Applied Concurrency Theory Lecture 3 : Next generation process calculi

Hubert Garavel

Alexander Graf-Brill

Saarland University

updated in Dec. 2021

Beyond classical process calculi - E-LOTOS and LNT

E(nhanced)-LOTOS

- Early 90s:
  - great academic expectations in LOTOS
  - but disapointing industrial feedback: steep learning curve and lack of trained designers/engineers
  - could LOTOS be made more ‘acceptable’ by industry?
- Between 1992 and 2001
  - ISO/IEC standardization work to ‘enhance’ LOTOS
  - modest repairs as well as ambitious new features (real-time)
  - converged to E-LOTOS international standard (ISO 15437)
  - much too complex
  - never implemented (?)

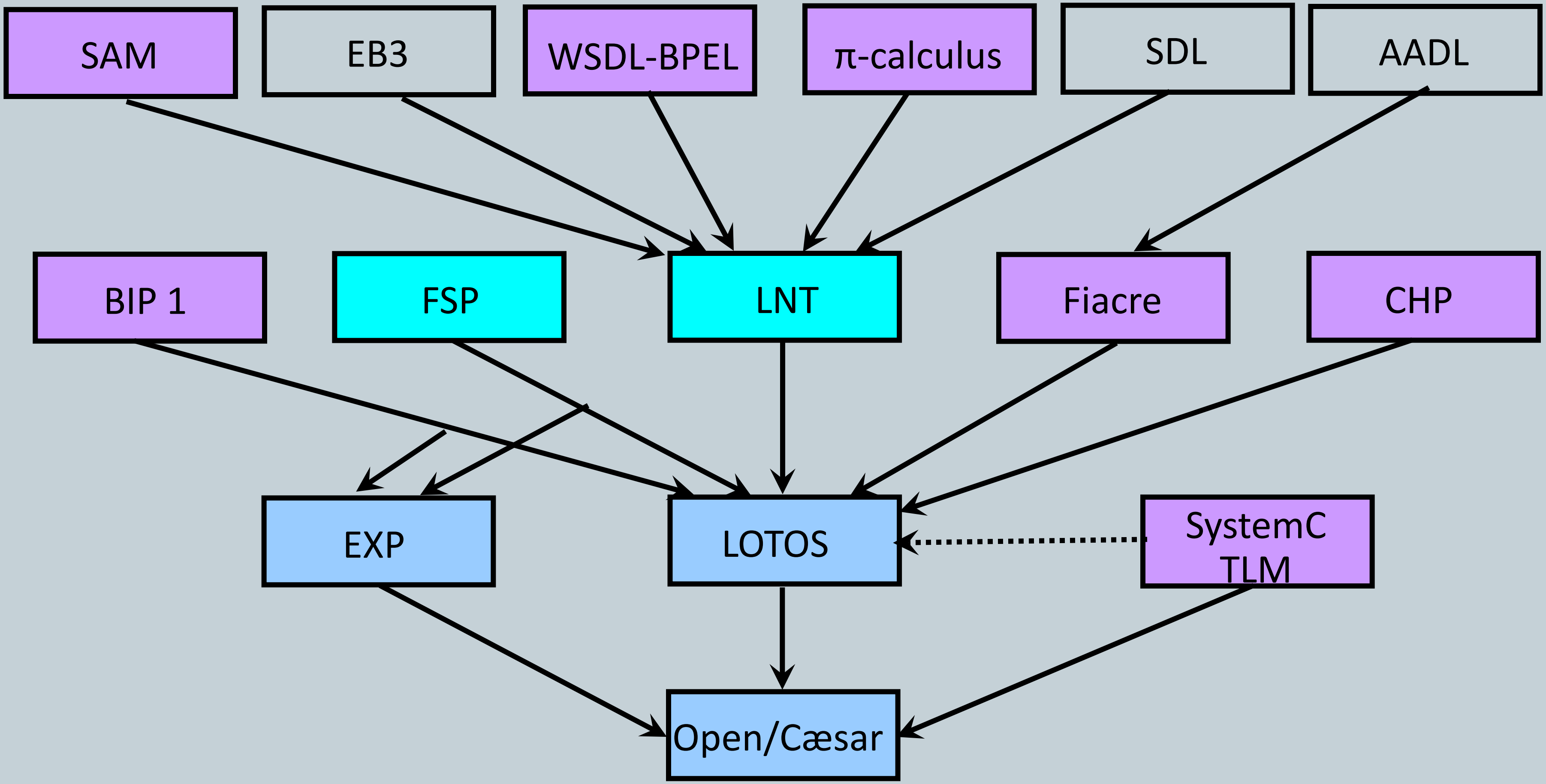
LNT

- Motivation at INRIA Grenoble:
  - LOTOS is expressive and adapted to study concurrency
  - it is well-equiped with tools (that took decades to build)
  - E-LOTOS has failed its initial expectations
  - persistent need of a better language for concurrency
  - what can be saved from LOTOS and E-LOTOS?
- LNT (= LOTOS New Technology)
  - dialect of E-LOTOS developed at INRIA since 1995
  - inspired by our participation to ISO committee on E-LOTOS

LNT history and tools

- First implementation: LNT $\rightarrow$ C
  - TRAIAN compiler (1998-2008)
  - alas: wrong compiler construction technology
  - only the data types are compiled
  - internally used to build compilers and translators (a dozen)
- Second implementation: LNT $\rightarrow$ LOTOS
  - goal: reuse of existing LOTOS tools at minimal cost
  - development of LNT2LOTOS / LNT / LPP (2005-now)
  - progressively built with funding of Bull
  - successfully used at Bull, CEA/Leti, STMicroelectronics
  - since Jan 1st 2010, we replaced LOTOS with LNT

LNT as a pivot language



Lexical/syntactic elements of LNT

- 1. Unify the data types and the process parts
- 2. Break away from the 'algebraic mania'
  - *computer scientists are not mathematicians ⇒ specifications do not need to be algebraic terms*
  - n-ary operators become possible (e.g., n-ary parallel)
  - imperative programming constructs are back (if, case, while)
  - Ada-like bracketed syntax (if ... end if) avoids ambiguities
- Also:
  - case-sensitive identifiers, with additional constraints: either 'X' or 'x', but not both in the same scope (LOTOS is case-insensitive: 'X' and 'x' are the same)
  - two types of comments: Pascal-like (* ... *) or Ada-like -- ... \n

LNT modules

LNT modules

- Compilation unit, containing
  - types
  - functions
  - channels (= gate types)
  - processes
- One module = one file (of the same name)
  - no modules nested within modules
- Modules can import other modules
- *Principal module* containing the *root process* (called "MAIN" by default)
- Case insensitive module names, but
  - all modules in the same directory
  - no two files differing only by case

Example of LNT modules


```
module PLAYER is
    ...
end module
```

file “PLAYER.lnt”

list of imported modules

```
module Team (PLAYER) is
    ...
end module
```

file “TEAM.lnt”

or (one of):

- “Team.lnt”
- “team.lnt”
- “TeAm.lnt”
- …

LNT types

Overview

- Inductive types
  - set of constructors with named and typed parameters
  - special cases: enumerations, records, unions, trees, etc.
  - shorthand notations for arrays, (sorted) lists, and sets
  - subtypes: range types and predicate types
  - automatic definition of standard functions: "==", "<=", "<", ">=", ">" , field selectors and updaters
  - pragmas to control the generated names in C and LOTOS
- Notations for constants (C-like syntax):
  - natural numbers: 123, 0xAD, 0o746, 0b1011
  - integer numbers: -421, -0xFD, -0o76, -0b110
  - floating point numbers: 0.5, 2E-3, 10.
  - characters: 'a', '0', '\n' , '\\', '\‘'
  - character strings: “hello world”, “hi!\n”

Examples of LNT types (1)

Enumerated type

```
type Weekday is (* LOTOS-style comment *)
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
end type
```

Record type

```
type Date is -- ADA-style comment (to the end of the line)
    date (day: Nat, weekday: Weekday, month: Nat, year: Nat)
end type
```

Inductive Type

```
type Nat_Tree is
    leaf (value: Nat),
    node (left: Nat_Tree, right: Nat_Tree)
end type
```

Examples of LNT types (2)

Shorthand notation

```
type Nat_List is
  list of Nat
end type
```

instead of

```
type Nat_List is
 nil,
 cons (head: Nat, tail:Nat_List)
end type
```

Automatic definition of standard functions

```
type Num is
  one, two, three
  with "==", "<=", "<", ">=", ">"
end type
type Date is
  date (d: Nat, wd: Weekday, month: Nat, year: Nat)
  with "get", "set" -- for selectors X.D, … and updaters X.{D => E}
end type
```

Examples of LNT types (3)

One-dimensional array

```
type Vector is
  array [ 0 .. 3 ] of Int
end type
```

Two-dimensional array

```
type Matrix is -- square-matrix
  array [ 0 .. 3 ] of Vector
end type
```

Array of records

```
type Date_Array is
  array [ 0 .. 1 ] of DATE
end type
```

Examples of LNT types (4)

Range types (intervals)

```
type Index is
   range 0 .. 5 of Nat
   with "==", "!="
end type
```

further automatically definable functions:
functions:
first, last, card

Predicate subtypes

```
type EVEN is
      n: NAT where n mod 2 == 0
end type
type PID is
   i: Index where i != 0
end type
```

LNT functions

Overview

- An imperative-like syntax (with assignments)
- But a strictly functional semantics (no side effects)
- Ensured by type checking and initialization analysis
- Expressions are much richer than in LOTOS:
  - Local variable declarations and assignments: "var"
  - Sequential composition: ";"
  - Breakable loops: "while" and "for"
  - Conditionals: "If-then-else"
  - Pattern matching: "case"
  - (Uncatchable) exceptions: "raise"
- Three parameter passing modes:
  - "in" (call by value)
  - "out" and "in out" (call by reference)
- Function overloading
- Support for external functions (LOTOS and C)

call syntax requires "eval" keyword

Examples of LNT functions (1)

Constants

```
function pi: Real is
  return 3.14159265
end function
```

Field accesses

```
function get_weekday (d: Date): Weekday is
    return d.wd
end function
function set_weekday (in out d: Date, newd: Weekday) is
    d := d.{wd => newd}
end function
```

Examples of LNT functions (2)

Access to the first element of a list L

```
function get_head [Empty_List: none] (L: Nat_List) : Nat is
   case L var head: Nat in
      nil ->                              raise Empty_List
    | cons (head, any Nat_List) ->        return head
   end case
end function
```

Update of element (i,j) of a matrix M

```
function update (in out M: Matrix, i, j: Nat, new_e: Nat) is
   var v: Vector in
      v := M[i];
      v[j] := new_e;
      M[i] := v
   end var
end function
```

Examples of LNT functions (3)


```
function reset_diagonal_elements (M: Matrix) : Matrix is
   var
      result: Matrix,
      i: Nat
   in
      result := M;
      for i := 0 while i < 3 by i := i + 1 loop
         eval update (!?result, i, i, 0)
      end loop;
      return result
   end var
end function
```

LNT channels

Channels (or: gate typing)

- In LOTOS, gates are untyped:
  - allowed: G !0 ; G !true; G !cons (A, nil) !false; stop
  - allowed: G !true; B_1 || G ?X:nat; B_2
  - typing errors are not caught statically and cause deadlock at run-time

- LNT enables ‘channels’ (i.e. gate types)
- Gates must be declared with a channel
- Channels can be overloaded (different type tuples for the same gate)
- There is a predefined channel ‘any’ (untyped) for backward compatibility with LOTOS (not recommended)
- Gate typing is implemented by generating extra LOTOS code that will not type check if there is a gate type error

Examples of channels


```
channel None is
    ()
end channel

channel BoolChannel is
    (B: Bool)
end channel

channel C2 is
    (P: Pid, B: Bool),
    (S: Signal, N1, N2: Nat)
end channel
```

LNT processes

Overview

- Processes are a superset of functions *(except return)*:
  - symmetric sequential composition
  - variable assignment, "if-then-else", "case", "loop", etc.
- Additional operators:
  - communication: rendezvous with value communication
  - parallel composition: "par"
  - gate hiding: "hide"
  - nondeterministic choice: "select"
  - "disrupt", etc.
- Static semantics constraints
  - variable initialization
  - typed channels (with polymorphism and "any" type)

LOTOS style
(see next slide)

Example of LOTOS process


```
type option is none, some (x: Nat) end type
channel option_channel is (o: Option) end channel
channel nat_channel is (n: Nat) end channel
```


```
process FILTER [GET: option_channel, PUT: nat_channel] (b: Nat) is
   var opt: Option in
      loop L in
         GET (?opt) ;
         case opt var x: Nat in
            none                    -> null
          | some (x) where x > b    -> PUT (x)
         end case
      end loop
   end var
end process
```

Rendezvous in LNT

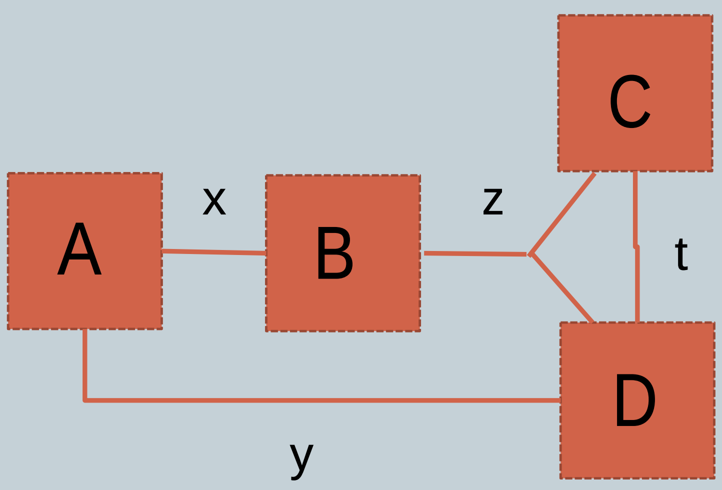
- Similar to LOTOS rendezvous, with extensions
- Features kept from LOTOS:
  - multiple offers exchanged during the same rendezvous
  - arbitrary combination of inputs/outputs
    G !1 ?X:NAT !true
  - value matching
    G !V_1 || G !V_2
  - value generation / constraint solving
    G ?X_1:S_1 [V_1] || G ?X_2:S_2 [V_2]
- New features in LNT
  - pattern matching in offers (richer patterns)
  - polymorphic gate typing (channels)

Sequential composition revisited

- In CCS, CSP, LOTOS, sequential composition is asymmetric (‘action-prefix’ operator)
  - syntax is G O_1, ..., O_n [V_0] ; B_0
  - left-hand side: gate, offers, optional guard
  - right-hand side: behaviour expression
- Drawbacks:
  - this is different from all classical algorithmic languages
  - one cannot write (B_1 [] B_2) ; B_3 nor (B_1 || B_2) ; B_3
  - action prefix makes sub-term sharing difficult (B_3 duplicated)
  - a symmetric operator is needed too: ‘exit’ and ‘>> accept’
  - ‘>>’ introduces a τ-transition (increases LTS size and no neutral element for sequential composition)
  - flow of variables becomes ugly: complexifies the syntax with ‘accept’ and *func* clauses
- In LNT: one single symmetric operator (noted ‘;’)

Parallel composition revisited

- Forget about binary parallel operators
- Think n-ary! Think graphically!
- Easy mapping from box diagrams to LNT




```
par
    x, y -> A
||
    x, z -> B
||
    z, t -> C
||
    y, z, t -> D
end par
```

Quick translation guide from LOTOS to LNT

Translation guidelines (1/6)

- Operator $\mathbf{stop}$
  - translates to ‘stop’ as well in LNT
  - there are much less stop’s in real programs than in tutorials!
- Operator $\mathbf{i} \;;\; B_0$
  - translates to ‘i ; B_0’ as well in LNT
  - key difference: (i ; B_0) in LOTOS and (i) ; (B_0) in LNT
- Operator $G\ O_1, \ldots\ O_n\ [[V_0]]\ ;\ B_0$
  - translates to G (O_1, ..., O_n) where V_0 ; B_0 -- where V_0 is optional
  - !V translates to V -- keeping ! is possible but not advised
  - ?X:S translates to ?X -- X must be declared before with ‘var’

Translation guidelines (2/6)

- Operator $B_1 \ [] \ B_2$
  - translates to 'select B_1 [] B_2 end select'
  - if more than 2 branches B_i, group them in the same 'select'

- Operator $B_1 \ op \ B_2$ with

$$
\begin{array}{lcl}
op & \equiv & || \\
 & | & ||| \\
 & | & |[G_0, \ldots \ G_n]|
\end{array}
$$

  - translates to 'par ... end par'
  - if only two operands:
    B_1 ||| B_2 translates to 'par B_1 || B_2 end par' and
    B_1 |[G_1, ..., G_n]| B_2 to 'par G_1, ..., G_n in B_1 || B_2 end par'
  - if more than two operands B_i, draw the connection network to propose an readable solution, avoiding useless nested par's

Translation guidelines (3/6)

- Operator $\textbf{hide}\ G_0, \ldots\ G_n\ \textbf{in}\ B_0$
  - translates to 'hide G_0:C_0, …, G_n:C_n in B_0 end hide'
  - gate declarations must be typed with channels

- Operator $\texttt{[}V_0\texttt{] ->}\ B_0$
  - translates to 'if V_0 then B_0 else stop end if'
  - 'else stop' must be present!
  - when an 'else' is missing, it is replaced with 'else null' to be compatible with classical sequential languages; but here, we want guarded commands and 'else null' would not be correct
  - usually, there are several [V_i] -> B_i as branches of a [] choice: if the V_i are exclusive and exhaustive, 'else stop' not needed

Translation guidelines (4/6)

- Operator $\mathbf{let}\ \widehat{X_0}:S_0=V_0, \ldots\ \widehat{X_n}:S_n=V_n\ \mathbf{in}\ B_0$
  - translates to: X_0 := V_0; ... ; X_n := V_n ; B_0
  - variables X_0, ..., X_n must have be declared before using 'var'
- Operator $\mathbf{choice}\ \widehat{X_0}:S_0, \ldots\ \widehat{X_n}:S_n\ [\,]\ B_0$
  - translates to: X_0 := any S_0; ... ; X_n := any S_n ; B_0
  - variables X_0, ..., X_n must have be declared before using 'var'
- Operator $B_1\ [> B_2$
  - translates to: disrupt B_1 by B_2 end disrupt

Translation guidelines (5/6)

- Operator $\textbf{exit}\ (R_1, \ldots\ R_n)$
  - translates to nothing (continuations are implicit in LNT) or to 'null' (if necessary to have an explicitly empty branch, for instance in a `case')
  - exit (V) should translate into some 'X := V'
  - exit (any S) should translate into some 'X := any S'
  - 'exit' and '>>' operators must be translated together to assign the right variables X

- Operator $B_1 \gg \textbf{accept}\ \widehat{X_1}\!:\!S_1, \ldots\ \widehat{X_n}\!:\!S_n\ \textbf{in}\ B_2$
  - translates to 'B_1 ; B_2' (or to 'B_1 ; i ; B_2' if one wishes to preserve the τ-transition created by '>>' in LOTOS)

Translation guidelines (6/6)

- Process call $P\ [G_1, \ldots\ G_n]\ (V_1, \ldots\ V_m)$

where

$$\textbf{process}\ P\ [G_1, \ldots\ G_m]\ (\widehat{X_1}{:}S_1, \ldots\ \widehat{X_n}{:}S_n)\ :\ \mathit{func}\ :=$$
$$B$$
$$\textbf{where}\ \mathit{block}_1, \ldots\ \mathit{block}_p$$
$$\textbf{endproc}$$

- many LOTOS processes are just there to encode iteration: replace these auxiliary processes with loops (possibly 'while' or 'for' loops)
- do not forget channels when declaring gates
- functionality *func* was related to sequential composition; if it is 'noexit' or 'exit' (without arg.) it does not need to be translated
- but functionality 'exit (S_0, ..., S_n)' usually requires to add a list of 'out' variables X_0:S_0, ..., X_n:S_n to process P

A few more details

Checking of semantic constraints

- Semantic checks performed on LNT code
  - Correct declaration (variables, gates)
  - Correct initialization (variables / parameters)
  - Non-ambiguous overloading
  - Breaks inside matching loops
  - Path constraints (e.g., presence of a return)
  - Parameters usage

- Semantic checks performed on LOTOS and C code
  - Type constraints (expressions and gates)
  - Availability of used types, functions, and processes
  - Exhaustiveness of case statements
  - Availability of external code (LOTOS, C)
  - Range/overflow checks for numbers