Structured textual descriptions of systems: process algebras

Process algebras (PA)

Theoretical formalisms on which various specification and verification tools rely.

Examples of PA for asynchronous systems:

- **CCS** (*Calculus of Communicating Systems*) [Milner-80, Milner-89]
- **CSP** (*Communicating Sequential Processes*) [Hoare-85]
- **ACP** (*Algebra of Communicating Processes*) [Bergstra-Klop-84]

Standardized language: **LOTOS** [ISO-88]

Modernized language: **LNT** (LOTOS New Technology)

Textual (non graphical) formalisms

- Unlike CA, PA have a textual syntax (≈ programs)
- Same confrontation as between structured textual programs and charts in the sequential domain
- Advantages :
  - more expressive power (structuring)
  - scaling (large size specifications)
- Drawbacks:
  - syntax may be less intuitive

Approaches to specify an automaton

- Graphical: CA, Petri nets, SDL language

- Enumeration of states and transitions (Estelle):

  state BUSY, IDLE, …

  trans from BUSY **to** IDLE

  x := x + 1

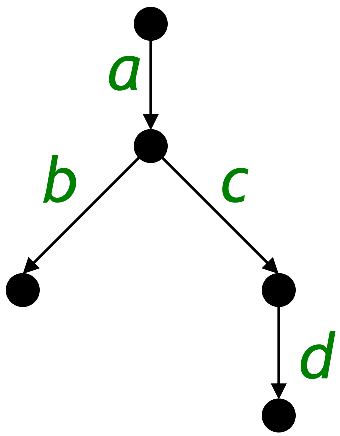
  Mecanism similar to « goto »

  Lack of structure $\Rightarrow$ « spaghetti » effect

- Use of regular expressions

Regular expressions

- In language theory: equivalence between regular expressions (RE) and finite automata
- PA use a variant of RE to define automata
- Some equivalences of language theory are not valid anymore: $a.(b + c) \neq a.b + a.c$



In CCS : $a.(b.\mathbf{nil} + c.d.\mathbf{nil})$

In LNT :

a; **select** *b* [] *c*; *d* **end select**; **stop**

Regular expressions (cont'd)

- PA use three operators similar to RE:
  - ***choice*** ('+' in CCS, '[]' in CSP, '**select**' in LNT), to model branches
  - ***sequence*** (' . ' in CCS, ' ; ' in CSP and LNT), to model action sequences
  - ***fix-point*** (recursive process call in CCS and LNT, '**loop**' in LNT), to model circuits

- Advantage of RE: structured programming

Parallel composition operators

PA have one (or several) parallel composition operators that can be freely combined with the other operators (choice, sequence, etc.)

- In CCS : ‘ | ’
- In CSP : ‘ || ’
- In LNT : ‘ **par** $G_1, \ldots, G_n$ ’ ($\approx \otimes_{\{G1,\ \ldots,\ Gn\}}$ of CA)

In PA, one can write (almost):

$a \,.\, (b \mid c) \,.\, d$

Building behaviours

- Complex behaviours can be described as algebraic expressions built using PA operators:

  $(a \mid b) \,.\, (c + d \,.\, (e \mid f))$

- **The idea of PA**: « lego » game
  A small number of primitive operators, each of which expresses one concept and that can be composed arbitrarily (orthogonality)

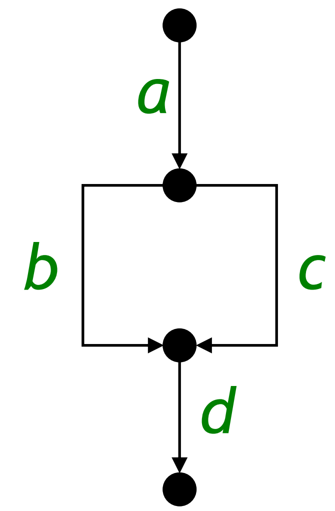
Formal semantics

- A program in PA is an algebraic term

  $a \,.\, (b + c) \,.\, d$

  to which can be associated a model.

  Most of the time, it is an LTS
  (e.g., for CCS, CSP, LOTOS, LNT).



- To define semantics of a PA, it suffices to define semantics of each operator

Algebraic approach (axiomatic semantic)

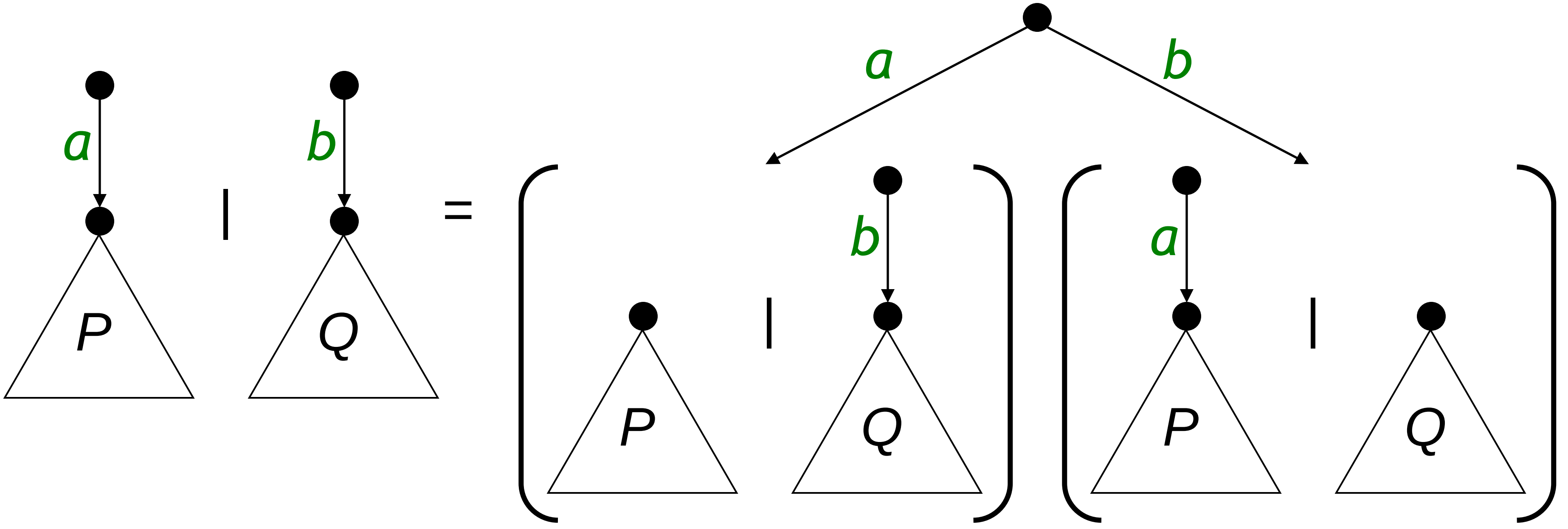
- Method used to define semantics of CCS and ACP
- Definition of a set of axioms describing the relations between operators
- Several axiomatisations can be possible $\Rightarrow$ problem of *consistency* and *completeness*
- Examples of axioms:

$B_1 + B_2 = B_2 + B_1$	commutativity of +
$\mathbf{nil} + B = B$	**nil** neutral element for +
$\mathbf{nil} \mid B = B$	**nil** neutral element for $\mid$

Expansion theorem

The *expansion theorem* [Milner] allows parallelism (|) to be replaced by choice (+) and sequence (.):

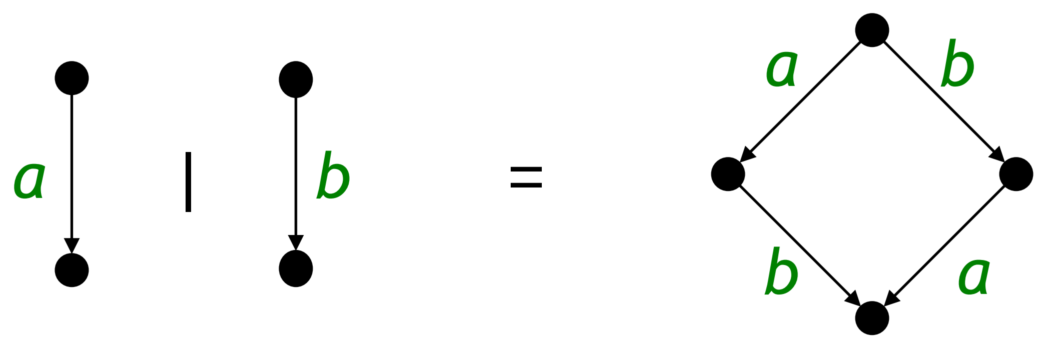
$$a.P \mid b.Q = a.(P \mid b.Q) + b.(a.P \mid Q)$$



Interleaving

- The expansion theorem is the basis of the interleaving semantics (cf. CA)

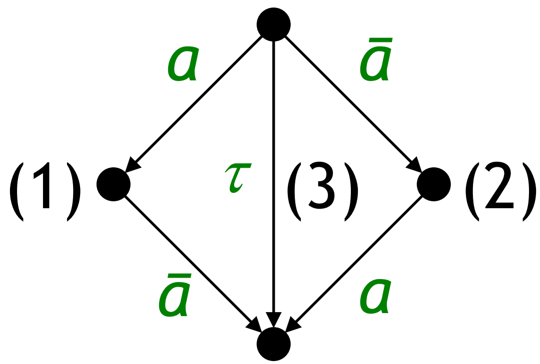
$$a.\mathbf{nil} \mid b.\mathbf{nil} = a.b.\mathbf{nil} + b.a.\mathbf{nil}$$



Synchronisation in CCS

- In CCS: notion of «complementary gates »
- Example of synchronisation on two gates a and $\bar{a}$:

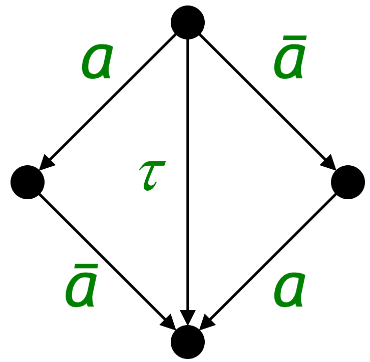
$$a.P \mid \bar{a}.Q = \begin{cases} a.(P \mid \bar{a}.Q) + & (1) \\ \bar{a}.(a.P \mid Q) + & (2) \\ \tau.(P \mid Q) & (3) \end{cases}$$



- The operator '|' expresses that both processes can (3) or not (1 and 2) synchronise
- In practice, we want to force synchronisation ⇒ a *restriction* operator is necessary to forbid (1) and (2):

$$(a.P \mid \bar{a}.Q) \setminus a$$

Synchronisation in CCS (cont'd)



- After synchronisation of a and $\bar{a}$, the transition becomes hidden (renamed to the internal action τ) $\Rightarrow$ in CCS, only binary rendezvous can be modeled
- Better solutions in CSP, LOTOS, LNT: *n-ary* rendezvous

Axiomatic semantics (cont'd)

- The **axiomatic semantics** allow program *transformations* by applying the axioms and the expansion theorem

  The correctness of a parallel program can be shown by transforming it in a simpler sequential program [Milner-89]
- **Drawback**: in practice, combinatory explosion due to the expansion of parallelism $\Rightarrow$ big size algebraic terms have to be analysed

Operational Semantics

- Used to define the semantics of LOTOS and LNT
- The behaviour of an algebraic term is defined by a set of derivation rules that allow the corresponding LTS to be generated
- Ex. of rules (semantics of choice) : op + of CCS

$$\frac{B_1 \xrightarrow{L} B_1'}{B_1 + B_2 \xrightarrow{L} B_1'} \qquad \frac{B_2 \xrightarrow{L} B_2'}{B_1 + B_2 \xrightarrow{L} B_2'}$$

Remark: no problems of consistency or completeness, if some sufficient conditions on the rule format are satisfied (SOS - *Structural Operational Semantics*)

Translation of terms into LTS

Let B_0 be a PA term

How to build the corresponding LTS model?

Associate to each term a state of the LTS

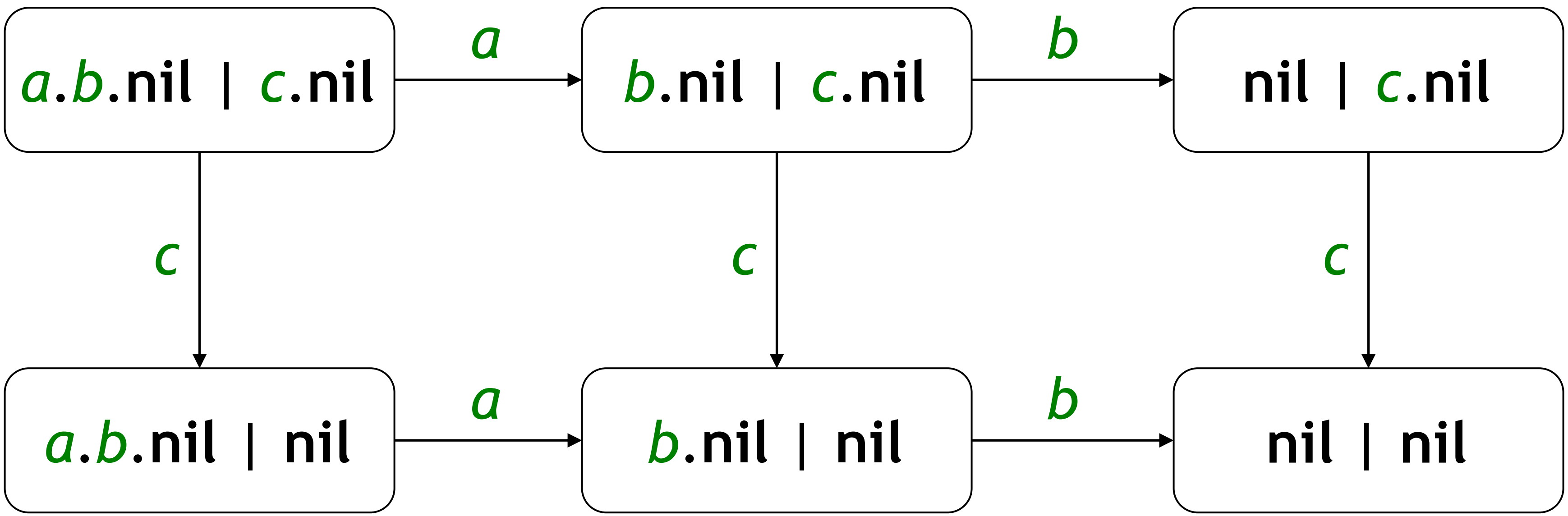
0. Initial set of states to be explored = { B_0 }
1. If states remain to be explored, choose a state B among them
2. For each rule applicable to B, which expresses the execution of a transition $B \xrightarrow{L} B'$ in the LTS, add B' to the set (if not already explored)
3. Add the transition $B \xrightarrow{L} B'$ to the LTS and continue in 1

Example

Construct the LTS corresponding to the CCS term ($a.b.\mathbf{nil} \mid c.\mathbf{nil}$)

Example

Construct the LTS corresponding to the CCS term $(a.b.\mathbf{nil} \mid c.\mathbf{nil})$



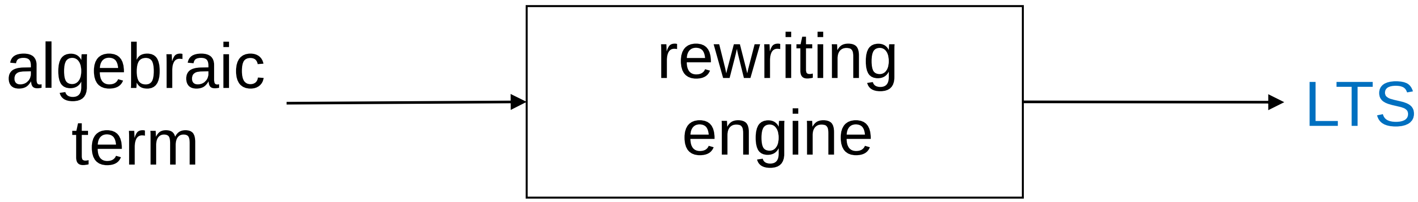
Remarks

- **Automated** Method to generate the LTS model corresponding to an algebraic specification
- Well-defined SOS rules $\Rightarrow$ termination and confluence of the construction method
- Similar to the construction of the product automaton for CA
- Once the LTS has been constructed, the state contents is not anymore necessary (the system behviour is defined by the LTS *actions*)
- Verification techniques (*model-checking*, *equiv.-checking*) can be applied to the LTS

Implementation

<u>1st solution</u>: term rewriting

- The SOS rules are given as input to a rewriting engine



- Not efficient (manipulation of large terms) but automatisable
- Examples : *Process Algebra Compiler*, *Concurrency Workbench* (CCS), *HIPPO*, *SMILE* (LOTOS)

Implementation (cont'd)

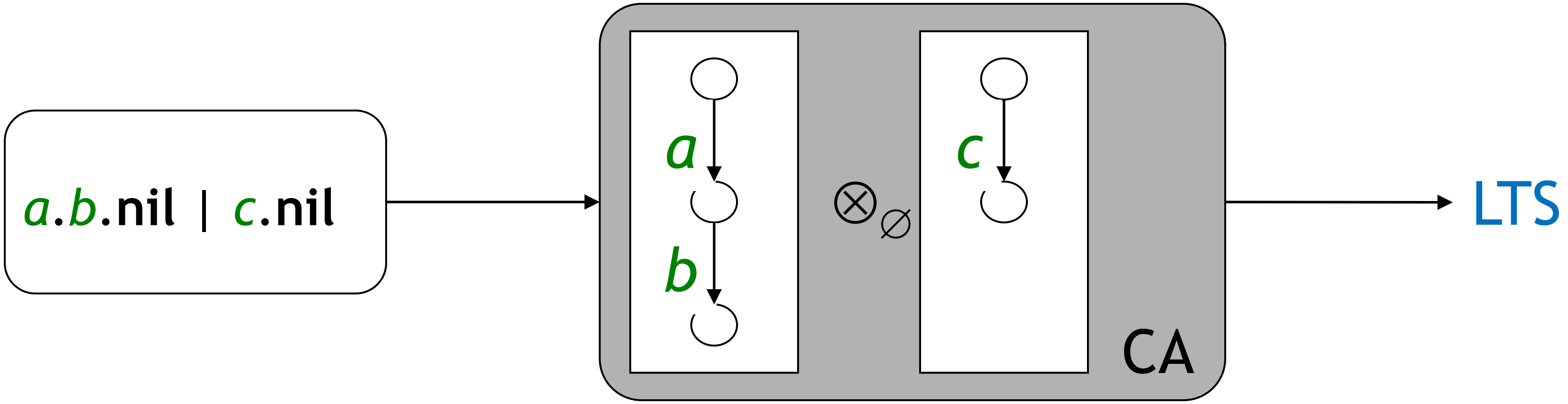
The solutions based on rewriting have two sources of inefficiency:

- **High memory consumption**
  - The LTS state is an algebraic term (syntax tree)
  - The initial state is the tree of the source program
  - The next states are the expanded trees
- **Low speed**
  - Slow execution of transitions (rewriting)
  - Slow state comparison (tree traversal)

Implementation (cont'd)

2nd solution: translation of the term into a system of CA, then construction of the product automaton

Example :

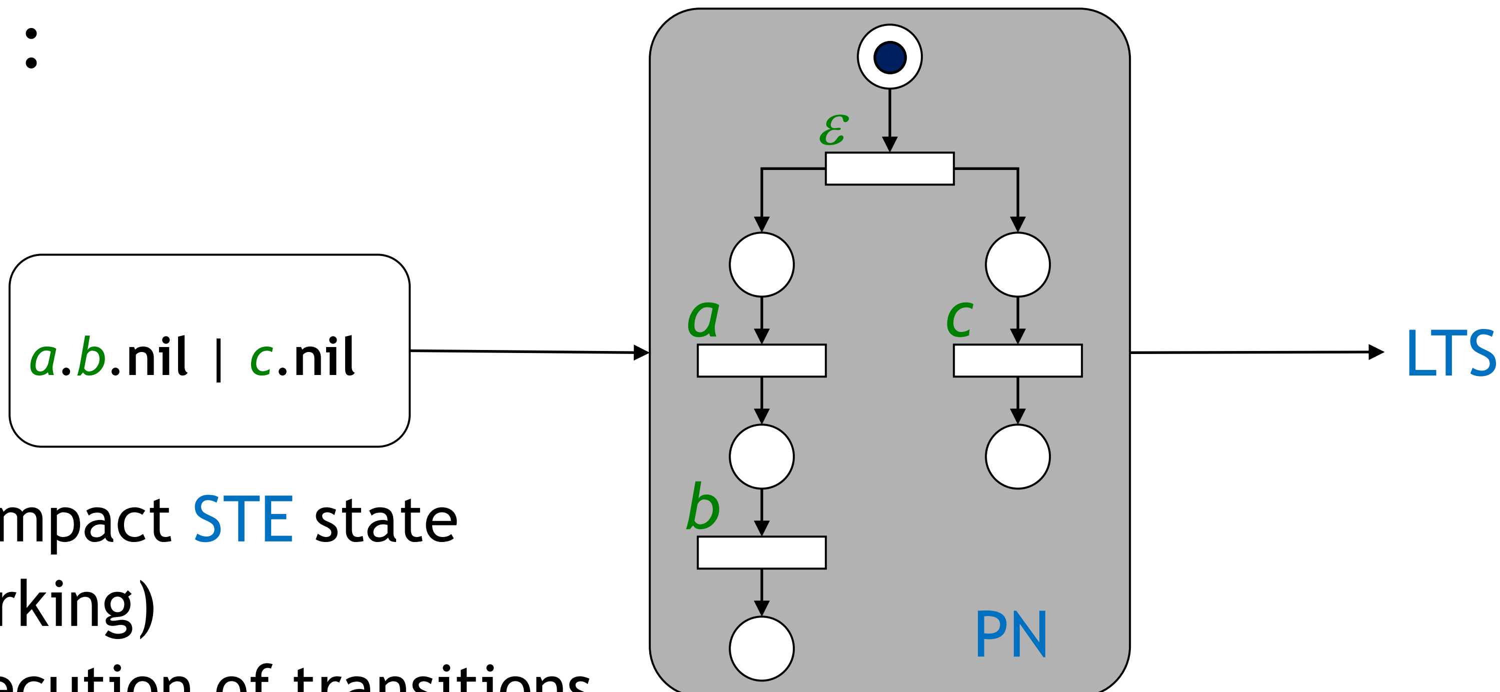


- very compact LTS state (list of CA state numbers)
- fast execution of transitions (synchronous product)
- but less general solution

Implementation (cont'd)

3rd solution: translation into a *Petri net* (PN), then construction of the *marking graph*

Example :



- very compact STE state (PN marking)
- fast execution of transitions (computation of the successor marking)
- more general than CA

Exercise

- Given the simplified SOS rules for CCS below (without synchro), draw the LTS of the term

a.(d.**nil** + e.**nil**) + b.c.(d.**nil** | e.**nil**)

$$\frac{}{a.B \xrightarrow{a} B}$$

$$\frac{B_1 \xrightarrow{a} B_1'}{B_1 + B_2 \xrightarrow{a} B_1'} \qquad \frac{B_2 \xrightarrow{a} B_2'}{B_1 + B_2 \xrightarrow{a} B_2'}$$

$$\frac{B_1 \xrightarrow{a} B_1'}{B_1 \mid B_2 \xrightarrow{a} B_1' \mid B_2} \qquad \frac{B_2 \xrightarrow{a} B_2'}{B_1 \mid B_2 \xrightarrow{a} B_1 \mid B_2'}$$

a.(d.**nil** + e.**nil**) + b.c.(d.**nil** | e.**nil**)

The LNT language

Origins and structure of LNT

Successor of LOTOS (international standard for the formal specification of telecommunication protocols and distributed systems [ISO 8807])

Variant of the E-LOTOS standard [ISO 15437]

Combination of the PA concepts with a classical algorithmic language (imperative / functional)

- *data* part (to define and access data structures): types, functions, instructions
- *control* part = super-set of the data part: processes

Remark: a data part is necessary to describe realistic systems. There exist subsets of PA without data (« pure CCS », « basic LOTOS »), which are not usable in practice.

Tools for the LNT language

- LNT supported by the CADP toolbox (INRIA/CONVECS): simulation, formal verification (*model checking*), ...
- Translation into LOTOS (lnt2lotos, lnt.open)
- Complete documentation [doc]: see Chamilo or
  $CADP/doc/pdf/Champelovier-Clerc-Garavel-et-al-10.pdf
- **Remark**: CADP is available on ensipc*x* where $x \in 82 .. 103$
  To use it, define the following environment variables:
  export CADP=/matieres/5MMMVSC7/Cadp
  PATH="$CADP/com:$CADP/bin.`$CADP/com/arch`:$PATH"

Preliminary example: the Peterson algorithm (1/3)

```
module Peterson is

channel bool is (bool) end channel

channel nat is (nat) end channel

process D [R, W : bool] is
   var b : bool in
      b := false;
      loop
         select R (b) [] W (?b) end select
      end loop
   end var
end process

[...]
```

Reminder pseudocode

Shared variables

var d_0 : bool := false
var d_1 : bool := false
var t $\in$ {0, 1} := 0

Preliminary example: the Peterson algorithm (2/3)

```
process P [Wm, Rn: bool, RT, WT: nat, NCS: none, CS: nat] (m: nat) is
  var dn : bool, t : nat in
    loop
      NCS; Wm (true); WT (m);
        loop wait in
          Rn (?dn);
          RT (?t);
          if not (dn) or (t != m) then
            break wait
          end if
        end loop;
        CS (m); Wm (false);
    end loop
  end var end process
```

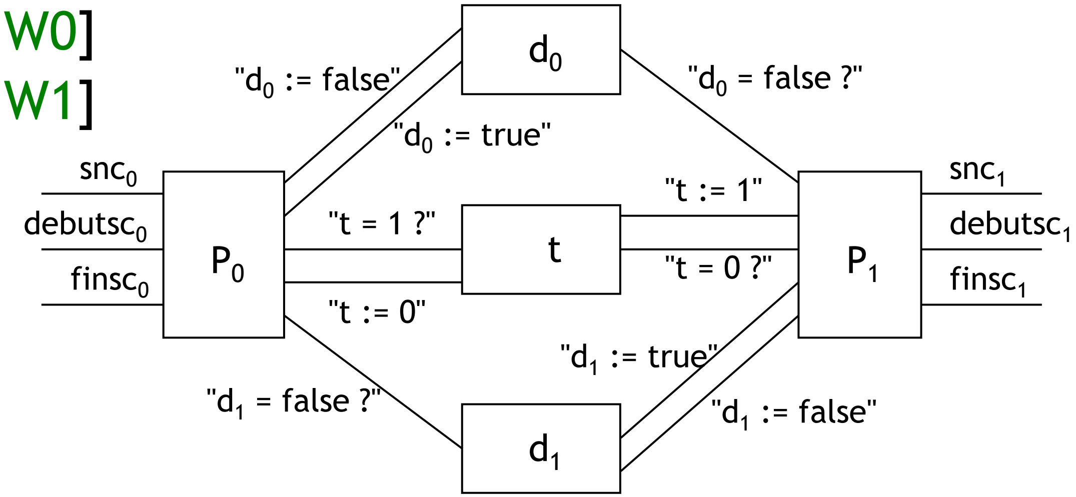
Reminder pseudocode

loop forever { P_m }
1 : { snc_m }
2 : d_m := true
3 : t := m
4 : **wait** ($d_{(1-m)}$ = false **or** t = m)
5 : { $debutsc_m$ }
6 : { $finsc_m$ }
7 : d_m := false
endloop

Preliminary example: the Peterson algorithm (3/3)

```
process Main [NCS : none, CS : nat] is
   hide R0, W0, R1, W1 : bool, RT, WT : nat in
      par R0, W0, R1, W1, RT, WT in
            par     P [W0, R1, RT, WT, NCS, CS] (0)
            ||      P [W1, R0, RT, WT, NCS, CS] (1)
            end par
      ||
            par     T [RT, WT]
            ||      D [R0, W0]
            ||      D [R1, W1]
            end par
      end par
   end hide
end process

end module
```



Data types

- Predefined type: **bool**, **char**, **nat**, **int**, **real**, **string**
- User-defined type: constructor type
  (inherited from functional programming: ML, Haskell)
  type *T* **is** *E* **end type**
  where *T* is an identifier and *E* is a type expression
- General case: E is a list of definitions of the form
  $C\ (X_1 : T_1, \ldots, X_n : T_n)$ where
  C is an identifier called constructor
  $X_1, \ldots, X_n$ are field identifiers
  $T_1, \ldots, T_n$ are type identifiers
- There exist abbreviations:
  set of *T*, **list of** *T*, **array [** n **..** m **] of** *T*, ...

Examples of user-defined types

- **Enumerated:** **type** gender **is** male, female **end type**
- **Record:**
  type pers **is**
  tuple (name : **string**, age : **nat,** gen : gender) **end type**
- **Union:** **type** intbl **is** intv (n : **int**), boolv (b : **bool**) **end type**
- **List:** **type** ilist **is** nil, cons (hd : **int**, tl : ilist) **end type**
  or more simply: **type** ilist **is list of int end type**
- **Binary tree:**
  type tree **is** leaf, node (fg : tree, fd : tree) **end type**
- **Array** (static size):
  type parray **is array [** 1 **..** 4 **] of** pers **end type**

Other type functionalities

- External types in C

  type T **is** !**external** !**implementedby** "C_T"

- Generated operations for user-defined types

  type natpair **is** p (n1, n2 : **nat**) **with** set, get **end**
  (field accessors and updaters)

  type gender **is** male, female **with** ==, != **end type**
  (comparison operations),

  type ilist **is list of int with** member, union **end type** (list operations)

- Interval types and predicate types

  type r **is range** -2 .. 2 **of int end type**

  type even_nat **is** X : nat **where** X mod 2 == 0 **end type**

Function definition (1/2)

Example (in parameter and result):

```
function fact (n : nat) : nat is
    var res : nat in
        res := 1;
        while n > 1 loop
            res := res * n; n := n - 1
        end loop;
        return res
    end var
end function
```

Call examples:

```
X := fact (4 of nat)
Y := fact (X) + 1
```

Function definition (2/2)

Example (in, out and in out parameters):

incrementation of a 16 bit counter with overflow check:

```
function incr16 (in out n : nat, in delta : nat,
                              out overflow : bool) is
    if n >= 65535 – delta then overflow := true
    else overflow := false; n := n + delta end if
end function
```

Call example:

```
eval incr16 (!?c, 4, ?b)
```