Reconciling Concurrency Theory with Other Branches of Computer Science

Hubert Garavel
Inria Grenoble – LIG
and Saarland University (part-time)

http://convecs.inria.fr
Concurrency theory in 2014

- Scientifically relevant, but difficult to defend
  - a rather mathematical branch of computer science
  - economical impact difficult to assess

- Argument #1
  - distributed computing is everywhere: from microarchitectures to the cloud
  - concurrency theory helps to correctly design and verify complex systems

- Argument #2
  - one lacks good languages to program parallel machines
  - concurrency theory studies languages with native parallel composition

- Yet:
  - students and engineers find process calculi difficult ("steep learning curve")
  - academic colleagues do not spontaneously adopt process calculi
Outline

- LNT: a born-again process calculus
- Upward encodings
- Expressiveness / Convenience
- Conclusion
LNT: a born-again process calculus
Action prefix  (1/2)

- A key operator of many process calculi:
  \[ a . P \mid a !x . P \mid a ?x . P \] with a action, P process, x variable

- Advantages:
  - well accepted by (most of) the concurrency theory community
  - simple syntax
  - simple SOS rules
  - convenient for proofs

- Drawback #1: non-standard wrt other programming languages
  - action prefix is asymmetric: \( a . P \) action a followed by a process P
  - everywhere else: symmetric sequential composition \( P ; P' \) process P followed by another process P'
  - students always tend to write symmetric sequential composition by default
Action prefix  (2/2)

- **Drawback #2:** incompatible with regular expressions
  - computer scientists know regular expressions (command shells, text editors)
  - they naturally tend to write regular expressions, rather than prefix terms

- **Drawback #3:** no "loop" operator
  - one is forced to use recursion and introduce extra processes
  - many proposals for introducing loops, but few implementations (if any)

- **Drawback #4:** prohibits control-flow sharing
  - action prefix forces to write trees and prohibits DAGs
  - Ex1:  \((a . c . \text{nil} + b . c . \text{nil})\) rather than \((a+b) . c . \text{nil}\)
  - Ex2:  \(\text{if } x \text{ then } (a . c . \text{nil}) \text{ else } (b . c . \text{nil})\) rather than \((\text{if } x \text{ then } a \text{ else } b) . c . \text{nil}\)
  - to avoid such undesirable unfoldings, one must introduce auxiliary processes
  - but this is poorly readable control flow ("goto"-like programming) and obscures the data flow (requires value parameters to be passed)
Attempt #1: LOTOS, CSP

- Action prefix was recognized to be insufficient as soon as 1985
- Idea: keep action prefix, add symmetric sequential composition
  - noted ">>" in LOTOS and ";" in CSP
- Many drawbacks:
  - two operators for almost the same purpose
    Ex (LOTOS): \( a ; b ; \text{exit} >> c ; d ; \text{stop} \)
  - each sequential composition `>>` creates a \( \tau \)-transition in the LTS
  - no neutral element for sequential composition (modulo strong bisimulation)
  - sub-term sharing for control flow is possible but heavy
    \( (a ; \text{exit} [] b ; \text{exit}) >> c; \text{stop} \)
  - In CSP, the values of variables do not move across sequential composition
    \( (?x : T -> \text{SKIP}) ; (x -> \text{STOP}) \) \( \) the left \( x \) remains local to \( (?x : T -> \text{SKIP}) \)
  - In LOTOS, the values of variables may move across sequential composition
    \( (\text{Recv} ?x:T; \text{exit} (x)) >> \text{accept} x:T \text{ in Send } !x; \text{stop} \) \( \) ok, but awfully complex
Attempt #2: ACP & Co (PSF, µCRL, mCRL2)

- Idea: discard action prefix; use symmetric sequential composition

- Advantages (in absence of value passing)
  - simplicity — and no creation of extra $\tau$-transitions
  - allows control-flow sharing
  - subsumes regular expressions (and even context-free grammars)

- Drawbacks (all related to value passing)
  - Input?x:Int ; Output !x ; exit cannot be written this way (i.e., as in LOTOS)
    it must be written $\Sigma (x:\text{Int}, \text{Input}(x) . \text{Output}(x))$ => no notation for input
  - the value of $x$ is not chosen during the input, but before (in the sum operator)
  - ambiguous: no dedicated syntax to distinguish between inputs and outputs
    $\Sigma (x:\text{Int}, a(x))$ can mean either $a?x:\text{Int} ; exit$ or choice $x:\text{Int} [] a !x ; exit$
  - certain forms of control-flow sharing cannot be expressed in these languages
    Ex: $(a ?x [] \tau ; b ?x) ; c !x ...$

where should the sum operator for "$b ?x$" be put?
Early conclusions

ACTION PREFIX IS THE ROOT OF ALL EVIL

- CCS, CSP, LOTOS are not optimal for describing complex systems
- ACP & Co. do slightly better, but do not solve all issues
- A better language (named "LNT") has to be designed

DECISION 1 for LNT:
- get rid of action prefix
- use ACP-style sequential composition

Next step: find a proper solution for value-passing issues
- must be intuitive for mainstream software engineers
- thus, necessarily different from both CCS/CSP/LOTOS and ACP & Co.
Control-flow and data-flow sharing

As mentioned before, control-flow sharing is intuitive and suitable

- Ex1: \((A [] B) ; C\)  \textit{nondeterministic choice}
- Ex2: \((\textit{if } x \textit{ then } A \textit{ else } B) ; C\)  \textit{deterministic choice}
- Ex3: \((\textit{case } x \textit{ in } a \rightarrow A \textit{ | } b \rightarrow B) ; C\)  \textit{deterministic choice}

The values of variables should implicitly move across ";" operators

- Ex4: \((A ?x [] B ?x) ; C !x \ldots\)
- Ex5: \((\textit{if } c \textit{ then } A ?x \textit{ else } x := 0) ; B !x \ldots\)

In most process calculi, variables are write-once

- they are so-called "dynamic constants"
- simple syntax: declaration and initialization of variables are bound together
- simple semantics: [value/variable] substitutions are sufficient

But dynamic constants are not mainstream in computer languages

- they isolate process calculi from the crowd of software developers
Introducing "true" variables

**DECISION 2 FOR LNT:**

- ordinary (i.e., "write-many") variables are suitable
- both in the data part (functions) and in the behavior part (processes)
- variable *declarations* and variable *modifications* need to be separated
- successive assignments to the same variable are permitted

**Variable declarations**

- `var X : T in ... end var`

**Variable modifications**

- `X := E` *assignment*
- `G ?X where E (X)` *input with (optional) predicate*
- `X := any T where E (X)` *nondeterministic assignment with predicate*
- calls to functions and processes (Ada-like "in", "out", and "in out" parameters)
Uninitialized variables  (1/2)

Problem: certain syntactically correct terms have no clear meaning
- Ex:  ( A ?x [] B ?y ) ; C !x+y
- but this term becomes meaningful if prefixed with  x := 0 ; y := 0

Whether a term has a meaning or not is undecidable  \( \approx \) halting

Solution #1: reading uninitialized variables has undefined effects
- usual solution in imperative languages (as in C, etc.)
- unacceptable if a formal semantics is sought

Solution #2: initialize all variables implicitly when they are declared
- e.g. set integers to zero, Booleans to false (as in Eiffel)
- allows formal semantics but hides user mistakes

Solution #3: give uninitialized variables  nondeterministic values
- tricky: implicit summation operator by reading an uninitialized variable
- allows formal semantics but hides user mistakes
Uninitialized variables (2/2)

- Solution #4: add restrictions to reject "dubious" programs

- Either using syntactic restrictions:
  - CCS: **asymmetric action prefix** is just a means to avoid \((a \ ?x + b \ ?y) . c \ !x+y\)
  - ACP: **output-only syntax for actions** is another means for the same issue
  - syntactic restrictions are very primitive defense means; better solutions exist

- Or using static semantics restrictions:
  - standard means to rule out syntactically correct, yet problematic programs
  - process calculi neglect static semantics and try to do everything using syntax

- **DECISION 3 FOR LNT: static semantics constraints on initializations**
  - reject programs in which variables are not provably set before used
  - sufficient conditions based on static data-flow analysis
  - inspired by the Hermes (IBM) and Java (Sun) languages
  - well-accepted by programmers, catches many mistakes
"Context-free" recursion

- Symmetric sequential composition allows context-free recursion
  - Example: `process P = null [] ( A ; P ; B )`
  - (note that action prefix syntactically prohibits this)

- Assessment:
  - this recursion is not so useful in practice
  - the same behaviour can be easily described using regular processes with value parameters

- DECISION 4 for LNT: static semantic restrictions on recursion
  - LNT processes: only tail-recursion is allowed
    - note: non-tail recursion could yet be eliminated automatically (e.g. µCRL)
  - LNT functions: no restriction on the use of recursion
Shared variables

Separation of declaration and assignment allows shared variables

Example: \texttt{var X:int in ( Input ?X || Input ?X ) ; Output !X}

(no note that this is impossible when variables are write-once)

Assessment

This could be an opportunity to combine message-passing and shared-variable paradigms in the same formal language

A nice semantics could probably be found for shared variables

For the moment, LNT remains in the message-passing framework

DECISION 5 for LNT: static semantic restrictions on shared variables

LNT parallel branches may inherit variables from their enclosing scope

In principle, all parallel branches can read all shared variables

If a branch writes a shared variable, the other branches can neither write nor read this variable (i.e., exclusive write access policy)
Dynamic semantics of LNT

  - written by Frédéric Lang (16 pages)

- For LNT functions:
  - state = memory store (mapping: variable $\rightarrow$ value)
  - LNT instructions define transitions between states (i.e., store updates)

- For LNT processes:
  - Labelled transition systems
  - LTS state = <process term, memory store>
  - SOS rules define transitions between LTS states
  - Sequential composition: ACP-like rules + store updates
  - Static semantics restrictions avoid complications in the dynamic semantics
Upward encodings
Encoding reg. exp. and ACP in LNT

- Regular expressions  ------------->  LNT
  \( \varepsilon \)  
  \( a \)  
  \( R_1 . R_2 \)  
  \( R_1 | R_2 \)  
  \( R^* \)  

  null  — but adds a tick √
  a  — but adds a tick √
  R1 ; R2
  select R1 [] R2 end select
  loop R end loop

- ACP  -------------->  LNT
  0  
  1  
  \( \Sigma (x : T, P(x)) \)  

  stop  
  null  
  var x:T in x := any T; P(x) end var

Parallel composition and renaming are orthogonal issues
Encoding CCS in LNT

- CCS  \[ \longrightarrow \]  LNT

  - nil \[ \rightarrow \] stop
  - a . P \[ \rightarrow \] a ; P
  - a !x . P \[ \rightarrow \] a (x) ; P
  - a ?x:T . P \[ \rightarrow \] var x:T in a (?x) ; P end var
  - P1 + P2 \[ \rightarrow \] select P1 [] P2 end select

- Other CCS operators
  - recursion: translates to either a loop operator or an LNT process call
  - CCS "complement" gates, parallel and restriction are orthogonal issues
Encoding LOTOS in LNT

For those LOTOS operators that also exist in CCS:
- apply the same rules as for the CCS to LNT translation
- but LOTOS has additional operators that do not exist in CCS

- LOTOS
  \[\begin{align*}
  &G \ ?x:T \ [V] \ in \ P \\
  &\text{let} \ x:T = V \ in \ P \\
  &\text{choice} \ x:T \ [\ ] \ P \\
  &\text{exit} \\
  &\text{exit} \ (V1, ..., Vn) \\
  &P1 >> P2 \\
  &P1 >> \text{accept} \ x:T \ in \ P2
  \end{align*}\]

  \[\begin{align*}
  &\text{var} \ x:T \ in \ G \ (?x) \ \text{where} \ V \ ; \ P \ \text{end var} \\
  &\text{var} \ x:T \ in \ x := V \ ; \ P \ \text{end var} \\
  &\text{var} \ x:T \ in \ x := \text{any} \ T \ ; \ P \ \text{end var} \\
  &\text{null} \\
  &\text{null} \\
  &P1 ; \tau ; P2 \\
  &P1 ; \tau ; P2 \quad (\text{where} \ P1 \ \text{assigns} \ x)
  \end{align*}\]
The quest for a unifying framework for process calculi

- The usual approach
  - search for a "core" calculus of very primitive elements
  - try to express classical process calculi using this "core" calculus
  - the core calculus is "low level", whereas the process calculi are "high level"

- LNT: a different approach
  - translate classical process calculi to LNT
  - the classical process calculi are "low level", whereas LNT is "high level"
  - the translations to LNT are straightforward (i.e., "syntactical" substitutions)
  - the classical process calculi appear as a "subset" or a particular "specification style" of LNT, which is more general
Expressiveness / Convenience
Reusing algorithmic control structures

Once symmetric sequential composition and "standard" value passing rules are adopted, all the usual constructs of algorithmic programming languages come "for free"

In LNT, 70% of constructs look familiar (Ada-like syntax):

- if-then-else (with elsif)
- case with pattern matching
- while ... loop, for ... loop, forever loop with break
- functions with return statement
- LNT functions and processes have many constructs in common

Additional process constructs (coming from concurrency theory):

- nondeterministic assignment: X := any T where P (X)
- nondeterministic choice: select ... [] ... [] ... end select
- parallel composition: par ... || ... || ... end par
- hiding: hide ... end hide
More flexible specification styles

- LNT favors alternatives to the traditional "condition/action" style
- A recent example:

```plaintext
select
    L := {}
    L := {0, 1}
    L := {1, 0, 2}
    ...
end select;
SEND (L);
while L != {} loop
    X := X - head (L);
    L := tail (L)
end loop
```

nondeterministic choice used to produce a finite set of values among a potentially infinite domain

(there are no input/output actions in the branches of this select statement)

statically unbounded number of assignments
Challenge 1: Guarded commands

- Proposed by Dijkstra — used, e.g., in the PRISM model checker
- LNT can express guarded commands naturally and concisely:

```plaintext
process GuardedCommands [G1, G2, ... Gn : void] is
    var X1, X2, ... Xn : int in
    X1 := 0 ; X2 := 0 ; ... ; Xn := 0
    loop
        select
            only if X1 < 9 then G1 ; X1 := X1+1 end if
            [] ... []
            only if Xn < 9 then Gn ; Xn := Xn+1 end if
        end select
    end loop
end var
end process
```

Using traditional process calculi:
- 1 recursive process having \( n \) parameters
- \( n \) recursive process calls
- \( n^2 \) parameters passed (most of which unchanged)
- LNT = linear code size, others = quadratic code size
Challenge 2: DAG control patterns

- LNT can directly express DAG-like control patterns:
  - e.g., choice-DAGs: \((P1 \parallel P2) ; (Q1 \parallel Q2) ; (R1 \parallel R2)\)
  - but also if-DAGs, case-DAGs, etc.

```plaintext
process DAG [Input, Output : IntChannel] (X1, ..., Xn : Int) is
  if X1 = 0 then Input (?X1) end if ;
  if X2 = 0 then Input (?X2) end if ;
  ...
  if Xn = 0 then Input (?Xn) end if ;
  Output (combination (X1, X2, ..., Xn))
end process
```

Using traditional process calculi:
- \(n\) processes having \(n\) parameters each
- \(n^2\) parameters passed
- LNT = linear code size, others = quadratic code size
- tedious and error prone
Challenge 3: Map-Reduce

- Given \( n \) inputs \( X_1, X_2, ..., X_n \), compute \( g (f_1 (X_1), f_2 (X_2), ..., f_n (X_n)) \)
- Each computation \( Y_i = f_i (X_i) \) is given to one parallel processor

```plaintext
var X_1, X_2, ..., X_n : S,
    Y_1, Y_2, ..., Y_n : T
in
Input (?X_1, ?X_2, ..., ?X_n);
par
    Y_1 := f_1 (X_1)
    || Y_2 := f_2 (X_2)
    || ...
    || Y_n := f_n (X_n)
end par;
Output (g (Y_1, Y_2, ..., Y_n))
end var

Input ?X_1, X_2, ..., X_n : S ;
(  
    exit (f_1 (X_1), any T, ..., any T)
    || exit (any T, f_2 (X_2), ... any T)
    || ...
    || exit (any T, any T, ..., f_n (X_n))
)  
>> accept Y_1, Y_2, ..., Y_n : T in
     Output (g (Y_1, Y_2, ..., Y_n))
end var
```

LNT = linear code size, LOTOS = quadratic code size, **not compositional**
Conclusions
Revisiting classical process calculi

- Classical process calculi are good, yet not optimal
  - they are difficult to learn and to master
  - they face certain problems when scaling to large, complex systems (prohibition of control-flow sharing, quadratic explosion of code size, etc.)
  - a better tradeoff between convenience and semantic simplicity is possible

- A critical assessment of action prefix and write-once variables
  - forcing write-once variables is simple, but overly restrictive and clumsy
  - CCS action prefix is a "trick" to syntactically forbid write-many variables
  - ACP output-only syntax is another trick to also forbid write-many variables

- Why are (most) process calculi designed like this?
  - need for having a formal semantics (forbid uninitialized variables)
  - individual preferences for functional languages, algebras, etc.
  - ignores the difference between syntax checks and static semantics checks
  - process calculi came too early: Hermes (1986-92) and Java (95) arrived later
LNT: an alternative approach

Key concepts:
- remove action prefix
- add sequential symmetric composition
- separate variable declaration and modification
- allow write-many variables
- static semantics: use data flow analysis to reject dubious programs
- dynamic semantics: extend LTS states with memory stores

Benefits:
- generalizes regular expressions and the usual calculi: ACP, CCS, CSP, LOTOS
- generalizes sequential imperative languages
- better convenience than the usual calculi (dags, map-reduce, etc.)
- supports action refinement (replacement of an action by a process)
Design and implementation of LNT

First attempt: 1993-2000
- push ideas in the definition of E-LOTOS (ISO standard 15435:2001)

Second attempt: 1998-2008
- definition of LOTOS NT, a simplified version of E-LOTOS
- direct implementation: the TRAIAN compiler (data types only $\rightarrow$ C)
  Mihaela Sighireanu's PhD thesis

Third attempt: 2005-now
- indirect implementation: LNT $\rightarrow$ LOTOS (much harder than LOTOS $\rightarrow$ LNT)
- LNT2LOTOS translator (initially funded by Bull)
  Frédéric Lang: translation of LNT types and functions
  Wendelin Serwe: translation of LNT processes
  D. Champelovier, X. Clerc, etc.: implementation of the translator
- reuse of the LOTOS compilers and verification tools present in CADP

On the long run: resume direct implementation LNT $\rightarrow$ C
Feedback about LNT

- LNT is taught to engineering students
  - LNT is much easier and faster to learn than LOTOS
  - LNT builds on prior knowledge: regular expressions, programming languages students don't have to forget what they already learnt in programming courses
    - they can focus on concurrency theory concepts (choice, parallel, hide, etc.)
  - because LNT is intuitive, students tend to jump writing specifications without reading the formal semantics (a very questionable advantage!)

- LNT is used to model real-life applications
  - since 2010, LNT has entirely replaced LOTOS in our research team
  - a growing list of case-studies: ATVA'13, FMICS'13, FORTE'13, FORTE'14, IFM'13, ISSE'13, SAC'14, TACAS'13, SCICO journal (2013 and 2014)
  - STMicroelectronics: LNT enabled the development of hardware models that were too large to be realistically described in LOTOS