct (B ₁ [[...]] B ₂) =	funct (B ₁) if funct (B ₁) = funct (B ₂) noexit if funct (B ₁) = noexit or funct (B ₂) = noexit undefined otherwise (static semantic error)
funct (B ₁ [> B ₂) =	funct (B ₁) if funct (B ₁) = funct (B ₂) or funct (B ₂) = noexit funct (B ₂) if funct (B ₁) = noexit undefined otherwise (static semantic error)
funct (B ₁ >> B ₂) =	funct (B ₂) if funct (B ₁) ≠ noexit noexit otherwise

Generalized choice

$B_1 \sqcup B_2$ expresses the choice among **two** behaviours

By associativity of the choice operator, one can express the choice among a **finite number** of behaviours as follows:

$$B_1 \sqcup B_2 \sqcup \dots \sqcup B_n$$

The generalized choice operator allows one to specify the choice among a **possibly infinite set** of behaviours as follows:

Let $B(x)$ be a behaviour expression that depends on the (free) variable x of sort nat :

Choice $x:\text{nat} \sqcup B(x)$ expresses the choice among the behaviours $B(v)$ for all v of sort nat .

In other words, it is equivalent to: $B(0) \sqcup B(1) \sqcup B(2) \sqcup \dots$

General form : choice $x_1:t_1, \dots, x_n:t_n \sqcup B(x_1, \dots, x_n)$

Let operator

It allows the association of value expressions to free variables:

let $x_1:t_1 = E_1, \dots x_n:t_n = E_n$ **in** $B(x_1, \dots x_n)$

The x_i 's are free variables in B

The t_i 's are the sorts of these variables

The E_i 's are the value expressions associated with the x_i 's

Abstract data types The ACT ONE langage

- **Notion of abstract data type**
- **Specification of an ADT**
- **Combining of ADT**
- **Usage of ADT**

Notion of abstract data type

Classical data types

Ex: "Unsigned integers, declared **unsigned**, obey the laws of arithmetics modulo 2^n , where n is the number of bits in the representation."

C Language Definition, X/OPEN Portability Guide

Abstract Data Types (= ADT)

- Formal (mathematical) definition —> no ambiguity.
- No reference to implementation.
- (Almost) no "built-in" law - all properties are written.

What is to be defined ?

- Sets of data values = **sorts**

Ex: booleans, natural numbers, ...

- Functions to handle these values = **operations**

Ex: addition over naturals, negation over booleans, ...

These are abstract mathematical objects — No physical organization is defined !

Specification of an ADT: sorts and operations

Specification of an ADT in ACT ONE

In three parts: **sorts**, **operations** and **equations**

I. Specification of sorts

sorts *Nat* (* definition of a new sort "Nat" *)

—> introduces a domain whose content remains to be defined.

II. Specification of operations

opns *succ* : *Nat* -> *Nat* (* successor of a *Nat* is a *Nat* *)
 0 : -> *Nat* (* constant 0 is a *Nat* *)
 + : *Nat*, *Nat* -> *Nat* (* "+" is used in infix notation *)

sorts + operations (over these sorts) = a **signature**

Combining operations yields **terms**

= representations of values contained in the sorts.

Ex: *0*, *succ(0)*, *0+0*, *0+succ(0)*, *succ(0+0)*...

A priori *succ(0) ≠ 0+succ(0) ≠ succ(0+0)* —> we must add **equations**.

Specification of an ADT: equations

III. specification of equations

eqns *forall* $x, y : \text{Nat}$ (* for all terms x, y of sort Nat *)
 ofsort Nat (* sort of sides of equations *)
 $x + 0 = x ;$ (* $x+0$ denotes the same value as x *)
 $x + \text{succ}(y) = \text{succ}(x+y) ;$

Conditional equations are allowed:

$y = 0 \Rightarrow x + y = x ;$ (* if $y = 0$ then $x + y = x$ *)

Equations define an **equivalence relation** between terms.

Each **equivalence class** corresponds to a single value that all terms of this class denote.

Ex: the (abstract) value 1 is defined by the class (of terms)

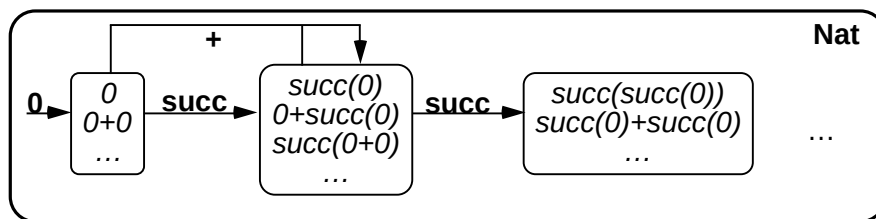
$\{ \text{succ}(0), \text{succ}(0)+0, \text{succ}(0+0), \dots \}$

Specification of an ADT: complete definition

I + II + III = specification of a data type

```

type      NaturalNumber is
sorts    Nat
opns     succ : Nat      -> Nat
           0 :          -> Nat
           _+_ : Nat, Nat -> Nat
eqns     forall x, y : Nat
           ofsort Nat
           x + 0      = x ;
           x + succ(y) = succ(x+y) ;
endtype
  
```



Combining of ADT: inheritance

```
type natural_number is  
  sorts nat  
  opns  0 : -> nat  
         succ : nat -> nat  
endtype
```

```
type nat_number_with_addition is nat_number  
  opns  + : nat, nat -> nat  
  eqns  forall x,y : nat  
         ofsort nat  
         x + 0 = x ;  
         x + succ(y) = succ(x+y) ;  
endtype
```

nat_number_with_addition inherits sorts, operations and equations of *natural_number*.

Permits structuring of definitions in a hierarchical manner.

Combining of ADT: library, renaming

library *Boolean, NaturalNumber, Bit* **endlib**

Only one predefined library, containing various base types:

Boolean, Element, NaturalNumber, Set, String,

DecimalDigit, HexGigit, OctDigit, Bit,

DecimalString, HexString, OctString, BitString, Octet, OctetString.

cf. annex A of ISO IS 8807.

type *bit is boolean* **renamedby**

sortnames *bit* **for** *bool*

opnnames *0* **for** *false*

1 **for** *true*

endtype

Creates a new type with same structure as source type (isomorphic).

Combining of ADT: parametrized types

```
type data is  
  formalsorts data  
  formalopns error_data :      -> data  
endtype  
  
type queue_data is data  
  sorts queue  
  opns empty_queue :      -> queue  
      add : data, queue   -> queue  
      first : queue       -> data  
  eqns forall x, y : data, q : queue  
    ofsort data  
      first(empty_queue)      = error_data ;  
      first(add(x,empty_queue)) = x ;  
      first(add(x,add(y,q)))   = first(add(y,q)) ;  
endtype
```

The formal sorts, operations and equations define necessary conditions that parameters must fulfill.

Combining of ADT: actualization

```
type nat_number_queue is queue_data
      actualizedby nat_number using
    sortnames   nat      for data
                natqueue for queue
    opnnames    0        for error_data
endtype
```

The formal elements of *queue_data* are actualized by elements of *nat_number*, according to the given bindings.

Remarks:

- a binding may be omitted if formal and actual elements have the same name.
- a binding for a non-formal element is considered as a renaming
(Ex: *queue* $\longrightarrow$ *natqueue*)

Usage of ADT: structure of a specification

```
specification <specification-identifier> <parameter-list>
...
  type ... endtype
  library ... endlib
  type ... endtype
...
behaviour <behaviour-expression>
where
  process <process-identifier> < parameter-list > :=
    < behaviour-expression >
  where ...
  endproc
...
  type ... endtype
  library ... endlib
...
  process < process-identifier > < parameter-list > :=
    < behaviour-expression >
  where ...
  endproc
endspec
```

Usage of ADT: ADT in behaviour expressions

- action denotation $g !v ?x:s; B(x)$
- selection predicates $g ?x:s [x = y]; B(x)$
 $g ?x:s [predicate(x)]; B(x)$
- guards $[x = y] \rightarrow B$
 $[predicate] \rightarrow B$
- parameters and functionality **process** $P [g] (x:s) : \mathbf{exit}(s) := B(x) \mathbf{endproc}$
 $P [g] (v) \gg \mathbf{accept} x:s \mathbf{in} B(x)$
- generalized choice **choice** $x:s [] B(x)$
- local value **let** $x:s = v \mathbf{in} B(x)$

Remark: s is the name of a sort, not of a type !

A type = a collection of sorts, operations and equations, useful only for structuring and organizing the ADT definitions.

LOTOS summary

- **Mathematically well-defined** : developed from a well-established body of theory
 - Process algebra + Abstract Data Types**
 - Operational semantics**
- **Constructive: Symbolically executable**
- **Expressive: allows the description of**
 - **Architectures**
 - **Dynamic behaviours of processes**
 - Concurrency
 - Nondeterminism
 - Synchronous and asynchronous communications
 - **But no explicit time, no priorities**
- **Supports various levels of abstractions**
- **Supports the description of large systems**