# Revisiting sequential composition in process calculi

Hubert Garavel

# Revisiting Sequential Composition in Process Calculi

Hubert Garavel[a,b,c,d]

[a] *INRIA, Grenoble, France*[1]
[b] *Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France*
[c] *CNRS, LIG, F-38000 Grenoble, France*
[d] *Saarland University, Saarbrücken, Germany*

**Abstract**

The article reviews the various ways sequential composition is defined in traditional process calculi, and shows that such definitions are not optimal, thus limiting the dissemination of concurrency theory ideas among computer scientists. An alternative approach is proposed, based on a symmetric binary operator and write-many variables. This approach, which generalizes traditional process calculi, has been used to define the new LNT language implemented in the CADP toolbox. Feedback gained from university lectures and real-life case studies shows a high acceptance by computer-science students and industry engineers.

*Keywords:* ACP, concurrency theory, CCS, CSP, E-LOTOS, formal specification, formal semantics, LOTOS, LNT, modelling language, muCRL, OCCAM, process algebra, process calculus, programming language, PSF, sequential composition, software engineering, specification language

## 1. Introduction

Process calculi (or process algebras) are a corner stone of concurrency theory and exist under many variants, among which CCS [1, 2], CSP [3, 4, 5], MEIJE [6], ACP[2] [7, 8, 9], LOTOS [10, 11], PSF [12, 13, 14], $\mu$CRL [15, 16], and mCRL2 [17], to name only a few. From a theoretical point of view, process calculi are *mathematical models* for the study of concurrency. To this aim, they are expected to be of manageable complexity, i.e., have a minimal number of constructs or, at least, a small set of core constructs to which other constructs can be translated.

In practice, however, this definition is too narrow and process calculi clearly have another role. They are often used as *modelling languages* to formally describe the behavior of complex concurrent systems, including telecommunication

[1] E-mail: `hubert.garavel@inria.fr` – Web: `http://convecs.inria.fr`

[2] For simplicity we will systematically use the name ACP rather than its sub-algebra acronyms, BPA (*Basic Process Algebra*) and PA (*Process Algebra*), even if parallel composition and interprocess communication are not the prime focus of the present article.

protocols, distributed software, and hardware circuits. The intent of applying process calculi to real-life problems can be traced back at least to the mid-80s, when LOTOS was defined as an international standard to formally describe OSI protocols and services. Moreover, many compilers and software verification tools have been developed to implement process calculi, e.g., CWB [18] and CWB-NB [19] for CCS, FDR [20, 21] and PAT [22, 23] for CSP, CADP [24] for LOTOS, the mCRL tools [25, 26] for $\mu$CRL and mCRL2, etc. All in one, this is a clear indication that more is expected from process calculi than merely providing a theoretical framework for abstractly reasoning about concurrency.

Whether a given process calculus can conveniently play these two roles is very much an open question. For instance, Milner believed that CCS was meant for theory only and he proposed, to describe more concrete systems, another language named $\mathcal{M}$ [2, Chapter 8] based on shared variables and algorithmic programming constructs, together with a translation algorithm from $\mathcal{M}$ to CCS. Although there have been follow-ups to this idea, e.g., [27] [28] [5, Chapter 18], one may wonder if having two languages instead of one is a true benefit. Moreover, most of the aforementioned process calculi claim to be modelling languages for concurrent systems as well as formalisms supporting mathematical reasoning, therefore indicating a general trend towards a single-language approach.

To describe and analyze real systems, process calculi have genuine advantages: they provide built-in parallel composition operators, they are equipped with formal semantics, and they have compositionality properties (e.g., congruence results) for scaling to large component-based systems. Despite a growing number of success stories, the practical impact of process calculi is not as high as it should be: formal approaches are not widely used in industry and few recent languages for modelling or programming take inspiration from concurrency theory.

There are several reasons for this situation [29]: fragmentation (multiple, similar yet incompatible process calculi), lack of expressiveness, and lack of user-friendliness, all resulting in a steep learning curve that discourages potential users. The aforementioned dichotomy between mathematical models and modelling languages also contributed to creating a gap: the simplicity and elegance of CCS, reinforced by the algebraic approach of ACP, shifted the focus towards mathematical models, weakening the links with mainstream programming languages.

We believe it is important to disseminate concurrency theory results to a wider audience, and that process calculi can be an essential vector for this. It is high time to reconsider the dichotomy between process calculi and programming languages, and to come up with enhanced process calculi acceptable by professionals. As part of this agenda, we review the way sequential composition is handled in process calculi. This topic is often seen as a matter of secondary importance with respect to concurrency and mobility, but the present article reopens the case by considering sequential composition as a major subject that almost entirely shapes a process calculus and determines its compatibility with mainstream programming languages. The core of the issue is not so much *expressiveness* — most process calculi provide the same expressiveness whatever

the approach chosen for sequential composition — but *conciseness* (objective) and *convenience* (subjective).

The present article is organized as follows. Sec. 2 sets a few definitions that will be used throughout the article. Sec. 3 reviews and gives a critical evaluation of the sequential composition operators available in traditional process calculi. Sec. 4 proposes an enhanced definition of sequential composition, which has been integrated in LNT [30], the most recent specification language supported by the CADP toolbox [24]. Sec. 5 discusses the expressiveness of LNT and proposes encodings that map large fragments of former process calculi (CCS, ACP, LOTOS, and CSP) to LNT. Sec. 6 illustrates the advantages of LNT on several design patterns that cannot be concisely expressed using traditional process calculi. Finally, Sec. 7 gives concluding remarks and perspectives for future work.

## 2. Preliminary definitions

The present article assumes that the reader has some basic familiarity with the general concepts of process calculi, but no extensive knowledge is required. As the various process calculi use different vocabulary and notations to define their syntax and semantics, we introduce here common terminology and notations that will be used for all languages throughout the article.

In the following, terminal symbols (i.e., identifiers or constants) are noted in lower case, while non-terminal symbols (defined by BNF rules) are noted in upper case. If $y$ (resp. $Y$) is a symbol of a given category, then $y', y'', y_0, y_1, y_2$, etc. (resp. $Y', Y'', Y_0, Y_1, Y_2$, etc.) are also symbols of that same category.

We note $t$ a *data type*, $x$ a *variable*, and $v$ a *constant*. Variables and constants are typed and we note "$v \in t$" the fact that $v$ is a constant of type $t$. We note $V$ an *expression* (also: *data expression*, *value expression*, or *value term*), i.e., a syntactic term which is built using variables, constants, and function symbols and that computes a typed value.

We note $a$ an *action* (also: *atomic action* or *atom*), i.e., a communication event proposed by the system under study or its environment. Any action $a$ can be decomposed as a tuple $a = g\ o_1\ ...\ o_n$, where $g$ is a *gate* (also: *channel* or *communication port*) and $o_1\ ...\ o_n$ a possibly empty list of *offers* (also: *experiment offers* or *action parameters*). Each offer is either an emission (noted "!$V$") of some expression $V$, or a reception (noted "?$x$:$t$", or simply "?$x$") of some constant value to be stored in variable $x$ of type $t$. We note $gate(a)$ the gate of action $a$. There exists an *internal* (also: *invisible* or *hidden*) gate noted either "$\tau$" or "**i**", which must be used without offer (and is thus also considered to be an action).

We note $B$ a *behavior* (also: *behavior expression* or *process term*), i.e., a syntactic term built using actions and the various operators of the process calculi being considered. Most of our attention focuses on sequential composition operators, so that other kinds of operators (e.g., parallel composition, disruption, etc.) will only be mentioned when needed. We note "$[v_1/x_1 ... v_n/x_n]B$" the behavior obtained from $B$ by replacing all free occurrences of variables $x_1, ..., x_n$

by constants $v_1, ..., v_n$, respectively. We note $p$ a *process identifier* (also: *agent name*), i.e., a symbolic name that can be associated to a behavior (this is needed to define recursive behaviors).

## 3. Sequential composition in traditional process calculi

### *3.1. The "action prefix" operator and its drawbacks*

The most widespread means of expressing sequentiality in process calculi is the *action prefix* operator (also called *prefixing*), which is present in many process calculi. This operator will be noted here ".", as in CCS; it is also noted ":" in MEIJE, "$\rightarrow$" in CSP, and ";" in LOTOS. Its syntax is defined by the BNF rule "$B ::= a.B$" and its semantics by the SOS rule:

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

meaning that "$a.B$" first performs action $a$ and then behaves as $B$.

Action prefix is simple, suitable for proofs, and well accepted by (most of) the concurrency theory community. In practice, however, action prefix suffers from several drawbacks:

1. *It is non-standard with respect to mainstream programming languages.* Action prefix is *asymmetric* in the sense that its left-hand and right-hand sides require different syntactic objects (an action must come first, followed by a behavior) whereas sequential composition is *symmetric* in most programming languages (a statement is followed by another statement). Asymmetry proves to be confusing to those that learn process calculi, and students always tend to write symmetric sequential composition by default.
2. *It does not support loops directly.* With action prefix, the standard way of expressing iterative behaviors is to use recursion, by introducing extra process identifiers and inserting a recursive process call at each loop-back point. There have been attempts at defining iteration operators in the presence of action prefix (e.g., [5, p. 134]), but their usefulness is limited (the values computed at some iteration cannot be passed to the next iterations) and such proposals are usually not implemented in software tools.
3. *It is incompatible with regular expressions.* Quite often, the components of concurrent systems can be described as automata. Regular expressions are a standard means to express automata and most industry engineers are familiar with regular expressions, which are routinely used in text editors and script languages, for instance. Unfortunately, regular expressions cannot be easily expressed using action prefix, because it is asymmetric and a "**loop**" (Kleene star) operator is missing.
4. *It prohibits control-flow sharing.* Often in programming, a choice must be made between two computations $B_1$ and $B_2$, only one of which can

be executed, and afterwards a third computation $B_3$ has to be executed, whatever which branch $B_1$ or $B_2$ was taken before. In sequential programming, the choice between $B_1$ or $B_2$ usually depends on some deterministic condition $v$, so that the whole pattern can be written "**if** $v$ **then** $B_1$ **else** $B_2$ **end if** ; $B_3$". So doing, the computation $B_3$ is *shared*, i.e. written only once. Action prefix makes sharing of $B_3$ impossible because prefixing favors tree-like patterns [31] and prohibits dag[3]-like patterns. For instance, in CCS, it is syntactically illegal to write "$(a_1 + a_2).B_3$", where the "+" operator denotes nondeterministic choice; one must write instead "$(a_1.B_3) + (a_2.B_3)$" at the expense of duplicating $B_3$. Such undesirable unfolding can be avoided by defining an auxiliary process identifier $p = B_3$ in order to write "$(a_1.p) + (a_2.p)$" without duplicating $B_3$; however, this solution is questionable, because each call to $p$ is similar to a "**goto** $p$" and decreases the readability of the specification by transferring the control elsewhere.

5. *It prohibits data-flow sharing.* The syntax of action prefix also forbids terms of the form "$(g_1?x + g_2?x).B_3$", where the variable $x$ is received on either gate $g_1$ or gate $g_2$ and its value passed to $B_3$. Such data-flow sharing is standard in sequential programming, where the variables assigned in the branches of an "**if**" or "**case**" statement can be reused afterwards. Unfortunately, action prefix forces either the duplication of $B_3$ (as in "$(g_1?x.B3) + (g_2?x.B_3)$") or the introduction of an auxiliary process identifier $p$ (as in "$(g_1?x.p(x)) + (g_2?x.p(x))$") with the aforementioned drawbacks.
6. *It is not well-adapted to action refinement*, i.e., program transformations based on the replacement of individual actions by more complex behaviors. For such transformations, symmetric sequential composition is usually preferred, e.g. [32] [33, 34] [35, 36, 37].

### *3.2. The "enable" operator and its drawbacks*

The limitations of action prefix have been recognized for long — at least as early as the mid-80s. The usual remedy is to introducing a new sequential composition operator that complements action prefix and tries palliating its shortcomings. Such operator is called *enable* in LOTOS and is noted "`>>`" (it also exists in CSP, where it is called *sequential composition* and is noted ";"). The behavior "$B_1$ `>>` $B_2$" expresses that $B_1$ executes first and is followed by $B_2$, which only starts after $B_1$ is done. The enable operator is used together with another operator called *successful termination* (noted "**exit**" in LOTOS and "*SKIP*" in CSP), which differs from the *inaction* or *deadlock* operator (noted "**nil**" in CCS, "*STOP*" in CSP, and "**stop**" in LOTOS).

We consider here the LOTOS "`>>`" and "**exit**" operators, which are the most flexible ones, as they support the passing of values. We note "$\surd$" a

[3]Directed Acyclic Graph.

special *termination gate*[4] that is different from any other gate, including "$\tau$". The BNF rule defining the syntax of the "**exit**" operator is:

$$B ::= \mathbf{exit}(R_1, ..., R_n)$$

where $R_1, ..., R_n$ is a (possibly empty) list of *results*, each $R_i$ being a syntactic term that is either an expression $V_i$ or a clause "**any** $t_i$". We note "$v_i \in R_i$" the fact that expression $V_i$ evaluates to constant $v_i$ (if $R_i$ has the form $V_i$) or the fact that $v_i \in t_i$ (if $R_i$ has the form "**any** $t_i$"). The semantics of the "**exit**" operator is defined by the following SOS rule:

$$\frac{v_1 \in R_1 \ \wedge \ ... \ \wedge \ v_n \in R_n}{\mathbf{exit}(R_1, ..., R_n) \stackrel{\surd v_1 ... v_n}{\longrightarrow} \mathbf{stop}}$$

The BNF rule defining the syntax of the enable operator is:

$$B ::= B_1 \ \texttt{>>} \ [\mathbf{accept} \ x_1{:}t_1 \ ... \ x_n{:}t_n \ \mathbf{in}] \ B_2$$

where $x_1{:}t_1, ..., x_n{:}t_n$ declares a (possibly empty) list of variables $x_i$ of type $t_i$, which are visible in $B_2$ only — if the list is empty, the "**accept**...**in**" clause is omitted. When $B_1$ terminates by invoking an "$\mathbf{exit}(R_1, ..., R_n)$" operator, each variable $x_i$ captures the value returned by the corresponding result $R_i$. Typing constraints ensure that the type of each $R_i$ is the same as $t_i$. The semantics of the "**>>**" operator is defined by two SOS rules:

$$\frac{B_1 \stackrel{a}{\longrightarrow} B_1' \ \wedge \ gate(a) \neq \surd}{B_1 \ \texttt{>>} \ \mathbf{accept} \ x_1{:}t_1 \ ... \ x_n{:}t_n \ \mathbf{in} \ B_2 \stackrel{a}{\longrightarrow} B_1' \ \texttt{>>} \ \mathbf{accept} \ x_1{:}t_1 \ ... \ x_n{:}t_n \ \mathbf{in} \ B_2}$$

$$\frac{B_1 \stackrel{\surd v_1 ... v_n}{\longrightarrow} B_1'}{B_1 \ \texttt{>>} \ \mathbf{accept} \ x_1{:}t_1 \ ... \ x_n{:}t_n \ \mathbf{in} \ B_2 \stackrel{\tau}{\longrightarrow} [v_1/x_1 ... v_n/x_n]B_2}$$

The enable operator has been used in many formal models described using LOTOS or CSP. However, it presents several drawbacks:

1. Each occurrence of an "**exit**" operator creates a $\surd$-transition that is turned into a $\tau$-transition when caught by an enable operator — as pointed out in [2, pp. 191], the enable operator relies on concurrency, synchronization, and communication to perform sequential composition. In practice, the introduction of such extra transitions is a nuisance because it makes labelled transition systems larger, causing or worsening state explosion issues. Another worrying consequence is that the enable operator lacks a neutral element modulo strong bisimulation (the extra $\tau$-transitions can only be ignored if a weaker bisimulation is used), which is different from mainstream programming languages, most of which provide some "empty" statement that does nothing and is a neutral element with respect to sequential composition.

[4]The "$\surd$" gate was noted "$\delta$" in the official definition of LOTOS [10], but we adopt here the CSP notation to avoid any confusion with ACP, where "$\delta$" means inaction.

2. Keeping action prefix and introducing the enable operator leads to two different operators for almost the same purpose: one sequential composition operator that is asymmetric and another one that is symmetric — at least, if one forgets about the "**accept**...**in**" clause. This goes against Occam's razor law[5], and against the idea that a process calculus should contain a minimal set of operators expressing independent ("orthogonal") concepts. For instance, the same sequence of actions can be written in many different ways, e.g. "$(a_1; a_2; \mathbf{exit})$ >> $(a_3; a_4; B)$" vs. "$a_1; a_2; \tau; a_3; a_4; B$". This problem was acknowledged in [38, Sec. 3.4 and 6.5], which proposed a syntactic (yet not semantic) unification of both operators into a single construct noted "**seq** ...**endseq**"; an assessment of this proposal can be found in [39, Sec. 12 and 13].
3. Control-flow sharing is supported by the enable operator, but quite heavy in practice, e.g., "$((a_1; \mathbf{exit})$ [] $(a_2; \mathbf{exit}))$ >> $B_3$", where "[]" is the nondeterministic choice operator (noted "+" in CCS).
4. Data-flow sharing is not possible in CSP, as the values of variables do not pass left-to-right over sequential composition [5, p. 132–133]. In LOTOS, values can pass over sequential composition, e.g.:

$$g?x{:}t\,;\mathbf{exit}(x) \;\texttt{>>}\; \mathbf{accept}\ x'{:}t\ \mathbf{in}\ B(x')$$

making data-flow sharing feasible, yet cumbersome, e.g.:

$$(g_1?x_1{:}t\,;\mathbf{exit}(x_1) \;\texttt{[]}\; g_2?x_2{:}t\,;\mathbf{exit}(x_2)) \;\texttt{>>}\; \mathbf{accept}\ x_3{:}t\ \mathbf{in}\ B_3(x_3)$$

A lighter syntax for the first example above was proposed in [38, Sec. 6.5], but never implemented and assessed on large-size examples.

### *3.3. The "product" operator and its drawbacks*

The action prefix and enable operators are not the only possible approach. From the beginning, ACP explored a different way by adopting a symmetric sequential composition operator, called *product* and noted "·". Its syntax is thus defined by the BNF rule "$B ::= B_1 \cdot B_2$". Following the tradition of ACP, the semantics of this operator is usually given by algebraic axioms, but it can alternatively be defined using SOS rules [12, 9]:

$$\frac{B_1 \xrightarrow{a} B_1'}{B_1 \cdot B_2 \xrightarrow{a} B_1' \cdot B_2} \qquad \frac{B_1 \xrightarrow{a} \surd}{B_1 \cdot B_2 \xrightarrow{a} B_2}$$

where "$\surd$" is a special behavior (distinguishable from all other behaviors) with an associated SOS rule "$a \xrightarrow{a} \surd$" applicable to each action $a$, including "$\tau$". Clearly, this product operator avoids many of the aforementioned pitfalls of action prefix and enable operators: there is a unique symmetric sequential composition, as in mainstream programming languages; it does not create extra

[5] "Entities must not be multiplied beyond necessity".

$\tau$-transitions; it can be equipped with a neutral element (usually noted "$\varepsilon$") [40, 41]; it can be extended with iteration operators to express loops [42]; it subsumes regular expressions and context-free languages [43]; finally, it allows control-flow sharing, which is explicitly permitted by axiom A4 of ACP: $(B_1 + B_2) \cdot B_3 = (B_1 \cdot B_3) + (B_2 \cdot B_3)$ — notice that there is no "dual" axiom $B_1 \cdot (B_2 + B_3) = (B_1 \cdot B_2) + (B_1 \cdot B_3)$ in order to preserve the branching structure.

It remains to examine whether this product operator also supports data-flow sharing. This question cannot be directly answered within ACP, which is a pure process algebra (i.e., does not handle data). Moreover, there is an impossibility claim [44] stating that variable-binding actions are incompatible with associativity of sequential composition; indeed, assuming the existence of a variable-binding action (e.g., an action "$g?x$" that declares variable $x$ and assigns to it an input value), then the term "$g?x \cdot B_1 \cdot B_2$" would be ambiguous, as it can be parsed either as "$(g?x \cdot B_1) \cdot B_2$" or "$g?x \cdot (B_1 \cdot B_2)$", which gives two incompatible scopes for $x$ (either $B_1$ or $B_1 \cdot B_2$); thus, variable binding in ACP should only be achieved by means of unary "prefix" operators.

This principle is followed by the two value-passing languages based on ACP: PSF [12, 13, 14] and $\mu$CRL [15, 16], which both rely on abstract data types and are similar enough to be considered as the consensual (if not official) value-passing extension of ACP. PSF introduces variables and expressions in ACP behaviors at four different places:

$$
\begin{array}{rcll}
B & ::= & \ldots & \\
 & | & g\ (V_1, \ldots, V_n) & \textit{(parameterized action)} \\
 & | & \texttt{[}V\texttt{]} \rightarrow B_0 & \textit{(boolean guard)} \\
 & | & \textbf{sum}\ (x\ \textbf{in}\ t, B_0) & \textit{(summation over data)} \\
 & | & p\ (V_1, \ldots, V_n) & \textit{(parameterized process call)}
\end{array}
$$

The syntax of $\mu$CRL is almost identical, the main difference being that PSF uses (unary) boolean guards whereas $\mu$CRL uses (binary) "then-if-else" conditionals "$B_1 \lhd V \rhd B_2$" that can also be expressed as "$\texttt{[}V\texttt{]} \rightarrow B_1 + \texttt{[}\neg V\texttt{]} \rightarrow B_2$".

The PSF/$\mu$CRL approach to data handling is simpler than its counterparts in CSP and LOTOS. Precisely, it is a strict subset of LOTOS, "$g\ (V_1, \ldots, V_n)$" denoting "$g\,!\,V_1 \ldots !\,V_n$", "**sum** $(x$ **in** $t, B_0)$" denoting "**choice** $x$:$t$ [] $B_0$", PSF/$\mu$CRL guards and process calls having the same syntax as in LOTOS. Even if this restricted set of PSF/$\mu$CRL operators provides the same expressiveness as LOTOS (in the sense that it can generate the same set of labelled transition systems), it has several limitations with respect to convenience and conciseness:

1. Contrary to CCS, CSP, and LOTOS, there is no dedicated syntactic notation to distinguish between emissions and receptions, thus creating omnipresent ambiguities that significantly degrade the readability of behavioral specifications. For instance, a reader confronted to "**sum** $(x$ **in** $t, g(x))$" must rely on some intuitive understanding of the specification to decide whether the writer intended to model the input of $x$ on gate $g$ (noted "$g?x$:$t$ ; ..." in LOTOS) or the output of some non-deterministic value of type $t$ on gate $g$ (noted "**choice** $x$:$t$ [] $g\,!\,x$ ; ..." in

LOTOS). Paradoxically, LOTOS has been often reproached for having untyped gates [45], an issue that does not exist in PSF/$\mu$CRL, where the types of action parameters must be explicitly declared; but LOTOS provides different syntax for inputs and outputs, whereas PSF and $\mu$CRL cannot express in which direction communications take place.

2. In PSF and $\mu$CRL, data values can pass from left to right over the product operator, but only to some limited extent. For instance, it is not possible to directly specify "$g_1\,?x{:}t \cdot g_2\,!\,x$", as the lack of difference between emissions and receptions is equivalent to having emissions only. Instead, one must specify "**sum** $(x$ **in** $t, g_1(x) \cdot g_2(x))$", with the difference that $x$ is not assigned during an input on $g_1$ but before, in the summation operator — said differently, a choice that should be external (i.e., determined by the environment) has to be syntactically written as internal (i.e., locally decided by the current process).
3. PSF and $\mu$CRL permit simple forms of data-flow sharing. For instance, the aforementioned behavior "$(g_1\,?x{:}t + g_2\,?x{:}t) \cdot B_3$", where variable $x$ is visible in $B_3$, can be written "**sum** $(x$ **in** $t, (g_1(x) + g_2(x)) \cdot B_3)$" — again, a single internal choice is used to model two different external choices. However, data-flow sharing ceases to be possible as soon as the behavior becomes slightly more complex, e.g., when at least one reception is not an initial action. For instance, "$(g_1\,?x{:}t + \tau \cdot g_2\,?x{:}t) \cdot B_3$" must not be written "**sum** $(x$ **in** $t, (g_1(x) + \tau \cdot g_2(x)) \cdot B_3)$", as the summation cannot be moved before the $\tau$-transition without altering the branching structure. One has no other option than writing this behavior "**sum** $(x$ **in** $t, g_1(x) \cdot B_3) + \tau \cdot$ **sum** $(x$ **in** $t, g_2(x) \cdot B_3)$", thus duplicating $B_3$, or introducing an auxiliary process that avoids duplicating $B_3$ but degrades the readability of the specification.

## 4. Rationale for enhancing sequential composition

From the previous section, it is clear that none of the widespread process calculi (namely, CCS, CSP, MEIJE, LOTOS, ACP, PSF, and $\mu$CRL) provides an optimal approach to sequential composition. The present section addresses this problem and proposes rationale for a better solution, which has been retained for the LNT language [30] implemented in the CADP toolbox [24]. The purpose of this section is not to describe LNT in full detail, but to explain how sequential composition should be handled in a modern language.

### *4.1. Forerunner languages*

Certain design ideas behind LNT have been put forward in a few precursor languages, which never reached the same level of audience as the widespread process calculi listed above. We pay here a tribute to these almost-forgotten yet inspiring languages by mentioning their contributions and limitations:

- ACP$_\varepsilon$ is a variant of ACP defined in [40, 41] and mentioned in [46] [8] and [42, Chp. 5 Sec. 7.2]. ACP$_\varepsilon$ extends ACP with a new "*empty process*"

constant noted "$\varepsilon$" that is the neutral element for sequential composition. SOS rules for $\mathrm{ACP}_\varepsilon$ can be found in, e.g., [47, Tab. 1] and [48, Tab. 1] as examples of general rule formats; slightly different rules are given in [8, Tab. 11] and discussed in [49, Sec. 6.1]. Value-passing issues for $\mathrm{ACP}_\varepsilon$ are not addressed in the aforementioned references. It seems that $\mathrm{ACP}_\varepsilon$ was never implemented, and its "$\varepsilon$" operator is absent from the value-passing extensions of ACP (namely PSF, $\mu$CRL, and mCRL2).

- $\mathrm{ACP}_G$ (*Algebra of Communicating Processes with Guards*) [50, 51, 52] is a process calculus based on $\mathrm{ACP}_\varepsilon$; it assumes the existence of global variables, the values of which can be consulted using guards (e.g., Boolean conditions) and modified using nondeterministic state transformers (e.g., assignments). The semantics of $\mathrm{ACP}_G$ is carefully defined, both algebraically and operationally, but in a context of open terms (i.e., with free variables only), so that issues related to uninitialized variables (see Sec. 4.4 below) do not show up. Although the work on $\mathrm{ACP}_G$ was somewhat eclipsed by the research on $\mu$CRL by the same authors and had no immediate continuation (excepted perhaps [53]), it provides a convincing treatment of value passing in the presence of symmetric sequential composition, attempts at making process calculi closer to existing specification and programming languages, and investigates the relationships between Hoare logic, on the one hand, and algebraic and operational semantics, on the other hand.

- ACBS& (*Algebra of Broadcasting Systems with Fork*) [54] is a process calculus based on $\mathrm{ACP}_\varepsilon$ in which parallel composition and handshake communication are replaced by forking and (asymmetric) broadcasting, respectively. This design choice impacts the sequential composition operator "$B_1 \cdot B_2$", as $B_2$ can start executing while the processes forked by $B_1$ are still running as background tasks. Value-passing issues for symmetric sequential composition are discussed, but the problem of uninitialized variables is not mentioned. The conclusion sounds quite pessimistic, stating that symmetric sequential composition has no obvious advantage over action prefix in the presence of value passing.

- Extended LOTOS is a language defined by Ed Brinksma in his PhD thesis [38]. Brinksma, who headed the ISO committee that designed the LOTOS standard, proposed in his thesis a different language bringing ideas and extensions not included in the official version of LOTOS. With respect to the present article, the two most relevant features of Extended LOTOS are its bracketed syntax for $n$-ary behavioral operators and its tentative merging of LOTOS action prefix and enable operators into a single, compound operator — these features are summarized and discussed in [39, Sec. 9, 12, and 13].

- E-LOTOS (or Enhanced LOTOS) [55] is an ISO international standard designed between 1992 and 2001. Undertaking a revision of LOTOS to

increase its expressiveness and usability, the E-LOTOS committee eventually produced a new formal language significantly different from LOTOS in many respects, including symmetric sequential composition [56]. Because of its complexity, E-LOTOS has never been implemented, but gave birth to a simplified version named LOTOS NT [57, 58], which itself progressively evolved into LNT.

- OCCAM is a concurrent programming language derived from the original version of CSP [59] [60]. An interesting trait of OCCAM is that its successive versions (1.0 [61], 2.0 [62, 63], 2.1 [64], and 3.0 [65]) were primarily driven by industrial needs, developed away from the theoretical influence of the CCS and ACP schools, and focused on language usability and implementation issues rather than formal semantics. Yet, academia bridged the gap by proposing various semantics for OCCAM, either denotational [66] [67], algebraical [68] [69], or operational [70] [71]. OCCAM made valuable suggestions for symmetric sequential composition and $n$-ary behavioral operators; unfortunately, the language has been abandoned after the Transputer project was cancelled, so that the place left by OCCAM has been taken by the (more traditional) Theoretical CSP process calculus.

In addition to these general-purpose languages, one can also mention a few domain-specific process calculi that also depart from traditional process calculi regarding value passing and sequential composition, e.g., CHP (*Communicating Hardware Processes*) [72], for which an operational semantics is given in [73], MoDeST (*Modeling and Description Language for Stochastic Timed Systems*) [74, 75, 76], Chi [77, 78, 79], and AWN (*Algebra for Wireless Networks*) [80].

### *4.2. Design decision #1: Have symmetric sequential composition*

In spite of its simplicity and its adoption in most process calculi, action prefix has too many drawbacks that cannot be solved. It is actually a limiting factor that hampers a wider dissemination and acceptance of process calculi. One should thus get rid of action prefix and use instead one single symmetric sequential composition operator, for which the product operator of ACP and the enable operator of LOTOS and CSP provide healthy inspiration. Therefore, LNT includes a binary operator noted ";"[6] and two constant operators. So doing, LNT follows the tracks of E-LOTOS and ACP$_\varepsilon$. The corresponding BNF rules are the following ones:

| | | | |
|---|---|---|---|
| $B$ | ::= | **stop** | *(inaction)* |
| | \| | **null** | *(successful termination)* |
| | \| | $a$ | *(action)* |
| | \| | $B_1; B_2$ | |
| | \| | ... | |

[6] The semicolon in LNT is a statement separator rather than a statement terminator.

- Operator "**stop**" is the same as "**nil**" in CCS, "*STOP*" in CSP, "$\delta$" in ACP, and "**stop**" in LOTOS. It has no associated SOS rule.

- Operator "**null**" (the name of which was borrowed from Ada) is the same as "*SKIP*" in CSP, "**exit**" in LOTOS, "**null**" in E-LOTOS, and "$\varepsilon$" in ACP$_\varepsilon$. Its operational semantics is given by the following rule:

$$\frac{-}{\textbf{null} \xrightarrow{\surd} \textbf{stop}}$$

  Notice that "$\surd$" is a special action in LNT, rather than a special behavior as in ACP. So doing, LNT follows the choice of CSP, LOTOS, and ACP$_\varepsilon$ not to have distinguished states and to put all information on transition labels only. Having two different classes of states with respect to termination brings undesirable complexity in software implementations. For instance, equivalence checking and model checking algorithms have to consider information attached not only to transitions but also to states, i.e., operate on Kripke transition systems rather than labelled transition systems. This would also break the expected backward compatibility between LNT and LOTOS.

- In LNT, each action $a$ (different from "$\surd$") is also a behavior, and its semantics is given (as in ACP and ACP$_\varepsilon$) by the following SOS rule:

$$\frac{-}{a \xrightarrow{a} \textbf{null}}$$

- The sequential composition operator of LNT is defined by two SOS rules similar to those for the enable and product operators. The first rule is the same as in CSP, LOTOS, and ACP$_\varepsilon$. The second rule is borrowed from ACP$_\varepsilon$ and differs from CSP and LOTOS by not creating a $\tau$-transition when the control flow passes over sequential composition:

$$\frac{B_1 \xrightarrow{a} B_1' \ \wedge \ a \neq \surd}{B_1; B_2 \xrightarrow{a} B_1'; B_2} \qquad \frac{B_1 \xrightarrow{\surd} B_1' \ \wedge \ B_2 \xrightarrow{a} B_2'}{B_1; B_2 \xrightarrow{a} B_2'}$$

*4.3. Design decision #2: Have "true" (i.e., write-many) variables*

Another feature that isolates process calculi from mainstream programming languages is the status of variables. Following the leading influence of CCS, most process calculi use exclusively *immutable variables* (also: *dynamic constants*), which are assigned only once, at the point they are declared, and whose values cannot change afterwards — unless by creating new, distinct instances of the variables (e.g., parameter instantiation). This functional style sharply contrasts with mainstream programming languages, in which imperative style with multiple assignments to variables is the norm.

The main advantage of the functional style is a slight simplification of syntax (as declarations and initializations of variables occur at the same place)

and operational semantics (as assignments to variables can be expressed using substitutions). But, from a practical point of view, the functional style is not flexible enough to let values naturally pass from left-to-right over sequential composition and permit data-flow sharing.

Below are given six desirable value-passing examples; the four latter ones cannot be properly specified without departing from the functional style in two key points: (i) declarations of variables and assignments to variables must be syntactically dissociated, and (ii) it must be possible to assign the same variable at different places, e.g., in the various branches of deterministic and nondeterministic choices:

$$g_1?x \textbf{ where } x > 0 \,;\, g_2!(x+1)$$
$$g_1?x_1 \,;\, g_2?x_2 \,;\, g_3!(x_1 + x_2)$$
$$(g_1?x \,[]\, g_2?x) \,;\, g_3!(x+1)$$
$$((g_1?x_1 \,;\, g_2?x_2) \,[]\, (g_2?x_2 \,;\, g_1?x_1)) \,;\, g_3!(x_1 + x_2)$$
$$\textbf{if } x_2 > 0 \textbf{ then } g_1?x_1 \textbf{ else } x_1 := 0 \textbf{ end if } \,;\, g_2!(x_1 + x_2)$$
$$\textbf{case } x_2 \textbf{ in } 0 \rightarrow g_1?x_1 \mid 1 \rightarrow x_1 := 0 \textbf{ end case } \,;\, g_2!(x_1 + x_2)$$

In the four latter examples, assignments to the same variables are *alternative*, as they occur on mutually exclusive branches. One may go further by permitting *successive* assignments, so that the values of variables can evolve in time; this is clearly needed to directly express loops (i.e., without using recursive process definitions), as the values computed at some iteration must be passed to the next iterations.

So far, very few process calculi dared to deviate from the functional style set by CCS. Interestingly, the original version of CSP [59] was imperative, as well as its descendents CHP and OCCAM, but CSP later adopted the functional style with Theoretical CSP [3] and shows no intention to switch back [5, p. 132–133]. Extended LOTOS proposed a mixed approach, in which locally imperative sequential code fragments are immersed in a globally functional process calculus [38, chapter 6]. The E-LOTOS standard went further by introducing a unique symmetric sequential composition operator and implementing points (i) and (ii) above: dissociation between variable declarations and assignments, and possibility of alternative assignments; however, E-LOTOS only featured "write-once" variables, explicitly deferring "write-many" variables to a future revision of the standard.

In the realm of mobile processes, the *update calculus* [81] proposes a similar dissociation between, on the one hand, the declaration of a name $x$ local to behavior $P$ (which is done using a scope operator noted "$(x)P$") and, on the other hand, the substitution of all occurrences of $x$ by $y$ (noted "$[y/x]$") or by some value received on port $a$ (noted "$ax$"); such dissociation is a key difference with the $\pi$-calculus where declarations and input actions are bound together; notice however that the update calculus is based on action prefix and uses name substitutions rather than genuine assignments.

Actually, write-many variables are only used in a handful of process calculi.

Beside CHP and OCCAM, one can mention ACBS&, $\text{ACP}_G$ (where modifications of state variables, such as assignments, must be attached to visible or "$\tau$" transitions), AWN and Chi (where each assignment systematically creates a potentially undesirable $\tau$-transition), and MoDeST (where assignments may only occur as operands of a probabilistic choice taking place after a visible or "$\tau$" action). LNT does not have such restrictions and handles "write-many" variables in the same way as most imperative programming languages do. Focusing on the places where variables are declared and modified, the BNF rules for LNT behaviors are extended as follows (brackets "[...]" meaning optional):

| | | | |
|---|---|---|---|
| $B$ | ::= | ... | |
| | \| | **var** $x$:$t$ **in** $B_0$ **end var** | *variable declaration* |
| | \| | $x := V$ | *(deterministic) assignment* |
| | \| | $x :=$ **any** $t$ [**where** $V$] | *nondeterministic assignment* |
| | \| | $g\ (O_1, ..., O_n)$ [**where** $V$] | *value-passing action* |
| | \| | $p$ `[`$g_1, ..., g_m$`]` $(N_1, ..., N_n)$ | *process call* |
| | \| | ... | |
| $O$ | ::= | [`!`]$V$ | *emission offer* |
| | \| | `?`$x$ [**where** $V$] | *reception offer* |
| $N$ | ::= | $V$ | *"in" parameter* |
| | \| | `?`$x$ | *"out" parameter* |
| | \| | `!?`$x$ | *"in-out" parameter* |

These rules deserve a few comments:

- The "**var**" operator declares a variable that only exists in the scope of behavior $B_0$. This operator (similar to, e.g., the "**var**" operator of Esterel) is the dual of the "**hide**" operator of LOTOS, which declares a gate only visible in the scope of a given behavior; such duality between variables and gates goes further because the only other places where variables (resp. gates) can be declared are the formal variable (resp. gate) parameters of process definitions. Notice that the ability to declare local variables does not exist in all process calculi; for instance, $\text{ACP}_G$ only allows global variables.

- The "**where** $V$" optional clauses (called *selection predicates* in LOTOS) are Boolean constraints specifying the acceptable values for nondeterministic assignments and value receptions.

- To follow Ross's *uniform referents* principle [82] and to pave the way for action refinement [37], the syntax adopted for value-passing actions in LNT is a mix between CCS (the "`?`" symbol is kept and the "`!`" symbol becomes optional) and PSF/$\mu$CRL (value-passing actions are written with the same conventions as function and procedure calls in programming languages).

- In the imperative style of LNT, processes can have "**in**" parameters (call by value), "**out**" parameters (call by result), and "**in out**" parameters (call by value-result). The evaluation strategy is strict.

Notice that the dissociation between variable declarations (using the "**var**" operator) and variable modifications (i.e., assignments or receptions) is the only way to defeat the impossibility claim of [44] recalled in Sec. 3.3 above: sequential composition can indeed be associative as expected if its operands do not declare variables but only modify them.

The syntax proposed for LNT behaviors is permissive enough to express all aforementioned examples. However, it is not free from difficulties, which we consider in the next sections.

### *4.4. Design decision #3: Prohibit uninitialized variables*

Unfortunately, certain behaviors accepted by the syntax of LNT have no obvious semantics. For instance, the behavior $B$ defined as "$(g_1\texttt{?}x_1\,\texttt{[]}\,g_2\texttt{?}x_2)\,;\,g_3\texttt{!}(x_1 + x_2)$" is problematic because, depending on which execution branch is taken, either $x_1$ or $x_2$ is not assigned, and thus the sum $(x_1 + x_2)$ may be undefined. However, this behavior $B$ becomes trouble-free if inserted in some larger behavior, such as "$x_1 := 0\,;x_2 := 0\,;B$". Various approaches can be considered for this problem:

1. The semantics could (explicitly or implicitly) state that reading uninitialized variables has "undefined" (i.e., implementation-dependent) effects. This is a usual solution for imperative languages (e.g., C) and sometimes for process calculi as well (e.g., CHP and, to some extent, the early versions of Chi in which undefined variables are said to remain symbolic when evaluating expressions [83, pp. 54 and 250] [77, p. 375]). However, this solution is unacceptable in the context of formal methods; moreover, determining whether a behavior is "undefined" or not is undecidable in general — this is equivalent to the halting problem.
2. The semantics could implicitly initialize each variable to some default value when the variable is declared. This is the approach taken in Eiffel, which automatically sets integers to zero, Booleans to false, etc. Such a semantics is formal, but presents the risk of silently hiding user mistakes, namely all forgotten assignments with values different from the default ones.
3. The semantics could state that reading an uninitialized variable triggers some particular behavior. For instance, the various definitions of OCCAM are not very precise on this matter, but it seems that any behavior requiring the evaluation of an expression containing some uninitialized variable should behave like "`STOP`" [63, pp. 17 and 215], although the OCCAM compiler can also be instructed to halt the whole system in such event [62, Annex E]. Alternatively, a catchable exception [84] could be raised. In such approaches, the detection of user mistakes is postponed at run time, which is not advisable for safety- and security-critical designs.
4. The semantics could state that reading an uninitialized variable provokes a nondeterministic assignment ("$x :=$ **any** $t$") to this variable. Such a semantics would be formal, but it would be non-standard (although it has been implemented in a CHP-to-LOTOS translator [73]) and would also

hide user mistakes (yet, complexity explosions occurring during state-space generation could indirectly inform the user about potential mistakes).

5. Our idea consists in adding *static semantics* constraints that reject syntactically correct, yet semantically problematic behaviors. This is standard practice in programming languages, where static semantics rules out programs containing, e.g., undefined identifiers, mistyped expressions, etc. This way, one can introduce restrictions to ensure that all variables are properly initialized; this idea was first experimented in IBM's NIL and Hermes languages [85] and later adopted in Java.

LNT follows the fifth approach. The LNT compiler implements data-flow analysis techniques to statically detect and reject "dubious" specifications in which some variables might be used before set.

Of course, it is not possible to decide, for all specifications, whether all variables are set before used, but one can check instead *sufficient conditions* that only accept those specifications in which a proper initialization of variables can be proven, and reject all other specifications in which such initialization is uncertain. There is obviously a risk of rejecting specifications that make sense; for instance, the current LNT compiler front-end does not accept the following behavior:

$$\textbf{if } x_1 \leq x_2 \textbf{ then } x_3 := 0 \textbf{ end if}; \textbf{if } x_2 > x_1 \textbf{ then } g\ (x_3) \textbf{ end if}$$

because the analysis is not involved enough to discover that the second condition is implied by the first one. In practice, such *false positives* are not too frequent, and it is not so difficult to modify an LNT specification to get it accepted by the compiler. Moreover, the experience with Java indicates that data-flow constraints uncover many mistakes and are appreciated by programmers. Finally, the data-flow algorithms are always perfectible and will certainly benefit, for instance, from recent advances in static analysis and SAT/SMT solvers; the set of accepted LNT specifications is thus expected to grow as time passes.

Looking retrospectively at traditional process calculi, it is clear they also set constraints to rule out unwanted behaviors. For instance, action prefix in CCS and "input-only" actions in PSF/$\mu$CRL are two different means to forbid problematic behaviors such as "$(g_1\texttt{?}x_1\ \texttt{[]}\ g_2\texttt{?}x_2)\,;g_3\texttt{!}(x_1 + x_2)$". But these are *syntactic* restrictions, i.e., very primitive defense means that bluntly reject many meaningful and useful behaviors. To the contrary, the LNT approach has a more permissive syntax combined with *static semantics* restrictions, leading to a language that is more flexible and closer to end-user intuition and expectations.

### *4.5. Design decision #4: Prohibit shared variables*

The dissociation between declarations of variables and assignments to variables, together with the introduction of "write-many" variables creates new issues with parallel composition as well. For instance, it allows behaviors of the form "**var** $x{:}t$ **in** $(g_1(\texttt{?}x)\ \texttt{||}\ g_2(\texttt{?}x))\,;g_3(x)$ **end var**", where "||" denotes some

parallel composition operator, so that $x$ is actually a shared variable, whose semantics depends on the possible interleavings.

This issue has been diversely addressed in the various forerunner languages that permit variable assignment. In CHP, for instance, such situation cannot occur due to a restrictive syntax: there is no explicit parallel composition operator, so that concurrent processes operate in disjoint memory spaces without shared variables. In OCCAM, a parallel composition is considered to be invalid if any of its branches may change the value of a variable used in another branch; this situation is handled at run time: in such case the parallel composition may stop (while other parts of the system continue their execution) or halt the entire system. In ACBS&, each parallel branch forks to execute in its own memory space that contains copies of shared variables; race-condition issues are thus deferred until join points when forked processes return, with several possible semantics to merge once-separate memory spaces [54, Sec. 6.4].

LNT follows a similar approach to OCCAM, but ensures the existence of a formal semantics. Rather than stopping or halting at run time, LNT adds static semantic constraints that forbid at compile time all (syntactically correct) behaviors involving shared variables. Namely, all parallel branches can see the variables declared in an enclosing scope; by default, all parallel branches can read such shared variables, but if a branch writes to a shared variable, the other branches can neither write nor read this variable. When parallel composition terminates, the disjoint variables computed by each branch are merged into a single memory space, e.g., "**var** $x_1, x_2{:}t$ **in** $(g_1(?x_1) \mid\mid g_2(?x_2)) \,;\, g_3(x_1 + x_2)$ **end var**" — actually, this corresponds to the "combine" semantics of ACBS& [54, Sec. 6.4.3].

With such a design decision, LNT remains in the strict message-passing framework of traditional process calculi, as the most direct motivation behind LNT was the need for a better language to replace LOTOS in academia and industry. On the longer term, static semantics restrictions on shared variables could be relaxed, giving a nice opportunity to combine message-passing and shared-memory paradigms in the same formal language. Work in this direction already exists, such as $\text{ACP}_G$, which allows shared variables without restriction (race conditions leading to nondeterminism, as each variable keeps the last value it was assigned during the interleaved execution of parallel branches). The aforementioned shared-variable languages that translate to traditional process calculi [2, Chapter 8] [27] [28] [5, Chapter 18] also provide a basis for reflection. However, given the multiplicity and complexity of shared-memory models, machines, and languages, a deep reflection is needed if one wants to produce a language general enough to accurately model a variety of parallel hardware and software.

### *4.6. Design decision #5: Prohibit complex forms of process recursion*

It is well-known from the ACP body of knowledge that symmetric sequential composition opens the way to context-free recursion, e.g., process definitions of the form "$p = (a_1 \cdot p \cdot a_2) + a_3$", a situation syntactically prohibited by action

prefix operator. Also, "write-many" variables may lead to problematic behaviors when combined with recursion and other operators.

Such issues have not yet been fully explored in LNT, for two reasons. First, such forms of process recursion are not so useful in practice, as the desired behaviors can be described in a different way, using regular processes with additional variables to encode recursion stacks. Second, LNT is currently implemented by translation to LOTOS, and the LOTOS compiler of CADP [86] does not accept recursion through parallel composition operators, nor on the left-hand side of the enable and disable operators.

Therefore, restrictions have been added to the static semantics of LNT in order to disallow such forms of recursion. Relaxing these restrictions should trigger future investigation, based on existing work to automatically eliminate non-tail process recursion [87]. Notice that no similar restrictions exist for recursive functions, because LNT functions are translated to C functions, thus deferring the handling of recursion to the C compiler.

### *4.7. Design decision #6: Have structured programming constructs*

Most of the discussion above is about sequential composition. Regarding the remaining behavioral operators, the following remarks can be made:

1. With traditional process calculi, it was sometimes tricky to introduce certain operators, such as the empty process or loops. These difficulties vanish once symmetric sequential composition and "write-many" variables are adopted. Introducing the usual structured programming constructs then becomes straightforward. LNT has thus most Ada-like constructs, including "**if-then-else**" with "**elsif**", "**case**", forever "**loop**" with "**break**", "**while**" loop, "**for**" loop, etc. Notice that the "**case**" operator of LNT supports ML-like pattern matching, whereas traditional process calculi have no "**case**" operator at all in their control part, partly compensated with algebraic equations in their data part only. Nearly 70% of LNT behavioral operators look familiar to average computer scientists; this favors the acceptance of the language.
2. Opting for such imperative programming constructs (disciplined by static semantics constraints on variable initialization) brings a proper solution to another problem faced by most specification languages based on process calculi: the coexistence of two different languages, one to describe the control part (namely, communication channels and processes), and another one to describe the data part (namely, types and functions). For the latter part, algebraic data types have been often used (e.g., in LOTOS, ACP/SDF, ACP/COLD, PSF, $\mu$CRL, mCRL2, etc.), but functional language are also possible, as in LCS (which combines CCS and ML) [88] or CSP$_M$ (which combines CSP and Haskell) [89, Appendix B]. Whatever the approach chosen, juxtaposing two languages increases complexity and often violates the uniform referents principle: in most process calculi, for instance, "**if-then-else**" conditionals are expressed very differently whether

they occur in the control part or in the data part, causing much confusion among beginners.

To our knowledge, the ISO committee that designed E-LOTOS was the first to state this problem and propose a unifying approach in which the language to describe functions is a subset of the language to describe processes. LNT follows the same inspiration, by considering a function as a restricted form of process that is deterministic, cannot perform (visible nor internal) actions, and in which only sequential programming constructs can be used — to be complete, there is a minor exception to this rule at the moment: functions can return a result using a "**return**" construct that processes do not have, but this could easily be addressed by allowing processes as well to (optionally) return results when they terminate, keeping in mind that processes already have "**out**" and "**in out**" parameters.

3. The remaining behavioral operators that can be used in processes (but not in functions) are: nondeterministic choice (noted "**select**" as in Ada), nondeterministic assignment ("$x$ := **any** $t$"), parallel composition (which, in LNT, is based on the so-called *graphical* parallel operators [90]), action hiding, and disruption (i.e., the disable operator noted "`[>`" in LOTOS).
4. Process calculi were originally designed to reason about small yet salient examples but have been later used to model increasingly large systems. At this point, the algebraic syntax becomes a nuisance for readability, leading to (LISP-like) excessive nestings of parentheses. LOTOS tried to address this issue by defining eight levels of precedence between behavioral operators, but these levels are not intuitive and provoke mistakes that are difficult to catch.

   A better solution was proposed by OCCAM, which replaced binary operators by $n$-ary operators ("`SEQ` $B_1 ... B_n$" for sequential composition, "`ALT` $B_1 ... B_n$" for choice, "`PAR` $B_1 ... B_n$" for parallel composition, etc.), while relying on indentation levels[7] to disambiguate nested behaviors. Such a proposal actually conveyed the hidden yet subversive message that a language for concurrency could choose the most appropriate syntax and was not fatally bound to imitate arithmetics.

   The OCCAM solution was improved in Extended LOTOS, which suppressed the questionable reliance on indentation levels by introducing fully bracketed $n$-ary constructs, e.g., "**seq** $B_1 ; ... ; B_n$ **endseq**" for sequential composition, "**sel** $B_1$ `[]` ... `[]` $B_n$ **endsel**" for choice, "**par** $B_1$ `||` ... `||` $B_n$ **endpar**" for parallel composition, etc. This idea, retained for E-LOTOS, was also adopted for LNT with two differences inspired from Ada: removing the "**seq**" and "**endseq**" keywords (which are not needed if all other constructs are properly bracketed), and splitting each keyword "**endxxx**" into "**end xxx**" to reduce the overall number of keywords. MoDeST chose a similar approach, using instead a C-like

[7] Namely, the so-called *off-side rule*.

syntax with braces. Notice that bracketed $n$-ary operators are the most sensible option to introduce "**case**", graphical parallel composition [90], and probabilistic choice operators, at least.

### *4.8. Formal definition of LNT*

The definition of traditional process calculi consists of two parts: syntax and dynamic semantics, the latter being specified algebraically (axioms) or operationally (SOS rules). Due to the design choices for LNT, a third part must be added, static semantics, which is usually absent from traditional process calculi or kept to a minimum (e.g., a few rules stating that value expressions must be well-typed). In LNT, static semantics is larger and cannot be overlooked, as it is a condition for the soundness of dynamic semantics.

In practice, one observes that new users learn the syntax and static semantics of LNT, but tend to ignore the dynamic semantics, based upon the (reasonable) assumption that anything permitted by the syntax and not forbidden by the static semantics should work simply just as expected. Although such a situation is annoying from a semanticist's point of view, it is in line with mainstream programming languages, the formal semantics of which is only studied by standardization committees, compiler writers, and test-suite producers, rather than being a prerequisite to start using the language.

The syntax, static semantics, and dynamic semantics of LNT are given in [30]. Formalized by Frédéric Lang, the dynamic semantics is defined operationally [30, Annex B] and comprises two parts:

- For LNT functions, each memory state is represented as a *store*, i.e., a mapping from variables to their current values; the execution of LNT constructs defines transitions between these states (i.e., store updates).

- For LNT processes, the dynamic semantics is given in terms of labelled transition systems. Following a minoritarian yet longstanding tradition [91, 92] [93] [94] [50, 51, 52] [54] [95] [96, 97], each state is defined as a pair[8] $\langle B, \sigma \rangle$, where $B$ is a behavior and $\sigma$ a store, rather than a behavior only[9]. SOS rules define transitions between states; static semantics restrictions avoid complications, ensuring that each variable read has a defined value and avoiding race conditions when the store is accessed.

The dynamic semantics of LNT being too large (17 pages) to be entirely reproduced here, we only give a few salient SOS rules and compare them with the rules of $\text{ACP}_G$ [51, Fig. 3 and 9]. Let "$\langle V, \sigma \rangle \longrightarrow v$" denote that a value expression $V$ evaluates to $v$ in store $\sigma$, and let "$\langle B, \sigma \rangle \xrightarrow{a} \langle B', \sigma' \rangle$" denote that it exists a transition labelled $a$ from state $\langle B, \sigma \rangle$ to state $\langle B', \sigma' \rangle$. The two first rules extend, by adding stores, those already given in Sec. 4.2:

[8] Such a pair is often called a *configuration* in the literature.

[9] Systematic conversion from the former to the latter approach is studied in [98].

$$\frac{-}{\langle \mathbf{null}, \sigma \rangle \xrightarrow{\surd} \langle \mathbf{stop}, \sigma \rangle} \qquad \frac{-}{\langle g, \sigma \rangle \xrightarrow{g} \langle \mathbf{null}, \sigma \rangle}$$

The two next rules consider actions with emission and reception offers (these are not supported in $\mathrm{ACP}_G$):

$$\frac{\langle V, \sigma \rangle \longrightarrow v}{\langle g\,(V), \sigma \rangle \xrightarrow{g\,(v)} \langle \mathbf{null}, \sigma \rangle} \qquad \frac{v \in \mathit{type}(x) \ \wedge \ \sigma' = \sigma[x/v] \ \wedge \ \langle V, \sigma' \rangle \longrightarrow \mathit{true}}{\langle g\,(?x) \ \mathbf{where}\ V, \sigma \rangle \xrightarrow{g\,(v)} \langle \mathbf{null}, \sigma' \rangle}$$

where $\mathit{type}(x)$ denotes the type of variable $x$, and where $\sigma[x/v]$ is the store obtained from $\sigma$ by assigning value $v$ to variable $x$. The two next rules define deterministic and nondeterministic assignments:

$$\frac{\langle V, \sigma \rangle \longrightarrow v}{\langle x := V, \sigma \rangle \xrightarrow{\surd} \langle \mathbf{stop}, \sigma[x/v] \rangle} \qquad \frac{v \in t \ \wedge \ \sigma' = \sigma[x/v] \ \wedge \ \langle V, \sigma' \rangle \longrightarrow \mathit{true}}{\langle x := \ \mathbf{any}\ t\ \mathbf{where}\ V, \sigma \rangle \xrightarrow{\surd} \langle \mathbf{stop}, \sigma' \rangle}$$

Although such assignments can also be expressed in $\mathrm{ACP}_G$ (by means of state transformers), they must be attached to visible or "$\tau$" transitions, whereas LNT is more general by attaching them to $\surd$-transitions; as a consequence, $\surd$-transitions never modify the store in $\mathrm{ACP}_G$, whereas they can in LNT. Another difference is that $\mathrm{ACP}_G$ requires state transformers to have at least one solution, whereas LNT does not — namely, the condition $V$ used in a Boolean guard "**where** $V$" is allowed to be always false, in which case the assignment or reception containing this guard is simply equivalent to "**stop**".
Finally, the two main rules for value-passing sequential composition are the following ones (a third rule is actually needed to model the "**break**" statements used to exit from loops):

$$\frac{\langle B_1, \sigma \rangle \xrightarrow{a} \langle B_1', \sigma' \rangle \wedge\, a \neq \surd}{\langle B_1; B_2, \sigma \rangle \xrightarrow{a} \langle B_1'; B_2, \sigma' \rangle} \qquad \frac{\langle B_1, \sigma \rangle \xrightarrow{\surd} \langle B_1', \sigma' \rangle \wedge \langle B_2, \sigma' \rangle \xrightarrow{a} \langle B_2', \sigma'' \rangle}{\langle B_1; B_2, \sigma \rangle \xrightarrow{a} \langle B_2', \sigma'' \rangle}$$

These rules are similar to those of $\mathrm{ACP}_G$, except that the second premiss of the second rule differs: $\mathrm{ACP}_G$ uses $\langle B_2, \sigma \rangle \xrightarrow{a} \langle B_2', \sigma'' \rangle$ whereas LNT uses $\langle B_2, \sigma' \rangle \xrightarrow{a} \langle B_2', \sigma'' \rangle$ because $\surd$-transitions can modify the store in LNT but not in $\mathrm{ACP}_G$.

## 5. Comparison with traditional process calculi

In this section, we discuss the relationship between LNT and established process calculi, focusing on sequential composition and excluding from the comparison the well-known differences arising from parallel composition, binary vs. multiway synchronization, hiding and/or renaming of actions, etc.

As a preliminary remark, it is worth noticing that LNT without its choice, parallel, hiding, and disabling operators, provides a small yet powerful language offering a combination of imperative and functional traits. This allows LNT to be taught in successive steps, starting from well-known sequential aspects, then progressively introducing concurrency-related features.

*5.1. Expressiveness of LNT*

The design of LNT is not driven by the desire to increase theoretical expressiveness, but by pragmatic considerations about the wide applicability of the language. At present, LNT is implemented by translation to LOTOS, so as to reuse the compilers already available in the CADP toolbox. More precisely, LNT is translated into a combination of LOTOS (slightly extended by allowing the enable operator "`>>`" not to create extra $\tau$-transitions) and C (which serves to efficiently implement certain LNT types that do not exist in LOTOS, e.g., floating-point numbers and character strings). Despite these extensions, we conclude that the expressiveness of LNT is roughly the same as that of LOTOS.

*5.2. Encoding of regular expressions in LNT*

To start with a simple example, the following function $\mathcal{T}$ translates a regular expression $R$ to an LNT behavior such that the automaton obtained from the labelled transition system of $\mathcal{T}(R)$ by considering as accepting states all states having an outgoing $\surd$-transition recognizes the language of $R$; the two last equations extend the definition of $\mathcal{T}$ to $\omega$-regular expressions:

$$\begin{aligned}
\mathcal{T}(\varepsilon) &= \textbf{null} \\
\mathcal{T}(g) &= g \\
\mathcal{T}(B_1 \cdot B_2) &= \mathcal{T}(B_1)\,;\mathcal{T}(B_2) \\
\mathcal{T}(B_1 + B_2) &= \textbf{select } \mathcal{T}(B_1) \texttt{ [] } \mathcal{T}(B_2) \textbf{ end select} \\
\mathcal{T}(B^*) &= \textbf{loop select } \mathcal{T}(B) \texttt{ [] } \textbf{break end select end loop} \\
\mathcal{T}(B^\infty) &= \textbf{loop } \mathcal{T}(B) \textbf{ end loop} \\
\mathcal{T}(B^\omega) &= \textbf{select } \mathcal{T}(B^*) \texttt{ [] } \mathcal{T}(B^\infty) \textbf{ end select}
\end{aligned}$$

Studies of the correspondences between regular expressions and process calculi when considering stronger relations rather than language equivalence can be found in [99, 100, 101] and [102, 103].

*5.3. Encoding of CCS in LNT*

Migrating from CCS to LNT can easily be done using the following function $\mathcal{T}$ that translates a CCS term to an LNT behavior:

$$\begin{aligned}
\mathcal{T}(\textbf{nil}) &= \textbf{stop} \\
\mathcal{T}(g.B) &= g\,;\mathcal{T}(B) \\
\mathcal{T}(g(V).B) &= g(V)\,;\mathcal{T}(B) \\
\mathcal{T}(\overline{g}(x).B) &= g(\texttt{?}x)\,;\mathcal{T}(B) \\
\mathcal{T}(\tau.B) &= \textbf{i}\,;\mathcal{T}(B) \\
\mathcal{T}(B_1 + B_2) &= \textbf{select } \mathcal{T}(B_1) \texttt{ [] } \mathcal{T}(B_2) \textbf{ end select} \\
\mathcal{T}(\textbf{if } V \textbf{ then } B) &= \textbf{only if } V \textbf{ then } \mathcal{T}(B) \textbf{ end if}
\end{aligned}$$

The "**only**" keyword before the "**if**" implicitly adds a blocking "**else stop**" rather than the default clause "**else null**" added by default when "**only**" is absent. Also, each CCS process call is translated to a corresponding LNT process call, but it is also possible to express recursive CCS processes using the "**loop**" operators of LNT. As stated above, the parallel, relabelling, and restriction operators are left out of the discussion. The same remarks will also apply to the next sections.

*5.4. Encoding of (value-passing) ACP in LNT*

Similarly, many operators of ACP, PSF, and $\mu$CRL have a simple counterpart in LNT, given by the following function $\mathcal{T}$ that translates an ACP term to an LNT behavior:

$$\begin{aligned}
\mathcal{T}(\delta) &= \textbf{stop} \\
\mathcal{T}(\varepsilon) &= \textbf{null} \\
\mathcal{T}(g) &= g \\
\mathcal{T}(g(V1,...,Vn)) &= g(V1,...,Vn) \\
\mathcal{T}(\tau) &= \textbf{i} \\
\mathcal{T}(B_1 \cdot B_2) &= \mathcal{T}(B_1)\,;\mathcal{T}(B_2) \\
\mathcal{T}(B_1 + B_2) &= \textbf{select } \mathcal{T}(B_1) \texttt{ [] } \mathcal{T}(B_2) \textbf{ end select} \\
\mathcal{T}(\texttt{[}V\texttt{]} \rightarrow B) &= \textbf{only if } V \textbf{ then } \mathcal{T}(B) \textbf{ end if} \\
\mathcal{T}(B_1 \triangleleft V \triangleright B_2) &= \textbf{if } V \textbf{ then } \mathcal{T}(B_1) \textbf{ else } \mathcal{T}(B_2) \textbf{ end if} \\
\mathcal{T}(\textbf{sum } (x \textbf{ in } t, B)) &= \textbf{var } x{:}t \textbf{ in } x := \textbf{any } t\,;\mathcal{T}(B) \textbf{ end var}
\end{aligned}$$

One should of course keep in mind that "$\surd$" is a state in ACP and a transition in LNT, leading to minor differences in the labelled transition systems: for instance, the LTS of "$\varepsilon$" in ACP has one state and no transition, whereas the LTS of "**null**" in LNT has two states and one $\surd$-transition; similarly, the LTS of an action in ACP has two states and one transition, whereas the LTS of that same action in LNT has three states and two transitions (the last one being a $\surd$-transition).

*5.5. Encoding of LOTOS and CSP in LNT*

As mentioned above, LNT is currently implemented by translation to LOTOS. But the reverse translation from LOTOS to LNT is possible — and much simpler. It is given by the following function $\mathcal{T}$ that translates a LOTOS term to an LNT behavior; this translation also holds (modulo notational changes) for

a large fragment of CSP:

$$
\begin{array}{rcl}
\mathcal{T}(\textbf{stop}) & = & \textbf{stop} \\
\mathcal{T}(\textbf{exit}) & = & \textbf{null} \\
\mathcal{T}(g\,;B) & = & g\,;\mathcal{T}(B) \\
\mathcal{T}(g\,!V\,;B) & = & g(V)\,;\mathcal{T}(B) \\
\mathcal{T}(g\,?x{:}t\,;B) & = & \textbf{var}\ x{:}t\ \textbf{in}\ g(?x);\mathcal{T}(B)\ \textbf{end var} \\
\mathcal{T}(g\,?x{:}t\ [V]\,;B) & = & \textbf{var}\ x{:}t\ \textbf{in}\ g(?x)\ \textbf{where}\ V;\mathcal{T}(B)\ \textbf{end var} \\
\mathcal{T}(\textbf{i}\,;B) & = & \textbf{i}\,;\mathcal{T}(B) \\
\mathcal{T}(B_1\,[]\,B_2) & = & \textbf{select}\ \mathcal{T}(B_1)\ []\ \mathcal{T}(B_2)\ \textbf{end select} \\
\mathcal{T}([V]\rightarrow B) & = & \textbf{only if}\ V\ \textbf{then}\ \mathcal{T}(B)\ \textbf{end if} \\
\mathcal{T}(\textbf{let}\ x{:}t = V\ \textbf{in}\ B) & = & \textbf{var}\ x{:}t\ \textbf{in}\ x := V\,;\mathcal{T}(B)\ \textbf{end var} \\
\mathcal{T}(\textbf{choice}\ x{:}t\ []\,B) & = & \textbf{var}\ x{:}t\ \textbf{in}\ x := \textbf{any}\ t\,;\mathcal{T}(B)\ \textbf{end var} \\
\mathcal{T}(B_1 >> B_2) & = & \mathcal{T}(B_1)\,;\textbf{i}\,;\mathcal{T}(B_2) \\
\mathcal{T}(B_1\,[>B_2) & = & \textbf{disrupt}\ \mathcal{T}(B_1)\ \textbf{by}\ \mathcal{T}(B_2)\ \textbf{end disrupt}
\end{array}
$$

Despite their apparent simplicity, certain equations above are not trivial. For instance, the third equation converts an asymmetric action prefix operator into a symmetric sequential composition operator. The two next equations are given only for the emission/reception of a single offer, but naturally extend to the case of multiple offers.

The translation of value-passing "**exit**" and "**>> accept**" operators is more involved and not described by the above equations. The key idea is that LNT totally suppresses the need for such complex operators, which have been (for good reasons) introduced in LOTOS long ago. In essence, the elimination of these operators is summarized by the two following equations:

$$
\begin{array}{l}
\mathcal{T}(...\ \textbf{exit}(V)\ >>\ \textbf{accept}\ x{:}t\ \textbf{in}\ B) = \\
\qquad ...\ \textbf{i}\,;\textbf{var}\ x{:}t\ \textbf{in}\ x := V\,;\mathcal{T}(B)\ \textbf{end var} \\
\mathcal{T}(...\ \textbf{exit}(\textbf{any}\ t)\ >>\ \textbf{accept}\ x{:}t\ \textbf{in}\ B) = \\
\qquad ...\ \textbf{i}\,;\textbf{var}\ x{:}t\ \textbf{in}\ x := \textbf{any}\ t\,;\mathcal{T}(B)\ \textbf{end var}
\end{array}
$$

## 6. Assessment on design patterns

In Sec. 3, several examples have been given that cannot be conveniently expressed in traditional process calculi. The present section goes further by providing larger examples (borrowed from typical design patterns) that can clearly be better expressed in LNT than with former process calculi.

### *6.1. Design pattern #1: Control-flow sharing*

The benefits of control-flow sharing (i.e., the ability to specify dag-like patterns and not only tree-like patterns) have already been demonstrated on small

examples. Here is a more involved LNT example featuring a sequence of "**if-then**" conditionals — similar examples could be built using sequences of "**if-then-else**", "**case**", or "**select**" operators:

```
-- here, let x_1, x_2, ..., x_n be initialized integer variables
if x_1 = 0 then g (?x_1) end if;
if x_2 = 0 then g (?x_2) end if;
...
if x_n = 0 then g (?x_n) end if;
g' (x_1 + x_2 + ... + x_n)
```

Traditional process calculi do not allow to concisely express this example: they require a quadratic complexity in the number of variables, whereas LNT permits a linear complexity. Using LOTOS would be the best option: a sequence of $(n-1)$ enable operators could be used, the $(i+1)$-th operator having an "**accept**" clause to get all variables $x_1, ..., x_i$; therefore, $n(n-1)/2$ auxiliary variables would be needed. For other languages based on action prefix, the only option to avoid unfolding $2^n$ branches would be to introduce $(n-1)$ auxiliary processes, the $(i+1)$-th process having $i$ parameters to get the values of $x_1, ..., x_i$; again, $n(n-1)/2$ auxiliary variables would be needed (in addition to the processes themselves). For the ACP-based languages PSF and $\mu$CRL, despite symmetric sequential composition, the introduction of $(n-1)$ auxiliary processes totalling $n(n-1)/2$ parameters is also unavoidable, given that input actions are not allowed, and that summations and process calls are the only places where variables can be bound to values. No need to emphasize that quadratic-size specifications are error prone, and tedious to write, read, and maintain.

### *6.2. Design pattern #2: Guarded commands*

A second class of patterns that are easier to express using LNT than traditional process calculi are Dijkstra's *guarded commands* [104], which many model checkers adopted as their input language. LNT can express guarded commands naturally and concisely, e.g.:

```
var x_1, x_2, ..., x_n:int in
  x_1 := 0 ; x_2 := 0 ; ... ; x_n := 0 ;
  loop
    select
      only if x_1 < 10 then x_1 := x_1 + 1 ; B_1 end if
      []
      only if x_2 < 10 then x_2 := x_2 + 1 ; B_2 end if
      [] ... []
      only if x_n < 10 then x_n := x_n + 1 ; B_n end if
    end select
  end loop
end var
```

where $B_1, B_2, ..., B_n$ can be arbitrary behaviors. Using traditional process calculi that lack a "**loop**" operator, one must introduce a recursive process with $n$ parameters $x_1, x_2, ..., x_n$. Because this process is called $n$ times (once per branch), $n^2$ actual parameters are passed. Therefore, the size of the code, linear in LNT, becomes quadratic. The need to extend ACP-like value-passing process calculi to properly express guarded commands was acknowledged in [51, Sec. 6].

*6.3. Design pattern #3: Map-Reduce*

In distributed computing, the *map-reduce* pattern is a means to divide a task between parallel workers. For instance, given $n$ inputs $x_1, x_2, ..., x_n$ and an expression "$f(f_1(x_1), f_2(x_2), ..., f_n(x_n))$" to compute, where $f, f_1, f_2, ..., f_n$ are function symbols, one can ask $n$ workers to compute the partial results "$y_i = f_i(x_i)$" in parallel. This pattern can easily be expressed in LNT, where parallel branches can assign variables declared in their enclosing scope(s):

```
-- here, let x_1, x_2, ..., x_n be initialized variables
var y_1, y_2, ..., y_n:t in
  par
    y_1 := f_1(x_1)
    ||
    y_2 := f_2(x_2)
    ||...||
    y_n := f_n(x_n)
  end par;
  g (f(y_1, y_2, ..., y_n))
end var
```

Among traditional process calculi, LOTOS is probably the one that can most concisely express this pattern. This can be done using the "**>> accept**" operator, but with a quadratic complexity as $n$ "**exit**" operators (each with $n$ arguments) are needed:

```
-- here, let x_1, x_2, ..., x_n be initialized variables
(
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 g !f(y_1, y_2, ..., y_n); ...
```

It is worth noticing that the above LOTOS fragment is not compositional, as each branch must be aware of its own position in the parallel composition to properly pass arguments to "**exit**". Moreover, the addition of an $(n+1)$-th branch would require to modify all the branches.

Other traditional process calculi than LOTOS lack a native synchronized termination in which each parallel branch could contribute by passing its locally

computed results. The map-reduce pattern can still be expressed in these calculi by introducing auxiliary gates to mimic LOTOS "**exit**" and "`>>`" operators; the complexity is also quadratic (counting the number of offers passed to these gates) and the modelling remains not compositional.

*6.4. Beyond condition/actions models*

Close in essence to guarded commands, *condition-action* transition systems are often used in model checkers and academic lectures on automated verification. Such systems have states, variables, and transitions; a transition can be fired if its associated Boolean *condition* is true; firing a transition triggers its associated *action*, i.e., a set of assignments that modify variables.

Condition/action models have the merit of simplicity, but lead to transition systems larger than actually needed. For this reason, they have been sharply criticized in [105], where an alternative model named NTIF is proposed. LNT follows the principles of NTIF: in particular, it features blocks of sequential code (deterministic and nondeterministic assignments, conditionals, iterations) that execute *atomically*, i.e., without creating any visible or $\tau$-transition. This is implemented by the definition of sequential composition in LNT: for instance, each assignment creates one $\surd$-transition but a sequence of assignments only creates one single $\surd$-transition, as the LNT sequential composition operator merges two successive $\surd$-transitions in one.

The flexible style permitted by LNT can be illustrated on the small excerpt below, which was taken from a much larger specification. The structure of the code shows that what is needed in practice is far more involved than mere conditions/actions. For instance, the second "**select**" is used to enumerate a finite number of values from an infinite type domain — notice that there are no visible or $\tau$ actions at all in the branches of this choice. Also, the "**while**" loop corresponds to a statically unbounded number of assignments.

```
-- let x and x' be two lists of integers
x := {} ;
loop
  select
    g (?x') ; x := insert (x, x')
  []
    select
      x' := {}  []  x' := {0, 1}  []  x' := {1, 0, 2}
    end select ;
    g' (x') ;
    while x' ≠ {} loop
      x := remove (x, head(x')) ; x' := tail(x')
    end loop
  end select
end loop
```

## 7. Conclusion

### *7.1. Summary of Contributions*

Process calculi are the prime vehicle for disseminating concurrency theory knowledge to other branches of computer science and to industry as well. Questioning tradition and designing better languages is an essential part of this agenda. Sequential composition is often overlooked as an easy problem of secondary importance. But, as discussed in Sec. 3, the various approaches to sequential composition adopted by mainstream process calculi (namely, CCS, CSP, ACP, LOTOS, PSF, $\mu$CRL, and mCRL2) all suffer from serious shortcomings. A few other process calculi mentioned in Sec. 4.1 (namely, $\text{ACP}_\varepsilon$, $\text{ACP}_G$, ACBS&, Extended LOTOS, E-LOTOS, and OCCAM) tried to propose better approaches, but had little success and are almost forgotten.

Considering sequential composition as a central topic in the design of modern process calculi, the present article has presented a carefully-motivated solution, which features a unique sequential composition operator, together with "write-many" variables and static semantics data-flow constraints ensuring, among other suitable properties, that all variables are properly set before use.

The advantages of the proposed solution, which has been retained for the LNT language [30], are fourfold: (i) it is simple and easy to learn, as it matches the knowledge and intuition of software engineers; (ii) it allows to describe data and control using the same language (functions being a subset of processes) rather than two different sub-languages; (iii) it unifies and supersedes the various sequential composition operators that exist in traditional calculi, which can be "upgraded" to LNT using simple translation functions given in Sec. 5; (iv) it enables useful design patterns to be expressed in LNT with a linear complexity, where traditional process calculi require a quadratic complexity as shown in Sec. 6.

### *7.2. Implementation and Feedback*

From 2005 to 2014, the proposed solution has been fully implemented in the LNT→LOTOS translator that is part of the CADP toolbox [24]. Since 2011, LNT has been successfully taught at ENSIMAG (Grenoble INP), the first French engineering school in computer science, and appears significantly faster to learn than LOTOS: lectures can indeed focus on "important" operators (e.g., choice, parallel composition, and hiding) rather than spending time explaining the intricacies of action prefix and enable. Quoting Hoare's remark [60] about the design of the "`ALT`" operator in OCCAM, and hoping that the same remark also holds for sequential composition in LNT: "*as in all branches of engineering, the user of a product should never have to notice the skill with which the engineer has improved the design; true skill is that which conceals itself*".

LNT is also used in a growing number of case studies[10] and receives positive feedback from industry partners noticing major productivity boosts; quoting

[10] These case studies are too many to be exhaustively cited here; please refer to the CADP web site (`http://cadp.inria.fr`) or search for LNT-related publications in venues such

STMicroelectronics [106]: "*although modeling the DTD [Dynamic Task Dispatcher] in LOTOS is theoretically possible, using LNT made the development of a formal model practically feasible*". This reinforces our conviction that LNT is the right kind of approach for transferring concurrency theory results to industry.

### *7.3. Further Work*

We believe that long standing issues with sequential composition in value-passing process calculi are satisfactorily solved by the approach proposed in the present article, which also brings a new vision of how modern process calculi should be defined. From a language-design point of view, this approach could be expanded and refined into three directions:

- Data-flow constraints in the static semantics constraints of LNT could be enhanced, e.g., with abstract interpretation techniques, to accept more specifications (see Sec. 4.4) and also provide users with accurate warnings about questionable code fragments.
- One could try relaxing some constraints currently in the dynamic semantics of LNT, such as the prohibition of non-tail recursion (see Sec. 4.6) and of shared variables in parallel branches (see Sec. 4.5).
- Finally, LNT could be extended with quantitative aspects (time, probabilities, distributions, etc.); much has been done in Chi [77, 78, 79] and MoDeST [74, 75, 76], the latter having already symmetric sequential composition, bracketed $n$-ary operators, and write-many variables, but still with residual action prefix and syntactic constraints associated to probabilistic choice.

From a language-implementation perspective, verification techniques based on state-space exploration (namely, model checking and equivalence checking) are crucial topics, as no other technique today seems better capable of analyzing complex specifications of concurrent systems. It would therefore be worth designing a direct compiling scheme for LNT in order to replace the existing multi-step translation chain (*LNT* $\rightarrow$ *LOTOS* $\rightarrow$ *interpreted Petri nets* $\rightarrow$ *labelled transition systems*).

Keeping in mind that process calculi have originally been designed for proving concurrent systems formally [1], and putting aside the fact that state-space exploration techniques nowadays predominate over proof techniques for the verification of industrial systems, one may wonder whether the proposed approach to sequential composition is still compatible with the needs of formal proofs. Indeed, a large body of knowledge has been developed for value-passing calculi

as ATVA'13, CBSE'14, EICS'14, EICS'15, FMICS'13, FMICS'14, FORTE'13, FORTE'14, ICEFM'14, IFM'13, ISSE'13, PDP'15, SAC'14, SCICO'13, SCICO'14, TACAS'13, TACAS'15, VMCAI'15, etc.

with action prefix (e.g., [107, 108] [109, 110] [111] in the case of CCS) and one may wonder whether the same would be possible for LNT. We conjecture that the answer to this question is positive, for at least two reasons: (i) When languages with symmetric sequential composition and "write-many" variables can be translated to process calculi with action prefix (which is the case for LNT and CHP [73], both of which translate to LOTOS), one can reuse conventional proof techniques by applying them to the translated specifications; (ii) Prior work exists that gives algebraic semantics to forerunner languages sharing similar traits with LNT: in this respect, one can get inspiration from the algebraic laws for OCCAM [68, 60] and from the $\text{ACP}_G$ theory [50, 51, 52], the latter providing a sound and complete axiom system combining Hoare logic and process algebra. Anyway, formal specifications being a prerequisite for proofs, it is likely that proof techniques will adapt to match the style of specifications produced by system designers. Moreover, making process calculi closer to programming languages will enable reusing, for sequential processes at least, state-of-the-art verification approaches based on theorem proving and abstract interpretation.

*Acknowledgements*

This work benefited from scientific exchanges spanning over the last twenty years. Acknowledgements are due to the members of the ISO E-LOTOS committee (with particular thanks to Mihaela Sighireanu), to all colleagues at INRIA Grenoble who contributed to the design and implementation of LNT (especially Wendelin Serwe, who designed the translation algorithms for the control part and Frédéric Lang, who designed the translation algorithms for the data part and formalized the semantics of LNT), and to the users of LNT for their feedback (with a special mention to Bull and STMicroelectronics for their financial support). Comments from the anonymous reviewers, exchanges with Jan Friso Groote and Sjouke Mauw on $\text{ACP}_G$ and PSF, and discussions during the IFIP meeting on *Open Problems in Concurrency Theory* (Bertinoro, June 2014) and the SENSATION European project meeting (Aachen, September 2014) have been helpful in improving the present article.

*Bibliography*

[1] R. Milner, A Calculus of Communicating Systems, Vol. 92 of Lecture Notes in Computer Science, Springer, 1980.

[2] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

[3] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe, A Theory of Communicating Sequential Processes, J. ACM 31 (3) (1984) 560–599.

[4] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[5] A. W. Roscoe, Understanding Concurrent Systems, Texts in Computer Science, Springer, 2010.

[6] D. Austry, G. Boudol, Algèbre de Processus et Synchronisation, Theoretical Computer Science 30 (1984) 91–131.

[7] J. A. Bergstra, J. W. Klop, Algebra of Communicating Processes with Abstraction, Theoretical Computer Science 37 (1985) 77–121.

[8] J. Baeten, W. Weijland, Process Algebra, Vol. 18 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

[9] W. Fokkink, Introduction to Process Algebra, Texts in Theoretical Computer Science, Springer, 2000.

[10] ISO/IEC, LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva (Sep. 1989).

[11] T. Bolognesi, E. Brinksma, Introduction to the ISO Specification Language LOTOS, Computer Networks and ISDN Systems 14 (1) (1988) 25–59.

[12] S. Mauw, G. J. Veltink, An Introduction to PSFd, in: J. Díaz, F. Orejas (Eds.), Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'89), Barcelona, Spain, Vol. 352 of Lecture Notes in Computer Science, Springer, 1989, pp. 272–285.

[13] S. Mauw, G. J. Veltink, A Process Specification Formalism, Fundamenta Informaticae XIII (1990) 85–139.

[14] G. J. Veltink, PSF – A Retrospective, Fundamenta Informaticae 100 (1-4) (2010) 181–227.

[15] J. Groote, A. Ponse, The Syntax and Semantics of $\mu$CRL, CS-R 9076, Centrum voor Wiskunde en Informatica, Amsterdam (1990).

[16] J. Groote, A. Ponse, The Syntax and Semantics of $\mu$CRL, in: A. Ponse, C. Verhoef, S. Vlijmen (Eds.), Proceedings of the 1st Workshop on the Algebra of Communicating Processes (ACP'94), Utrecht, The Netherlands, Workshops in Computing Series, Springer, 1995, pp. 26–62.

[17] J. Groote, M. Mousavi, Modeling and Analysis of Communicating Systems, The MIT Press, 2014.

[18] R. Cleaveland, J. Parrow, B. Steffen, The Concurrency Workbench, in: J. Sifakis (Ed.), Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, Vol. 407 of Lecture Notes in Computer Science, Springer, 1989, pp. 24–37.

[19] R. Cleaveland, T. Li, S. Sims, The Concurrency Workbench of the New Century (Version 1.2) – User's Manual, State University of New York at Stony Brook (Jul. 2000).

[20] Formal Systems (Europe) Ltd, Oxford University Computing Laboratory, Failures-Divergence Refinement – FDR2 User Manual, 9th edition (Oct. 2010).

[21] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, A. W. Roscoe, FDR3 – A Modern Refinement Checker for CSP, in: E. Ábrahám, K. Havelund (Eds.), Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14), Grenoble, France, Vol. 8413 of Lecture Notes in Computer Science, Springer, 2014, pp. 187–201.

[22] J. Sun, Y. Liu, J. S. Dong, Model Checking CSP Revisited: Introducing a Process Analysis Toolkit, in: T. Margaria, B. Steffen (Eds.), Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'08), Porto Sani, Greece, Vol. 17 of Communications in Computer and Information Science, Springer, 2008, pp. 307–322.

[23] L. Shi, Y. Liu, J. Sun, J. S. Dong, G. Carvalho, An Analytical and Experimental Comparison of CSP Extensions and Tools, in: T. Aoki, K. Taguchi (Eds.), Proceedings of the 14th International Conference on Formal Engineering Methods (ICFEM'12), Kyoto, Japan, Vol. 7635 of Lecture Notes in Computer Science, Springer, 2012, pp. 381–397.

[24] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes, Springer International Journal on Software Tools for Technology Transfer (STTT) 15 (2) (2013) 89–107.

[25] D. Dams, J. F. Groote, Specification and Implementation of Components of a $\mu$CRL Toolbox, Technical Report Logic Group Preprint Series 152, Utrecht University (Dec. 1995).

[26] S. Cranen, J. Groote, J. Keiren, F. Stappers, E. de Vink, W. Wesselink, T. Willemse, An Overview of the mCRL2 Toolset and Its Recent Advances, in: N. Piterman, S. A. Smolka (Eds.), Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), Rome, Italy, Vol. 7795 of Lecture Notes in Computer Science, Springer, 2013, pp. 199–213.

[27] O. Sokolsky, Efficient Graph-based Algorithms for Model Checking in the Modal Mu-calculus, Ph.D. thesis, State University of New York at Stony Brook (May 1996).

[28] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat, FIACRE: An Intermediate Language for Model Verification in the TOPCASED Environment, in: J.-C. Laprie (Ed.), Proceedings of the 4th European Congress on Embedded Real-Time Software (ERTS'08), Toulouse, France, 2008.

[29] H. Garavel, Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular, in: C. Palamidessi, F. D. Valencia (Eds.), Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory, Ecole Polytechnique de Paris, France, November 13–15, 2006, Vol. 209 of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers, 2008, pp. 149–164, also available as INRIA Research Report RR-6368.

[30] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, G. Smeding, Reference Manual of the LNT to LOTOS Translator (Version 6.2), INRIA/VASY and INRIA/CONVECS, 130 pages (Mar. 2015).

[31] R. Milner, A Complete Inference System for a Class of Regular Behaviours, ournal of Computer and System Sciences 28 (3) (1984) 439–466.

[32] W. Janssen, M. Poel, J. Zwiers, Action Systems and Action Refinement in the Development of Parallel Systems – An Algebraic Approach, in: J. C. M. Baeten, J. F. Groote (Eds.), Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR'91), Amsterdam, The Netherlands, Vol. 527 of Lecture Notes in Computer Science, Springer, 1991, pp. 298–316.

[33] L. Aceto, M. Hennessy, Towards Action-Refinement in Process Algebras, Information and Computation 103 (2) (1993) 204–269.

[34] L. Aceto, M. Hennessy, Adding Action Refinement to a Finite Process Algebra, Information and Computation 115 (2) (1994) 179–247.

[35] U. Goltz, R. Gorrieri, A. Rensink, Comparing Syntactic and Semantic Action Refinement, Information and Computation 125 (2) (1996) 118–143.

[36] T. Gehrke, A. Rensink, Process Creation and Full Sequential Composition in a Name-passing Calculus, in: Proceedings of the 4th Workshop on Expressiveness in Concurrency (EXPRESS'97), Santa Margherita Ligure, Italy, Vol. 7 of Electronic Notes in Theoretical Computer Science, 1997, pp. 141–160.

[37] R. Gorrieri, A. Rensink, Action Refinement, in: J. Bergstra, A. Ponse, S. Smolka (Eds.), Handbook of Process Algebra, North-Holland, 2001, Ch. 16, pp. 1047–1147.

[38] E. Brinksma, On the Design of Extended LOTOS — A Specification Language for Open Distributed Systems, Ph.D. thesis, University of Twente (Nov. 1988).

[39] H. Garavel, A Wish List for the Behaviour Part of E-LOTOS, Input Document [LG5] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS, Liège, Belgium. Available from `http://vasy.inria.fr/elotos` (Dec. 1995).

[40] C. Koymans, J. Vrancken, Extending Process Algebra with the Empty Process, Logic Group Preprint Series, CIF Nr. 1, Department of Philosophy, Utrecht University (1985).

[41] J. Vrancken, The Algebra of Communicating Processes With Empty Process, Theoretical Computer Science 177 (2) (1997) 287–328.

[42] J. Bergstra, W. Fokkink, A. Ponse, Process Algebra with Recursive Operations, in: J. A. Bergstra, A. Ponse, S. A. Smolka (Eds.), Handbook of Process Algebra, North Holland, 2001, Ch. 5, pp. 333–389.

[43] J. A. Bergstra, J. W. Klop, The Algebra of Recursively Defined Processes and the Algebra of Regular Processes, in: J. Paredaens (Ed.), Proceedings of the 11th Colloquium on Automata, Languages and Programming (ICALP'84), Antwerp, Belgium,, Vol. 172 of Lecture Notes in Computer Science, Springer, 1984, pp. 82–94.

[44] J. C. M. Baeten, J. A. Bergstra, On Sequential Composition, Action Prefixes and Process Prefixes, Formal Aspects of Computing 6 (3) (1994) 250–268.

[45] H. Garavel, On the Introduction of Gate Typing in E-LOTOS, in: P. Dembinski, M. Sredniawa (Eds.), Proceedings of the 15th IFIP International Workshop on Protocol Specification, Testing and Verification (PSTV'95), Warsaw, Poland, Chapman & Hall, 1995, pp. 283–298.

[46] J. C. M. Baeten, R. J. van Glabbeek, Merge and Termination in Process Algebra, in: K. V. Nori (Ed.), Proceedings of the 7th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'87), Pune, India, Vol. 287 of Lecture Notes in Computer Science, Springer, 1987, pp. 153–172.

[47] J. F. Groote, F. W. Vaandrager, Structural Operational Semantics and Bisimulation as a Congruence (Extended Abstract), in: G. Ausiello, M. Dezani-Ciancaglini, S. Ronchi Della Rocca (Eds.), Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP'89), Stresa, Italy, Vol. 372 of Lecture Notes in Computer Science, Springer, 1989, pp. 423–438.

[48] B. Bloom, W. Fokkink, R. J. van Glabbeek, Precongruence Formats for Decorated Trace Preorders, in: Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00), Santa Barbara, California, USA, IEEE Computer Society, 2000, pp. 107–118.

[49] P. R. D'Argenio, C. Verhoef, A General Conservative Extension Theorem in Process Algebras with Inequalities, Theoretical Computer Science 177 (2) (1997) 351–380.

[50] J. F. Groote, A. Ponse, Process Algebra with Guards: Combining Hoare Logic with Process Algebra (Extended Abstract), in: J. C. M. Baeten, J. F. Groote (Eds.), Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR'91), Amsterdam, The Netherlands, Vol. 527 of Lecture Notes in Computer Science, Springer, 1991, pp. 235–249.

[51] J. F. Groote, A. Ponse, Process Algebra with Guards: Combining Hoare Logic with Process Algebra, Formal Aspects of Computing 6 (2) (1994) 115–164.

[52] A. Ponse, Process Algebra and Dynamic Logic, in: J. van Eijck, A. Visser (Eds.), Logic and Information Flow, MIT Press, 1994, pp. 125–148.

[53] J. C. M. Baeten, V. Bos, Formalizing Programming Variables in Process Algebra, TUCS Technical Report 493, Turku Centre for Computer Science, Turku, Finland (Dec. 2002).

[54] M. Weichert, Algebra of Broadcasting Systems: Value Passing, Sequential Composition, and Fork, in: U. H. Engberg, K. G. Larsen, P. D. Mosses (Eds.), Proceedings of the 6th Nordic Workshop on Programming Theory, Aarhus, Denmark, 1994, pp. 428–443, BRICS Note Series NS-94-6.

[55] ISO/IEC, Enhancements to LOTOS (E-LOTOS), International Standard 15437:2001, International Organization for Standardization — Information Technology, Geneva (Sep. 2001).

[56] H. Garavel, M. Sighireanu, French-Romanian Integrated Proposal for the User Language of E-LOTOS, Rapport SPECTRE 96-05, VERIMAG, Grenoble, Input Document [KC3] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Kansas City, Missouri, USA, May, 12–21, 1996 (May 1996).

[57] H. Garavel, M. Sighireanu, Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS, in: J.-F. Groote, B. Luttik, J. Wamel (Eds.), Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems (FMICS'98), Amsterdam, The Netherlands, CWI, Amsterdam, 1998, pp. 187–230, invited lecture.

[58] H. Garavel, F. Lang, R. Mateescu, Compiler Construction using LOTOS NT, in: N. Horspool (Ed.), Proceedings of the 11th International Conference on Compiler Construction (CC'02), Grenoble, France, Vol. 2304 of Lecture Notes in Computer Science, Springer, 2002, pp. 9–13.

[59] C. A. R. Hoare, Communicating Sequential Processes, Commun. ACM 21 (8) (1978) 666–677.

[60] C. A. R. Hoare, The Transputer and Occam: A Personal Story, Concurrency – Practice and Experience 3 (4) (1991) 249–264.

[61] D. May, OCCAM, SIGPLAN Notices 18 (4) (1983) 69–79.

[62] INMOS Limited, OCCAM 2 Reference Manual, International Series in Computer Science, Prentice-Hall, 1988.

[63] G. Jones, M. Goldsmith, Programming in occam2, Prentice-Hall, 1988, out of print; web edition available from http://www.cs.ox.ac.uk/geraint.jones/publications/book/Pio2.

[64] INMOS Limited, OCCAM 2.1 Reference Manual, SGS-THOMSON Microelectronics Ltd (May 1995).

[65] G. Barrett, OCCAM 3 Reference Manual, iNMOS Limited, Draft (Mar. 1992).

[66] A. W. Roscoe, Denotational Semantics for Occam, in: S. D. Brookes, A. W. Roscoe, G. Winskel (Eds.), Proceedings of the Seminar on Concurrency, Carnegie-Mellon University, Pittsburgh, PA, USA, Vol. 197 of Lecture Notes in Computer Science, Springer, 1984, pp. 306–329.

[67] M. H. Goldsmith, A. W. Roscoe, B. G. O. Scott, Denotational Semantics for Occam2, Technical Monography PRG-108, Oxford University Computing Laboratory (Jun. 1993).

[68] A. W. Roscoe, C. Hoare, The Laws of Occam Programming, Theoretical Computer Science 60 (1988) 177–229.

[69] Y. Gurevich, L. S. Moss, Algebraic Operational Semantics and Occam, in: E. Börger, H. K. Büning, M. M. Richter (Eds.), Proceedings of the 3rd Workshop on Computer Science Logic (CSL'89), Kaiserslautern, Germany, Vol. 440 of Lecture Notes in Computer Science, Springer, 1989, pp. 176–192.

[70] J. Camilleri, An Operational Semantics for Occam, International Journal of Parallel Programming 18 (5) (1989) 365–400.

[71] G. Barrett, The Semantics of Priority and Fairness in Occam, in: M. G. Main, A. Melton, M. W. Mislove, D. A. Schmidt (Eds.), Proceedings of 5th International Conference on the Mathematical Foundations of Programming Semantics (MFPS'89), Tulane University, New Orleans, LA, USA, Vol. 442 of Lecture Notes in Computer Science, Springer, 1989, pp. 194–208.

[72] A. J. Martin, Compiling Communicating Processes into Delay-Insensitive VLSI Circuits, Distributed Computing 1 (4) (1986) 226–234.

[73] H. Garavel, G. Salaün, W. Serwe, On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP, Science of Computer Programming 74 (3) (2009) 100–127.

[74] P. R. D'Argenio, H. Hermanns, J. Katoen, R. Klaren, MoDeST – A Modelling and Description Language for Stochastic Timed Systems, in: Proceedings of the Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV'01), Aachen, Germany, 2001, pp. 87–104.

[75] H. Bohnenkamp, Pedro R. d'Argenio, H. Hermanns, J.-P. Katoen, MoDeST: A Compositional Modeling Formalism for Real-Time and Stochastic Systems, IEEE Transactions on Software Engineering 32 (10) (2006) 812–830.

[76] E. Hahn, A. Hartmanns, H. Hermanns, J. Katoen, A Compositional Modelling and Analysis Framework for Stochastic Hybrid Systems, Formal Methods in System Design 43 (2) (2013) 191–232.

[77] V. Bos, J. J. T. Kleijn, Redesign of a Systems Engineering Language: Formalisation of $\chi$, Formal Aspects of Computing 15 (4) (2003) 370–389.

[78] R. R. H. Schiffelers, D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, Formal Semantics of Hybrid Chi, in: K. G. Larsen, P. Niebert (Eds.), Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS'03), Marseille, France, Vol. 2791 of Lecture Notes in Computer Science, Springer, 2003, pp. 151–165.

[79] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, R. R. H. Schiffelers, Syntax and Consistent Equation Semantics of Hybrid Chi, Journal of Logic and Algebraic Programming 68 (1-2) (2006) 129–210.

[80] A. Fehnker, R. Glabbeek, P. Höfner, A. McIver, M. Portmann, W. L. Tan, A Process Algebra for Wireless Mesh Networks, in: H. Seidl (Ed.), Proceedings of the 21st European Symposium on Programming (ESOP'12),Tallinn, Estonia, Vol. 7211 of Lecture Notes in Computer Science, Springer, 2012, pp. 295–315.

[81] J. Parrow, B. Victor, The Update Calculus (Extended Abstract), in: M. Johnson (Ed.), Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97), Sydney, Australia, Vol. 1349 of Lecture Notes in Computer Science, Springer, 1997, pp. 409–423.

[82] D. T. Ross, Uniform Referents: An Essential Property for a Software Engineering Language, in: J. Tou (Ed.), Software Engineering, Vol. 1, Academic Press, 1970, pp. 91–101.

[83] V. Bos, J. J. T. Kleijn, Formal Specification and Analysis of Industrial Systems, Ph.D. thesis, Eindhoven University of Technology (Mar. 2002).

[84] H. Garavel, M. Sighireanu, On the Introduction of Exceptions in LOTOS, in: R. Gotzhein, J. Bredereke (Eds.), Proceedings of the IFIP

Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'96), Kaiserslautern, Germany, Chapman & Hall, 1996, pp. 469–484.

[85] R. E. Strom, S. Yemini, Typestate: A Programming Language Concept for Enhancing Software Reliability, IEEE Transactions on Software Engineering 12 (1) (1986) 157–171.

[86] H. Garavel, J. Sifakis, Compilation and Verification of LOTOS Specifications, in: L. Logrippo, R. L. Probert, H. Ural (Eds.), Proceedings of the 10th IFIP International Symposium on Protocol Specification, Testing and Verification (PSTV'90), Ottawa, Canada, North-Holland, 1990, pp. 379–394.

[87] J. F. Groote, A. Ponse, Y. S. Usenko, Linearization in Parallel pCRL, Journal of Logic and Algebraic Programming 48 (1–2) (2001) 39–70.

[88] B. Berthomieu, T. Le Sergent, Programming with Behaviors in an ML Framework – The Syntax and Semantics of LCS, in: D. Sannella (Ed.), Proceedings of the 5th European Symposium on Programming Languages and Systems (ESOP'94), Edinburgh, U.K, Vol. 788 of Lecture Notes in Computer Science, Springer, 1994, pp. 89–104.

[89] A. W. Roscoe, C. Hoare, R. Bird, The Theory and Practice of Concurrency, Prentice Hall, 1997.

[90] H. Garavel, M. Sighireanu, A Graphical Parallel Composition Operator for Process Algebras, in: J. Wu, Q. Gao, S. T. Chanson (Eds.), Proceedings of the IFIP Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'99), Beijing, China, Kluwer Academic Publishers, 1999, pp. 185–202.

[91] G. Plotkin, A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark (1981).

[92] G. D. Plotkin, A Structural Approach to Operational Semantics, Journal of Logic and Algebraic Programming 60–61 (2004) 17–139.

[93] G. Plotkin, An Operational Semantics for CSP, in: A. Salwicki (Ed.), Proceedings of the 1980 Conference on Logics of Programs and Their Applications, Poznan, Poland, Vol. 148 of Lecture Notes in Computer Science, Springer, 1983, pp. 250–252, extended version available from http://homepages.inf.ed.ac.uk/gdp/publications/An_Op_Sem_CSP.pdf.

[94] G. D. Plotkin, The Origins of Structural Operational Semantics, Journal of Logic and Algebraic Programming 60–61 (2004) 3–15.

[95] B. Bloom, F. Vaandrager, SOS Rule Formats for Parameterized and State-Bearing Processes, Unpublished manuscript available from http://www.cs.ru.nl/ita/publications/papers/fvaan/bardfrits.ps (Jul. 1994).

[96] M. Mousavi, M. A. Reniers, J. Groote, Congruence for SOS with Data, in: Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS'04), Turku, Finland, IEEE Computer Society, 2004, pp. 303–312.

[97] M. Mousavi, M. A. Reniers, J. Groote, Notions of Bisimulation and Congruence Formats for SOS with Data, Information and Computation 200 (1) (2005) 107–147.

[98] D. Gebler, E. Goriac, M. Mousavi, Algebraic Meta-Theory of Processes with Data, in: J. Borgström, B. Luttik (Eds.), Proceedings of the Combined 20th International Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics (EXPRESS/SOS'13), Buenos Aires, Argentina, Vol. 120 of Electronic Proceedings in Theoretical Computer Science, 2013, pp. 63–77.

[99] R. De Nicola, A. Labella, A Completeness Theorem for Nondeterministic Kleene Algebras, in: I. Prívara, B. Rovan, P. Ruzicka (Eds.), Proceedings of the 19th International Symposium on Mathematical Foundations of Computer Science (MFCS'94), Kosice, Slovakia, Vol. 841 of Lecture Notes in Computer Science, Springer, 1994, pp. 536–545.

[100] F. Corradini, R. De Nicola, A. Labella, Fully Abstract Models for Nondeterministic Regular Expressions, in: I. Lee, S. A. Smolka (Eds.), Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95), Philadelphia, PA, USA, Vol. 962 of Lecture Notes in Computer Science, Springer, 1995, pp. 130–144.

[101] F. Corradini, R. De Nicola, A. Labella, Models of Nondeterministic Regular Expressions, Journal of Computer and System Sciences 59 (3) (1999) 412–449.

[102] J. C. M. Baeten, F. Corradini, Regular Expressions in Process Algebra, in: Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS'05), Chicago, IL, USA, IEEE Computer Society, 2005, pp. 12–19.

[103] J. C. M. Baeten, F. Corradini, C. Grabmayer, A Characterization of Regular Expressions Under Bisimulation, Journal of the ACM 54 (2).

[104] E. W. Dijkstra, Guarded Commands, Non-determinacy and Formal Derivation of Programs, Communication of the ACM 18 (8) (1975) 453–457.

[105] H. Garavel, F. Lang, NTIF: A General Symbolic Model for Communicating Sequential Processes with Data, in: D. Peled, M. Vardi (Eds.), Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal

Techniques for Networked and Distributed Systems (FORTE'02), Houston, TX, USA, Vol. 2529 of Lecture Notes in Computer Science, Springer, 2002, pp. 276–291, full version available as INRIA Research Report RR-4666.

[106] E. Lantreibecq, W. Serwe, Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit Using CADP, in: G. Salaün, B. Schätz (Eds.), Proceedings of the 16th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'11), Trento, Italy, Vol. 6959 of Lecture Notes in Computer Science, Springer, 2011, pp. 180–195.

[107] M. Hennessy, A Proof System for Communicating Processes with Value-passing (Extended Abstract), in: C. E. V. Madhavan (Ed.), Proceedings of the 9th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'89), Bangalore, India, Vol. 405 of Lecture Notes in Computer Science, Springer, 1989, pp. 325–339.

[108] M. Hennessy, A Proof Sustem for Communicating Processes with Value-Passing, Formal Aspects of Computing 3 (4) (1991) 346–366.

[109] M. Hennessy, A. Ingólfsdóttir, A Theory of Communicating Processes with Value-Passing, in: M. Paterson (Ed.), Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90), Warwick University, England, Vol. 443 of Lecture Notes in Computer Science, Springer, 1990, pp. 209–219.

[110] M. Hennessy, A. Ingólfsdóttir, A Theory of Communicating Processes with Value Passing, Information and Computation 107 (2) (1993) 202–236.

[111] M. Hennessy, H. Lin, Proof Systems for Message-Passing Process Algebras, Formal Aspects of Computing 8 (4) (1996) 379–407.