Revisiting Sequential Composition in Process Calculi

Hubert Garavel
Inria Grenoble – LIG
http://convecs.inria.fr
Outline

- Overview of mainstream process calculi
- Sequential composition in process calculi — Rationale for the design of LNT
- Upward encodings
- Expressiveness / Convenience
- Conclusion
Overview of mainstream process calculi
Process calculi

- Definition #1
  - mathematical models for the study of concurrency
  - mostly asynchronous concurrency
  - some approaches towards synchronous concurrency: SCCS, Esterel

- Definition #2
  - specification languages with an explicit parallel composition operator
  - and a formal semantics

- A complicated history
  - many different process calculi
  - partly justified by technical differences between them
  - but also due to different schools that were reluctant to merge
  - international standardization efforts did not manage to bring unification
A tentative landscape

Oxford track

CSP (1975)
Th. CSP (1984)

Edinburgh track

CCS (1981)
CCS (1989)
π-calculus

ISO track

LOTOS (1989)

Grenoble track

LOTOS NT

Amsterdam track

ACP (1984)
PSF
μCRL
mCRL2

FDR2
bigraphs

FDR3

π-calculus

E-LOTOS (2001)

VPL

LOTOS v1 (2006)
...

LNT v6 (2016)
Sequential composition in process calculi — Rationale for the design of LNT
Action prefix  (1/2)

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

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

- Drawback #1: non-standard wrt other programming languages
  - action prefix is asymmetric: \( a . P \) action followed by a process
  - everywhere else: symmetric sequential composition
    \[ P ; P' \] process followed by another process
  - 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: \((a \cdot c \cdot \text{nil} + b \cdot c \cdot \text{nil})\) rather than \((a+b) \cdot c \cdot \text{nil}\)
  - Ex2: `if x then (a \cdot c \cdot \text{nil}) else (b \cdot c \cdot \text{nil})` rather than `(if x then a else b) \cdot c \cdot \text{nil}`
  - only solution to avoid undesirable unfoldings: define auxiliary processes
  - but poorly readable control flow ("goto"-like programming) obscures the data flow (requires value parameters to be passed)
Attempt #1: LOTOS, CSP

- Idea: keep action prefix, add symmetric sequential composition
  - noted ">>" in LOTOS and ";;" in CSP
  - action prefix recognized to be insufficient as soon as 1985

- Many drawbacks:
  - two operators for almost the same purpose
    a ; b ; \textbf{exit} >> c ; d ; \textbf{stop}
  - each sequential composition creates a τ-transition in the LTS
  - no neutral element for sequential composition (modulo strong bisimulation)
  - sub-term sharing is possible but heavy
    (a ; \textbf{exit} []; b ; \textbf{exit}) >> c; \textbf{stop}
  - In CSP, the values of variables do not move across sequential composition
    (?x : T -> SKIP) ; (x -> STOP)  
    the left x remains local to (?x : T -> SKIP)
  - In LOTOS, the values of variables may move across sequential composition
    \textbf{(let} x:T = 1 \textbf{in} \textbf{exit} (x)) >> \textbf{accept} x:T \textbf{in} Output !x; \textbf{stop}
    but awfully complex
Attempt #2: ACP & Co (PSF, µCRL, mCRL2)

- Idea: discard action prefix; use symmetric sequential composition

- Advantages (without 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
    it must be written \( \Sigma (x:\text{Int}, \text{Input} (x) . \text{Output} (x)) \)
  - \( x \) is not assigned 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 \( a?x:\text{Int} ; \text{exit} \) or \( \text{choice} x:\text{Int} [] a !x ; \text{exit} \)
  - certain suitable behaviours cannot be expressed
    Ex1: \( (a ; b ?x + c ; \text{stop}) ; d !x \)
    Ex2: \( x := 0 ; y := 0 ; (a ?x + b ?y) ; c !x+y \)
Early conclusions

- ACTION PREFIX IS THE ROOT OF ALL EVIL
- CCS, CSP, LOTOS are not optimal languages
- ACP & Co. do slightly better, but not solve all issues
- A better language (named "LNT") needs 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 ACP & Co.
Control-flow and data-flow sharing

- Control-flow sharing is intuitive and suitable
  - Ex1: \((A \mid B) ; C\)
  - Ex2: \((\text{if } x \text{ then } A \text{ else } B) ; C\)
  - Ex3: \((\text{case } x \text{ in } a \rightarrow A | b \rightarrow B) ; C\)

- The values of variables should implicitly move across ";" operators
  - Ex4: \((A ?x \mid B ?x) ; C !x \ldots\)
  - Ex5: \((\text{if } c \text{ then } A ?x \text{ else } x := 0) ; B !x \ldots\)

- In most process calculi, variables are write-once
  - they are so-called "dynamic constants"
  - simple syntax: declaration and assignments are bound together
  - simple semantics: [value/variable] substitutions are enough

- 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 ("in", "out", and "in out" parameters)
Uninitialized variables (1/2)

- Problem: certain syntactically correct terms have no 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 (= 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 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 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: \texttt{process P [A, B] = null \{ A ; P [A, B] ; B \}}
  - (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 be eliminated automatically (e.g. \( \mu \text{CRL} \))
  - LNT functions: no restriction on the use of recursion
Shared variables

- Separation of declaration and assignment allows shared variables
  - Example: `var X:int in ( Input ?X || Input ?X ) ; Output !X`
  - (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
Dynamic semantics of LNT

  - Written by Frédéric Lang (16 pages)

- For LNT functions:
  - state = memory store (mapping: variable → 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
The quest for a unifying framework for process calculi

- The usual approach
  - search for a "core" calculus of very primitive elements
  - encode the various calculi using this "core" calculus
  - the core calculus is low level, the process calculi are high level

- LNT: a different approach
  - translate process calculi to LNT
  - the process calculi are low level, LNT is high level
  - the translations to LNT are straightforward
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*  $\sqrt{}$  
  $a$  —  *but adds a tick*  $\sqrt{}$  
  $R_1 ; R_2$  
  $select$  $R_1$  $[$  $R_2$  $]$  $end$  $select$  
  $loop$  $R$  $end$  $loop$

- **ACP**  -------------->  **LNT**
  
  $0$  
  $1$  
  $\Sigma (x : T, P(x))$  
  
  $stop$  
  $null$  
  $var$  $x : T$  $in$  $x :=$  $any$  $T ; P (x)$  $end$  $var$  
  $or$  $var$  $x : T$  $in$  $G (?x) ; P (x)$  $end$  $var$
Encoding CCS in LNT

- CCS
  - nil
  - a . P
  - a !x . P
  - a ?x: T . P
  - P1 + P2

  -------------------------> LNT
  - stop
  - a ; P
  - a (x) ; P
  - var x: T in a (?x) ; P end var
  - select R1 [ ] R2 end select

- Other CCS operators
  - recursion: translates to either a `loop` operator or an LNT process call
  - "complement" gates: out of scope
Encoding LOTOS / CSP in LNT

- Common part with CCS to LNT translation
  - plus a few additional operators

LOTOS

G ?x:T [V] ; P
let x:T = V in P
choice x:T [ ] P
exit
P1 >> P2
P1 >> accept x:T in P2

LNT

var x:T in G (?x) where V ; P end var
var x:T in x := V ; P end var
var x:T in x := any T ; P end var
null
P1 ; τ ; P2
P1 (which assigns x) ; τ ; P2
Expressiveness / Convenience
Reusing algorithmic constructs

- Once symmetric sequential composition is 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

- Additional constructs (originating from concurrency theory):
  - nondeterministic assignment: \( X := \text{any } T \text{ where } P (X) \)
  - nondeterministic choice: \( \text{select} ... [] ... [] ... \text{end select} \)
  - parallel composition: \( \text{par} ... | | ... | | ... \text{end par} \)
  - hiding: \( \text{hide} ... \text{end hide} \)

- Functions and processes have many constructs in common
More flexible specification styles

LNT favors alternatives to the traditional "condition/action" style

```
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

```
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 $X_1, X_2, \ldots, X_n$, compute $g (f_1 (X_1), f_2 (X_2), \ldots, f_n (X_n))$
- Each computation $Y_i = f_i (X_i)$ is given to one parallel processor

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

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

LNT = linear code size, LOTOS = quadratic code size, non compositional
Conclusion
Questioning action prefix

- For "basic" process calculi
  - action prefix has little justification and seems inferior to ACP

- For value-passing process calculi
  - action prefix is mostly a "trick" to syntactically forbid write-many variables and force the use of write-once variables
  - simple, but overly restrictive and clumsy
  - ignores the difference between syntax checks and static semantics checks

- Why is (most of) concurrency theory built on this?
  - need for having a formal semantics (forbid uninitialized variables)
  - individual preferences for functional languages, algebras, etc.
  - process calculi came too early: Hermes and Java arrived later
  - a few forerunner languages tried to get rid of action prefix: ACPε, ACP₈, ACBS&, Extended-LOTOS, E-LOTOS, OCCAM
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)
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.)
  - LNT is intuitive, students tend to jump writing specifications without reading the formal semantics impossible with traditional process calculi, but a questionable advantage

- LNT is used to model real-life applications
  - since 2010, LNT has entirely replaced LOTOS in our team
  - a growing list of case-studies: ATVA'13, CBSE’14-15, EICS'14-15, FMICS'13-14, FORTE'13-14, ICEFM’14, IFM'13, ISSE'13, PDP’15, MARS'15, SAC'14, TACAS'13-15-16, SCICO’13-14, VMCAI’15
  - STMicroelectronics: "LNT enabled us to analyze systems too large to be realistically described in LOTOS"
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 → C)
    Mihaela Sighireanu's PhD thesis

- Third attempt: 2005-now
  - indirect implementation: LNT → LOTOS (much harder than LOTOS → LNT)
  - LNT2LOTOS translator (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 → C