# Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS

Hubert Garavel and Mihaela Sighireanu
INRIA Rhône-Alpes and DYADE / VASY group
655, avenue de l'Europe
38330 Montbonnot St Martin
France
Tel: +(33) 4 76 61 52 24 Fax: +(33) 4 76 61 52 52
E-mail: {hubert.garavel,mihaela.sighireanu}@inria.fr
Web: http://www.inrialpes.fr/vasy

**Abstract**

Process algebras are often advocated as suitable formalisms for the specification of telecommunication protocols and distributed systems. However, despite their mathematical basis, despite standardization attempts (most notably the Formal Description Technique LOTOS), and despite an ever growing number of successful case-studies, process algebras have not yet reached a wide acceptance in industry.

On the other hand, description languages such as PROMELA or SDL are quite popular, although they lack a formal semantics, which should prohibit their use for safety-critical systems.

In this paper, we seek to merge the "best of both worlds" by attempting to define a "second generation Formal Description Technique" that would combine the strong theoretical foundations of process algebras with language features suitable for a wider industrial dissemination of formal methods. Taking the international standard LOTOS as a basis, we suggest several enhancements, which fall into three categories: data part, behaviour part, and modules.

Our work was initiated in 1992 in the framework of the ISO/IEC Committee for the revision of the LOTOS standard. Several of our suggestions have been accepted and will be integrated into the revised standard E-LOTOS. The other suggestions are considered in the context of LOTOS NT, a variant of E-LOTOS for which a prototype compiler/model-checker is under development at INRIA.

# Introduction

Originally intended for the theoretical study of concurrency, process algebras can also be used for the formal description of communication protocols and distributed systems. The word "process algebras" encompasses pure mathematical calculi (e.g., Acp [BK84], Ccs [Mil80, Mil89], Csp [Hoa85, BHR84], Sccs [dS85]) as well as more elaborate languages (such as Fdr [For97], Lotos [ISO88b, BB88], $\mu$Crl [GP95], Occam [Cam89]), which provide additional features intended for the description of real systems (e.g., user-defined data types and modules).

Semantically speaking, process algebras have strong advantages: expressiveness, compositionality, formal semantics defined in terms of Labelled Transition Systems (Lts) [Par81] using structural operational semantics rules [Plo81, GV92], verification algorithms based on behavioural equivalences and preorders, refinement methods, etc. Process algebras have been used successfully many times to model the behaviour of real systems. In addition, simulators, model-checkers, and theorem-provers are available for analyzing process algebraic descriptions.

However, most of these efforts take place in the academic context. Even when applied to industrial problems, process algebras remain a formalism to be used by experts rather than average computer industry engineers. This is mostly due to the fact that process algebras require a substantial learning effort for non-specialists, which strongly limits their potential for industrial take-up.

On the one hand, the complexity of process algebras is intrinsic and can not be reduced: for instance, the concepts of non-determinism, concurrency, bisimulation, etc. are primitive ones and they can not be avoided when asynchronous systems have to be modelled accurately.

On the other hand, the "steep learning curve" of process algebras is mostly a consequence of wrong design choices. Taking argument of their mathematical origins, process algebras often neglected to integrate certain features considered as standard in most computer languages, so that beginners would have trouble when faced to very simple situations. For instance, most of the aforementioned process algebras lack an "**if-then-else**" operator, as well as an "**array**" type constructor. Although one can argue that alternative solutions can be used (e.g., guarded commands to express conditionals, abstract data types to simulate array types, etc.), it should be understood that engineers in industry have little time for similar subtleties; they tend to be productive as fast as possible and thus are reluctant to new approaches for reinventing the wheel.

In this respect, it is worth noticing that some other languages (e.g., Promela [Hol91], Sdl [IT92]) seem to find a faster acceptance. Technically, these languages are often behind process algebras: they lack a formal semantics, they have limitations for expressing concepts such as multi-way synchronization, non-determinism, internal transitions, etc. Among other reasons, their relative success is certainly due to features borrowed from imperative programming languages (Promela is close to the C language) or graphical languages (Sdl diagrams are close to flow charts). Building on standard practice enables beginners to capitalize on former knowledge when learning new formalisms.

Based upon our experience in applying, teaching, and developing tools for the first generation of description languages (including Estelle and Lotos), we believe that better languages could be designed, which would combine features from process algebras together with features taken from classical programming languages. As the part of safety-critical missions devoted to computer hardware and/or software is continuously increasing, there is a need for a second generation of Formal Description Techniques, provided that this new generation includes wide dissemination and industrial acceptance in its objectives.

We started to work on this topic in 1992, within the framework of an international standardization project (Iso/Iec Jtc1/Sc21/Wg7 New Work Item on Enhancements to Lotos, which recently moved to Iso/Iec Jtc1/Sc33/Wg9). This standardization project aims at producing a revised version of the Lotos standard. The revised version should include new features suitable for increasing both the expressiveness and user-friendliness of the language.

There have been numerous proposals for improving Lotos, some of which are anterior to the E-Lotos project, e.g., [Sco86, Bri88, Sto91, LL91]. Many proposals suggest enhancements to the data part of Lotos [Pec93, BL95, RdMS95, JGL$^+$95, Pec96] while other focus on the behaviour part [QA92, Gar95a, GS96], including the introduction of time in Lotos [Leo97, LL97, Her98], etc.

This paper presents our current views about the enhancement of Lotos. Due to lack of space, it was not possible to give an exhaustive technical definition[1]. Instead, we preferred to highlight the major concepts, design choices, and pending issues, and to illustrate them with examples.

Most of the ideas expressed in this paper have been submitted for standardization; some of them are integrated into the current version of E-Lotos [Que98]. Additionally, we are working on a variant of E-Lotos named Lotos NT[2] for which a prototype compiler/model-checker is under development at Inria Rhône-Alpes. The language features presented below belong to Lotos NT; many of them belong to E-Lotos as well.

The organization of this paper follows the structure of Lotos, which consists of two orthogonal parts: data and behaviours. Section 1 presents the data part of the language, including types, functions, and expressions. Section 2 introduces the main concepts of the behaviour part: gates, behaviours, and processes. E-Lotos and Lotos NT improve over standard Lotos by adding structuring capabilities (modules and interfaces) detailed in Section 3.

[1] The reader may find a chronological list of our proposals and other E-Lotos-related information at the Inria/Vasy Web site `http://www.inrialpes.fr/vasy/elotos`

[2] NT stands here for "New Technology"

# 1 Data part

A Formal Description Technique must be equipped with a sub-language for describing and handling data, e.g., the types and values of messages exchanged by a protocol. In the sequel, this sub-language will be called the *data part*. In practice, the data part can be either *algebraic* (e.g., Lotos, $\mu$Crl, and Sdl, which rely on abstract data types, especially ActOne), *functional* (e.g., Fdr [For97] or Lcs [BL94], which are based on the functional language Ml), or *imperative* (e.g., Estelle [ISO88a] based on Pascal, or Promela and Murphi based on C). Each approach has its own problems and limitations.

The inclusion of ActOne abstract data types in Lotos has been often criticized by both users and implementors [Mun91, JGL+95]. The most common problems are:

- Abstract data type definitions are too verbose and error-prone. For instance, defining the equality predicate on an enumerated type with $n$ values usually requires $n^2$ equations (a more elaborate definition can reduce this number to $n$ equations, which is still too much).
- Basic data types (e.g., enumerations, records, lists, etc.) and their associated operations have to be redefined again every time, and the standard library is not effective in cutting down this effort.
- Axiomatic definitions raise completeness and consistency issues: when specifying data types, users are never sure that they have written "enough" equations, nor that the equations do not contradict each other.
- The initial semantics approach is not appropriate for modelling partial functions (e.g., dividing by zero, accessing the head of an empty list, etc.), for which an exception mechanism is advisable.
- Equivalence of terms for an arbitrary algebra is undecidable. Thus, it is impossible to provide sound and complete Lotos simulators. In practice, users and implementors tend to use rewrite rules and pattern-matching style instead of the standard equational semantics (see, for instance, [Gar89]). However, different implementations may produce different results depending on the strategy used to turn equations into rewrite rules.
- Finally, ActOne definitions lack a proper structuring, as nothing prevents conceptually-related sorts, operations and equations from being scattered in many different places. In particular, this leads to the *persistency* issue discussed in Section 3.

Producing correct specifications requires a substantial theoretical background (e.g., initial semantics, quotient algebras, etc.) that is out of proportion with respect to the practical goal (e.g., specifying the messages exchanged by a protocol). For protocol specifiers as well as tool builders, abstract data types are often perceived as a problem rather than a solution.

There have been many attempts at tackling these problems. The algebraic specification community has proposed refined languages, e.g., Lpg [BE86] and Opal [Pep94, FGGP93, Exn94], which are more user-friendly and better implementation-oriented.

Within the Lotos community, improving the data part has also been a constant concern [Sco86, Bri88, LL91, Sto91, Pec93, BL95, RdMS95, Pec96]. After a detailed discussion, the E-Lotos standardization Committee concluded that, in spite of their formal semantics and high abstraction level, algebraic specifications were not suitable, and looked for a data specification formalism that could replace ActOne. To achieve symmetry with the behaviour part, which allows an operational definition of processes, this replacement formalism had to support constructive definitions of data types and associated functions.

Functional languages were considered first, and most notably Sml, which solved a lot of problems faced by ActOne users. However, it appeared that Sml was not entirely suitable. One the one hand, it was considered too complex: certain features of Sml (such as polymorphism and higher-order functions) would have been of little use for protocol descriptions, although they would have made tool implementations certainly more difficult. On the other hand, the purely functional approach had several limitations:

- As E-Lotos was intended for the design of distributed systems, some compatibility with Idl [ISO96] (the language used to specify the interfaces of Corba and Odp systems) was desirable. More precisely, Idl and E-Lotos had to be used jointly, the former to specify the interfaces, the latter the behaviours of distributed objects. In Idl, the methods permitted on a given object are specified as functions taking "**in**", "**out**", and/or "**in/out**" parameters, returning a result, and (possibly) raising some exceptions. The existence in Idl of "**out**" and "**in/out**" clashed with the approach taken in Sml, where functions have only "**in**" parameters and only a single result (possibly a tuple if several values have to be returned). It was decided that E-Lotos functions should have "**out**" parameters to achieve compatibility not only with Idl, but also with external libraries written using algorithmic languages (such as C, Ada, Java, etc.), which would have been difficult to interface from a purely functional language.

- The introduction of "**out**" parameters led to question the functional approach more thoroughly. The need for assignment (at least to give a return value to "**out**" parameters) and iterative constructs ("**while**" and "**for**" loops) was acknowledged. It was felt that a strictly functional approach would limit the acceptance and dissemination of E-Lotos in the same way as ActOne did for Lotos: because functional languages are not so common in industry, an extra-training effort would be needed before users could start producing E-Lotos descriptions.

The solution retained for the data part of E-Lotos is a compromise: the syntax looks imperative (explicit assignment, sequential composition operator, etc.), but the semantics remains functional (in the sense that there is no concept of "store": special substitutions are used instead).

In most cases, both the imperative and functional interpretations yield the same result for a given fragment of code. However, there are tricky situations (especially with language constructs having multiple exits, such as exception handling) in which the result given by the functional semantics is not the same that one would expect according to an imperative interpretation. Additionally, defining the semantics in terms of substitutions creates awkward problems, for instance with "**in/out**" parameters (not supported in E-Lotos) and loops (non-standard in E-Lotos).

To avoid these problems, we decided to explore a different solution by designing a language named Lotos NT with a truly-imperative semantics. We believe that an imperative language should be easier to learn by engineers and easier to implement by a translation to existing programming languages (especially, C or Java). We outline below the main design choices for the data part of Lotos NT.

The data part of a Lotos NT description is a collection of type and function definitions (which can be structured in modules, as explained in Section 3).

Lotos NT is a strongly typed language. Type-checking is done fully at compile-time (contrary to E-Lotos which requires some type-checking at run-time). Types are defined constructively by giving the list of their constructors (in the same way as "datatypes" in Sml). For instance:

```
type PACKET is
   PACKET (DATA:STRING, CRC:INTEGER)
end type

type PACKET_LIST is
   EMPTY,
   CONS (HEAD:PACKET, TAIL:PACKET_LIST)
end type PACKET_LIST
```

Type definitions can be recursive directly or transitively. Semantically, a type represent the set of closed, well-typed terms that can be generated by its (free) constructors. For each type defined, an equality relation (expressing the syntactic equivalence of terms) and its negation are derived automatically; these relations are needed in the behaviour part to define the semantics of value-passing rendezvous.

The arguments of each constructor are named in type definitions. This serves several purposes. First, argument names can be used in constructor invocations, as in Ada: for instance, the expression "`PACKET ("ABC", 3)`" can also be written "`PACKET (CRC -> 3, DATA -> "ABC")`". Second, field names implicitly define *projections*: if `P` is an expression of type `PACKET`, then "`P.DATA`" will select the first field of `P`. Third, they also define *updaters*: if `P` is an expression of type `PACKET`, then

"P.{CRC := 0}" denotes the value derived from P by setting to zero the field named CRC.

These "constructive" type definitions have other advantages:

- They are aligned with the various proposals for improving Lotos data types, almost all of which recommend to distinguish between constructors and defined functions.
- Lotos sort definitions involving only free constructors can easily be translated into constructive type definitions.
- The constructors of a given type are grouped together in the definition of the type, instead of being scattered anywhere, as it might be the case in Lotos descriptions.
- They allow to declare unbounded data structures (e.g., lists, trees, etc.) without handling pointers or union discriminants explicitly. By doing so, one avoids all the problems related to pointers (e.g., dangling references, incorrect pointer deallocations, etc.) and discriminated unions (e.g., accessing a field that is not available with the current value of the discriminant, modifying the discriminant separately from the fields, thus subverting the type-checking system, etc.).

A function is defined in Lotos NT in the same way as in algorithmic languages, by specifying its name, the names, types, and modes ("**in**", "**out**", "**in/out**") of its arguments, the type of its result (if any: a function without result is like a "procedure"), and the exceptions that it can raise. For instance:

```
function ** (X:REAL, N:INTEGER) : REAL is
   ...
end function

function PARTITION (in L:PACKET_LIST, out LEFT, RIGHT:PACKET_LIST) is
   ...
end function

function INVERT (inout M:MATRIX) raises NULL_DISCRIMINANT is
   ...
end function INVERT
```

The data part of Lotos NT provides various user-friendly features, several of them being absent in Lotos: there are built-in notations for integers, reals, characters, and strings; function names can be overloaded; user-defined functions can have mathematical names (e.g., "**" in the above example); additionally, the usual arithmetic, boolean and relational operators (e.g., "+", "and", "<=") are recognized as special cases with built-in priorities; functions with two arguments can be called either with prefix or infix notations (e.g., "`mod (X, Y)`" or "`X mod Y`"); argument lists in function invocations can be either positional or named (i.e., "`F (0, 1)`" or "`F (X->0, Y->1)`").

As most algorithmic programming languages, Lotos NT establishes a distinction between *expressions* and *instructions* (also called "statements"). This is an essential difference with E-Lotos, which has only expressions.

In Lotos NT, expressions are very simple: they are built using constants, variables, constructor and function invocations. An expression can not have side-effects, e.g., assigning a variable. The evaluation of an expression is deterministic (in identical contexts, it always yields the same result or raises the same exception) and instantaneous (evaluating an expression takes no time).

Instructions are more complex: they include variable assignment, sequential composition, "**if**" and "**case**" conditionals, "**for**" and "**while**" loops, exception raise and trap, etc. As for expressions, the execution of an instruction is always deterministic (in identical contexts, it always assigns the same variables with the same values and terminates with the same exception) and instantaneous. The body of a function is always an instruction. For instance:

```
function FIRST_PACKET (L: PACKET_LIST) : PACKET raises EMPTY_LIST is
   case L is
      NIL -> raise EMPTY_LIST
      CONS (P:PACKET, any) -> return P
   end case
end function FIRST_PACKET

function LIST_LENGTH (L: PACKET_LIST) : INTEGER is
   var N:INTEGER := 0 in
      while L <> NIL loop
         N += 1;
         L := L.TAIL
      end loop;
      return N
end function LIST_LENGTH
```

However, such an imperative approach has potential problems that must be addressed in order to have a well-defined formal semantics.

As mentioned above, problems related to discriminated unions are avoided by the use of constructive types and pattern-matching "**case**" statements.

Another problem remains, as variables can be used before they have been assigned. To get rid of this issue, Lotos NT follows the solution adopted by Hermes [SBL+91] and later by Java [Sun96, chapter 16]: a Lotos NT program is legal iff one can prove at compile-time that each variable will be assigned before used. This is done by extending the static semantics with additional checks expressing sufficient syntactic conditions on the data flow to rule out (potentially) unsafe programs. For instance, the following function definition will be rejected because the value of `Y` is not assigned when `X` is negative.

```
function F (in X:INTEGER, out Y:INTEGER) is
   if X >= 0 then
      Y := SQRT (X)
   end if
end function
```

We can also mention two important facilities offered by Lotos NT:

- Lotos NT supports *predicative subtyping*: one can define subtypes of a given type (called the "base type") by specifying a boolean predicate that restricts the domain of the base type. In the example below, `POSITIVE` is a subtype of the `INTEGER` base type:

  ```
  type POSITIVE is { N:INTEGER | N > 0 } end type
  ```

  Predicative subtyping allows a more precise description of data domains by adding restrictions to the domains generated by free constructors. The base type and the subtype are considered to be different types: conversion from the base type to the subtype (and vice-versa) must be done explicitly using (automatically generated) conversion functions that can raise an exception if the predicate associated to the subtype is not satisfied. Implicit conversions would not be possible due to the presence of overloading (this would make type-checking non-polynomial). The constructors of the subtype are the same as the constructors of its base type, except that the base type is replaced with the subtype in their profiles.

- In addition to its predefined type library (booleans, integers, etc.), Lotos NT provides syntactic facilities for a compact definition of usual types and functions. These syntactic notations are automatically expanded into type and function definitions, in the same way as [BM79, Pec93, Pec96]. For instance, the following type declaration defines an enumerated type equipped with an order relation:

  ```
  type STATUS is
     enum
        NONE, PENDING, INVOICED
       with >, ORD, SUCC, PRED
  end type STATUS
  ```

  Similar shorthand notations exist for records, arrays, lists, sets, etc.

## 2 Behaviour part

Although the behaviour part of Lotos is far superior to the data part, it is not fully satisfactory. In particular, two features are missing:

- Lotos does not allow gate typing. Although gates define interface points between a behaviour and its environment, it is not possible to specify the types of messages exchanged via a gate, nor their direction (sent or received), thus limiting the possibilities of type checking at compile-time (a typing error in the messages exchanged can only be detected because a deadlock occurs at run-time). This is a common concern [Ard97] for which solutions have been proposed [Gar95b].

- Lotos has no concept of quantitative time (i.e., delays, timeouts) nor urgency, which prevents it from being used for the description of timed systems, including real-time systems. There have been several proposals for introducing time in E-Lotos, e.g., Rt-Lotos [Cd95] and Et-Lotos [Leo97, LL97, Her98].

Initially, the E-Lotos Committee mainly focused on the two issues above, with the idea of minimizing changes to the behaviour part. However, as the data part evolved towards a functional/imperative language, it became clear that deeper changes were needed to align the behaviour part with the data part. The Committee agreed that a maximal symmetry between the data and behaviour parts would be highly desirable. By doing so, some defects of the behaviour part, which had been underestimated so far, became more apparent:

- For beginners, sequential composition is probably the most confusing aspect in the behaviour part of Lotos. Following the ideas of Ccs and Csp, Lotos has two different sequential composition operators: the *action prefix* operator and the *enabling* operator, both of which obey to different rules. Action prefix is asymmetric and passes to its right argument all the variables bound in its left argument, whereas enabling is symmetric, creates an internal event "**i**", and passes to its right argument only the variables declared in its "**accept**" clause and bound to the values returned by its left argument using "**exit**" statements. For instance, naive users have trouble in understanding why behaviour expressions such as

  ```
  (G1 ?X:NAT [] G2 ?X:NAT); H !X
  ```

  or:

  ```
  (G1 ?X1:NAT ||| G2 ?X2:NAT); H !(X1 + X2)
  ```

  are not allowed, and should be written instead:

```
G1 ?X:NAT; H !X; stop [] G2 ?X:NAT; H !X; stop
```

and:

```
(G1 ?X1:NAT; exit (X1, any NAT)
 |||
 G2 ?X2:NAT; exit (any NAT, X2))
>> accept X1, X2:NAT in H !(X1 + X2); stop
```

This problem was solved by replacing action prefix and enabling with a single sequential composition operator noted "`;`" (as in Acp). A neutral element (noted "`null`") for this operator was introduced.

- Lotos only provides guarded commands to express conditionals, which is tedious, error prone, and leads to inefficient implementations (as boolean conditions in mutually exclusive branches have to be evaluated twice or even more). This problem was solved by introducing "**if**" and "**case**" statements.
- Lotos has no iterative statements: recursive processes have to be used instead, which affects the readability of the control flow and requires the introduction of auxiliary processes. This problem was solved by introducing "**for**" and "**while**" loops.
- The need for specifying the abrupt termination of a given behaviour was acknowledged. For instance, it may be necessary to express the situation in which a parallel composition is terminated by one of its components: this feature is called *non-synchronized termination* as opposed to Lotos *synchronized termination*, which requires that a parallel composition can only terminate if all components synchronize on an "**exit**" action [QA92]. It was shown that non-synchronized termination could be addressed as a special case of exception handling [GS96].

The changes in the data part impacted the behaviour parts in several other places. E-Lotos processes received "**out**" parameters (Lotos NT even allows "**in/out**" parameters). Consequently, explicit variable assignment was introduced in the behaviour part of E-Lotos. This change was a significant improvement over Lotos, as it allows a direct modelling of "state variables". In Lotos, state variables have to be modelled as formal parameters of recursive processes and can only be assigned by parameter passing; to describe behaviours in which there is, conceptually, a single state variable, Lotos require several variables to be declared (especially if there are several mutually recursive processes). Multiplying the variables as such creates several shortcomings: the description becomes less readable, because the original intention of the specifier is lost; debugging by simulation becomes more difficult, because instead of tracing the value of a single state variable, several variables have to be inspected, only one of which is active in the current state; it makes model-checking verification harder, because it is not possible to express invariant properties on state variables and because multiple variables contribute to state explosion (unless the compiler is smart enough to allocate different instances of the same variable in the same memory location, a non-polynomial problem for which there is generally no unique solution).

To summarize, we are convinced that the functional-like style used in Lotos and other process algebras is inappropriate practically. In fact, this style has no other purpose than ensuring that each variable is properly assigned before used, and that concurrent behaviours do not communicate using shared variables. With the imperative style of Lotos NT, the same guarantees are obtained by adding data flow restrictions to the static semantics, e.g., by checking that if a variable is assigned in some component of a parallel composition, the other components can neither read nor write to this variable.

All these changes lead to a process description style very different from the Lotos style: descriptions are often shorter and more readable. For instance:

```
process BUFFER [in INPUT:PACKET, out OUTPUT:PACKET] is
   loop
      INPUT ?P:PACKET;
      OUTPUT !P
   end loop
end process BUFFER

process QUERY [in INPUT:INTEGER]
              (in X1, X2, X3:INTEGER, out Y1, Y2, Y3:INTEGER) is
   if X1 <> 0 then Y1 := X1 else INPUT ?Y1 end if;
   if X2 <> 0 then Y2 := X2 else INPUT ?Y2 end if;
   if X3 <> 0 then Y3 := X3 else INPUT ?Y3 end if
end process QUERY
```

It is worth noticing that the definition of process `QUERY` would be harder and less readable in process algebras based on action-prefix.

After the sequential operators, the wind of reform reached the parallel operators. Although multi-way rendezvous and parallel composition are far superior in Lotos than in Ccs and Csp, they are still perfectible. The E-Lotos Committee agreed on three major improvements:

- A parallel composition iterating over a finite set of values, or a finite type was introduced, as suggested in [Bri88]. For instance:

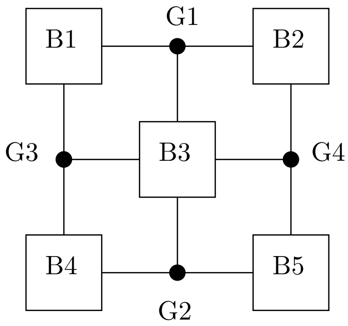
  ```
  par X:T || B (X) end par
  ```

starts several concurrent instances of the behaviour `B` parameterized by variable `X`, one instance for each value of type `T`. This operator is practically useful since it allows to describe networks of parameterized processes concisely (without using process recursion through parallel composition).

Such an operator does not exist in Lotos because, in the initial algebra framework, one can not decide in general whether a given type will be finite or not. Fortunately, this problem becomes decidable when types are defined by free constructors as in E-Lotos: a type is finite iff its definition is not recursive and if the types of the arguments of its constructors are themselves finite.

- Because of the "mathematical flavour" of process algebras, parallel composition is expressed by binary operators. This approach has several limitations:
  - It is not easy for new users to write an algebraic term denoting a network of concurrent processes (this approach has to be compared with graphical formalisms such as Sdl, in which the user simply has to draw the desired network).
  - There is no canonical form: for a given network, there are usually several different algebraic terms [Kar94, Kar97].
  - Finally, there are process networks that can not be expressed as algebraic terms [Bol90].

Following various proposals [Bri88, Gar95b], it was decided to replace the binary parallel operators with an $n$-ary operator, the semantics of which is very intuitive: this operator simply gives the list of concurrent behaviours; for each behaviour, the list of gates on which the behaviour has to synchronize is specified. For instance, the following process network:




can be specified as:

```
par
   G1, G3 -> B1
   ||
   G1, G4 -> B2
   ||
   G1, G2, G3, G4 -> B3
   ||
   G2, G3 -> B4
   ||
   G2, G4 -> B5
end par
```

This operator establishes a direct mapping between process networks and their textual representation, thus paving the way for graphical tools.

- The parallel composition operator was also extended to allow "2 among $n$" synchronization, i.e., when a set of $n$ processes synchronize two by two on the same gate(s) [Gar95b]: this feature (different from the *maximal cooperation* paradigm of Lotos, where all processes synchronized on a given gate have to perform rendezvous simultaneously) is useful for describing networks of distributed objects connected by binary communication links [FNLL96].

For the design of Lotos NT, we are considering other improvements:

- The hiding operator of Lotos is not fully satisfactory: when a gate is hidden, it is renamed into "**i**" (the Lotos notation for $\tau$-actions) and all values passed on this gate disappear. Although renaming into $\tau$-actions is needed when performing bisimulation-based verification, it may be unsuitable in other circumstances, especially when simulating Lotos descriptions: useful information about internal transitions is likely to be unavailable if the simulator implements the semantics of Lotos strictly.

  A more flexible hiding operator would be desirable, which would offer the choice between either renaming the hidden gate into $\tau$ and forgetting about the values exchanged on this gate, or simply assigning the gate a unique, local name (so that external synchronization on this gate becomes impossible).

- As many process algebras, Lotos is based on the so-called *interleaving semantics*: parallel composition is expressed in terms of sequential composition and non-deterministic choice. This approach is generally appropriate for the description of asynchronous systems (in spite of the criticisms formulated by *true concurrency* proponents). However, it may be unsuitable for safety-critical systems, e.g., avionics [GH94], for which non-deterministic behaviours are generally prohibited by certification authorities.

For such applications, a second generation Formal Description Technique should be able to describe networks of cooperative processes that produce a deterministic execution. Synchronous languages [Hal93] are one possible solution. In the asynchronous framework of LOTOS NT, we are considering two alternative approaches for a deterministic parallel operator:

- *coroutines*: we proposed an operator suitable for describing coroutines, i.e., a set of cooperating processes executing on a single processor [SG97]. For instance:

  ```
  exec
     G1 -> B1
     ||
     G2 -> B2
     ||
     G3 -> B3
  end exec
  ```

  denotes a set of three coroutines `B1`, `B2`, and `B3`, of which only one is active at a time. The active coroutine `Bi` (initially `B1`) executes as long as it does not suspend itself by performing an action `Gj` which transfers the control to another coroutine `Bj`; when suspended, a coroutine `Bi` keeps its current state until it becomes re-activated by another coroutine executing the action `Gi`. The coroutine operator extends to $n$ behaviours the *suspend-resume* operator [HF95, Her98] introduced in E-LOTOS to model interrupts.

- *priorities*: in the real-time world, priorities are the standard way to specify a deterministic execution for a set of concurrent processes (priorities are enforced by the scheduling algorithm and most real-time operating systems support at least 256 different priority levels). Modelling priorities is desirable, as it would allow process algebras to be used in the area of safety-critical, real-time systems. Moreover, recent work [BCL97] demonstrates that priorities can be an effective way of cutting down state explosion in model-checking verification.

  At present, E-LOTOS has no concept of priorities, although the introduction of priorities was a goal of the Committee initially. As regards LOTOS NT, based on [BCL97, CLN97], we propose to attach priorities to internal actions (using the "**hide**" operator to specify priorities) in such a way that prioritized actions can preempt other actions.

There are two other important issues for which we seek a better solution than E-LOTOS:

- *exceptions*: as stated in Section 1, the evaluation of an expression may raise an exception instead of returning a result. The behaviour part should therefore be prepared to deal with exceptions arising from the data part.

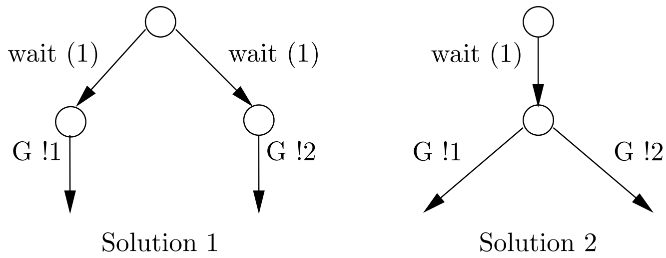
  For this purpose, E-LOTOS introduces a new concept of *signal*, which denotes an urgent event on which the environment can not synchronize. In this approach, exceptions arising from the data part are modelled as signals. Unfortunately, this approach has several drawbacks: because the new concept of signal has to coexist with the old concept of action, there is a sudden inflation in the number of rules in the syntax, static and dynamic semantics. Moreover, it creates methodological problems by forcing specifiers to classify events into signals and actions, a non-obvious decision which has to be made at the earliest steps of a formal design.

  An alternative approach would be to use actions for modelling exceptions arising from the data part, thus keeping actions as the unique concept.

- *time determinism*: initially, the time semantics was based on ET-LOTOS [Leo97, LL97, Her98], itself designed as an upward compatible extension of LOTOS with time. However, the deep changes brought to the behaviour part of LOTOS (such as the introduction of assignment, symmetric sequential composition, and signals) have led to new issues, among which the *time determinism* problem plays a central role. This problem can be summarized as follows. Given the behaviour:

  ```
  (wait (1) ; G !1) [] (wait (1); G !2)
  ```

  should the corresponding Timed Labelled Transition System be non-deterministic with respect to the passing of time (solution 1) or deterministic (solution 2)?

  For E-LOTOS, the Committee adopted solution 2. But the problem is so complex that E-LOTOS requires not less than three different mechanisms to ensure time-determinism:

  1. The structured operational semantics rules for choice are conceived so as to perform some determinization with respect to time transitions. For instance, both behaviours:

```
(wait (1) ; G !1) [] (wait (1); G !2)
```

and:

```
wait (1) ; (G !1 [] G !2)
```

will generate the same Timed Labelled Transition System (solution 2 above). This is also the case for parallel composition.

2. However, time determinization is not always tractable in presence of constructs that take no time to execute (e.g., assignments). For the following behaviour:

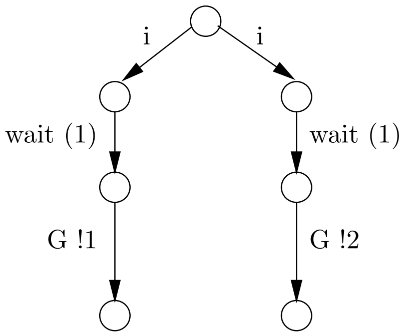
```
(V := 1 [] V := 2) ; wait (1) ; G !V
```

a strict application of E-LOTOS structural semantics rules will generate a non-deterministic Timed Labelled Transition System (solution 1 above). To circumvent the problem, the static semantics of E-LOTOS was modified to forbid any behaviour likely to create time non-determinism. In particular, any operand of the choice operator is required to perform at least an action (visible or hidden) or a signal. Such restrictions are contained in a predicate called "**guarded**", the definition of which obscures the static semantics.

3. Though, these restrictions are still unsufficient. Even if properly "**guarded**", certain behaviours may create time non-determinism, e.g.:

```
trap
   X -> wait (1); G !1
   Y -> wait (1); G !2
in
   signal X [] signal Y
end trap
```

The dynamic semantics of E-LOTOS was modified once more: in certain places, time non-determinism is avoided not by determinization, but by introducing $\tau$-actions! For instance, the Timed Labelled Transition System corresponding to the previous example is the following (with the paradoxical effect that signals `X` and `Y` generate internal actions, although signals and actions are claimed to be different objects everywhere else):



The E-Lotos attempt at time determinism is probably too complex; it will confuse most users and may cause non-compositionality issues. For Lotos NT, we are considering an alternative approach with simpler semantics rules that do not attempt to generate time deterministic models nor to forbid those behaviours that might create time non-determinism.

Instead, the (potentially time non-deterministic) models generated according to these rules are considered modulo a relation that behaves like strong bisimulation with respect to visible and hidden actions (thus preserving the branching structure), and like trace equivalence with respect to time transitions (thus performing time determinization). For instance, both graphs entitled Solution 1 and 2 above are considered to be equivalent modulo this relation.

# 3 Modules

Modularity is a desirable language feature, which allows to decompose a large, monolithic description into smaller parts of manageable sizes. To handle large descriptions, Lotos offers some basic structuring capabilities:

- Each process definition can contain nested definitions of types and processes;
- Each type definition can contain collections of sorts, functions and equations, and thus be seen as a "module". Existing types can be combined together by means of import, renaming, parameterization and actualization, in order to build more complex types.

These design choices have often been criticized, for several reasons:

1. they establish a dissymmetry between the data and behaviour parts;
2. Lotos types lack *abstraction*, as every sort or function declared in a type is automatically exported (a sort or a function can not be declared "local" to a type);
3. Lotos types do not enforce *persistency*, as one can modify the initial semantics of existing types by adding new equations (for instance, a user-defined type can contain an equation "*true* = *false*" that corrupts the semantics of the standard boolean type).

These problems have been known for long and various solutions have been proposed, borrowing ideas from programming languages supporting modularity, such as Modula-2 [Wir83], Ada [WWF87], Sml [MTH90]. Our proposal [SG96] was retained as the basis of E-Lotos [Que98]. It builds upon both Sml and [BL95], which itself improves over a previous proposal [Bri88]. The main ideas are presented below.

E-Lotos modules constitute an additional layer, defined on top of the data and behaviour parts. Contrary to Lotos, they deal with data and behaviours symmetrically. As in Ada, Modula-2, Sml, etc., a clear distinction is made between *interfaces* (which contain declarations) and *modules* (which provide implementations): this allows to define libraries of reusable components, especially suitable for team work.

An E-Lotos description is a "flat" collection of modules and interfaces. Contrary to Sml, interfaces and modules can not be nested: it is not allowed to declare an interface or a module within another interface or module. Yet, interfaces and modules can be organized using import relations (which form directed acyclic graphs).

An interface may contain declarations of types, constructors, functions, and processes.

Type declarations in an interface can be either *complete* (all the constructors of the type are specified), *opaque* (the constructors of the type are not specified, and thus will not be available for pattern-matching, projections, updaters, etc.), or *partial* (a list of constructors is specified and followed by the ellipsis symbol "...", meaning that any implementation for this type must have at least the mentioned constructors; similarly, the argument list of constructors can be incompletely specified).

A type can be qualified with the "**finite**" keyword, meaning that its set of values is finite, so that the type can be used in contexts requiring finite types, e.g., the "**par**" operator that enumerates over a domain. The example below illustrates the various possibilities of type declarations in interfaces:

```
-- example of complete type
type VECTOR is VEC (X, Y, Z: REAL) end type

-- example of opaque (finite) type
finite type COMMAND end type

-- example of partial type
type PACKET is
   DATA (N:INTEGER, ...),
   ACKNOWLEDGE (B:BOOL),
   ...
end type
```

Functions and processes are declared in interfaces by specifying their complete profiles (ellipsis is not allowed). By default, for each type declared in an interface, equality and non-equality functions are automatically made available. Additionally, the "**with**" keyword can be used for a concise specification of the functions available with a given type. For instance:

```
function SIN (X:REAL):REAL

function HEAD (L:LIST):ITEM raises EMPTY_LIST

process BUFFER [INPUT, OUTPUT:PACKET]

type ORDERED_ITEM is ... with <, <=, >, >= end type
```

Following the ideas proposed in Anna [LvKBO87], Larch [GHG+93], Eml [KST97], the interfaces in Lotos NT can be enriched with *properties*, i.e., annotations expressing the expected semantics of functions and processes. At present, we consider three different kinds of properties [GM96]:

- *algebraic properties:* introduced by the "**eqns**" keyword, they allow an axiomatic characterization of data, exactly as ActOne equations, for which they provide a replacement (thus providing some form of backward compatibility with Lotos). Naturally, equations have to be extended in order to characterize situations in which exceptions are raised.
- *behaviour properties:* introduced by the "**rels**" keyword, they use behavioural equivalences and preorders (e.g., bisimulations) to characterize the behaviour of processes. The list of supported relations can be extended as needed, in particular, to include timed bisimulations.
- *temporal properties:* introduced by the "**forms**" keyword, they use temporal logic (or $\mu$-calculus formulas) to characterize the behaviour of processes. As regards the choice of an appropriate temporal logic, [GM96] proposes Actl [NV90], although other choices are possible.

```
function HEAD (L:LIST):ITEM raises EMPTY_LIST

process BUFFER [INPUT, OUTPUT:PACKET]

eqns
   HEAD (NIL) raises EMPTY_LIST
   forall X:ITEM, L:LIST, HEAD (CONS (X, L)) = X;

rels
   hide INPUT, OUTPUT:PACKET in BUFFER [INPUT, OUTPUT] end hide = stop
   mod OBSERVATION_EQUIVALENCE

forms
   BUFFER [INPUT, OUTPUT] |= <true> true        -- absence of deadlock
```

Compared to the constructive definitions of functions and processes, properties are purely declarative. They are useful for requirement capture, design with iterative refinement, and documentation of formal descriptions.

Of course, there is no decision procedure for verifying properties (this problem is undecidable in general). Therefore, properties are only checked with respect to syntax and static semantics (i.e., identifier bindings, typing, etc.). Yet, one can imagine computer tools that would extract properties from Lotos NT descriptions and attempt to check their correctness in defined particular cases.

Interfaces can be combined together by means of import and renaming. An interface must be *self-contained*, i.e., every identifier used in an interface should be defined either in the interface itself, or in an imported interface, or in some predefined library.

A module may contain (constructive) definitions of types (together with their constructors), functions, and processes. Like interfaces, modules can also contain declarative properties.

An relation is defined to express under which conditions a module implements a given interface. For instance: each constructor declared in the interface must be implemented by a corresponding constructor defined in the module, whereas each function defined in the interface can be implemented either by a constructor or a function; each type declared as finite in an interface must be implemented by a non-recursive type, etc. A given interface can be implemented by several modules and, reciprocally, a given module can be seen through several interfaces (thus leading to the concept of *views*).

The list of objects exported by a module can be restricted by specifying an interface for this module: only the objects declared in the interface will be visible outside the module (*selective export*). This interface is optional: by default, every object defined in the module is exported.

Modules can be combined together by means of import, *selective import* (i.e., the contents of the imported module are filtered by an interface), and renaming.

Modules can also be parameterized by interfaces, resulting in *generic modules* to be instantiated by other modules. This genericity mechanism improves over Lotos: for instance, processes can be parameterized by formal types, or even formal processes. This was already possible in [BL95], but Lotos NT brings even more flexibility by introducing partial types (i.e., a notion of subtyping in instantiation). As in Ada and Lotos, the actualization is handled statically by substituting actual parameters to formal ones in the body of the generic modules to be instantiated. The dynamic semantics is defined only for fully instantiated modules.

# Conclusion

Roughly speaking, the first generation of description languages for concurrent systems can be divided in two classes: mathematical formalisms (among which process algebras play a prominent role), and design languages intended for industry engineers, which often lack semantic foundations. We believe that it is high time for a second generation of Formal Description Techniques, semantically based upon process algebras, but clearly oriented towards computer scientists and practical applications.

In this paper, we presented a few ideas to improve the Iso standard Lotos [ISO88b] by combining features from process algebraic and classical programming languages. These ideas have been submitted to Iso and some of them have been accepted for integration in the forthcoming E-Lotos standard [Que98]. In parallel, we are working on the definition of a simpler language named Lotos NT, which is reaching completion at Inria Rhône-Alpes (an early version of Lotos NT was used for an industrial case-study [SM97]), and a prototype compiler/model-checker is currently under development. We expect Lotos NT to be:

- *Expressive enough:* Lotos should be a strict subset of Lotos NT, the latter bringing new features such as partial functions (exceptions), generalized parallel operators, time, coroutines, priorities, etc.
- *Easy to implement:* process algebras and algebraic data types are notoriously difficult to implement, although efficient approaches are possible, e.g. for Lotos [Gar89, GS90]. In comparison, Lotos NT should be much easier to implement, because an important part of it (at least the data part) can be directly mapped to algorithmic languages (e.g., Ada, C, C++, Java, etc.) for which efficient compilers already exist.
- *Easy to learn:* we tried to keep the syntax and semantics of Lotos NT as simple as possible. Each construct can be justified by clear expressiveness or user-convenience reasons. Compared to Lotos, significant progresses towards simplicity have been achieved:
  - As regards syntax, cryptic algebraic symbols have been avoided as much as possible; behaviour operators have been syntactically "bracketed" with "**end**" keywords to avoid ambiguities, as suggested in [Bri88]. Readability was a major design concern, so as to refute the common misconception "formal" means "unreadable".
  - Contrary to description languages that embody two different languages for the description of data and behaviours (e.g., Lotos, $\mu$Crl, Sdl), a maximal unification between the data and behaviour part has been achieved: many language constructs are common and the data part can even be considered as a subset of the behaviour part.
  - As regards the data part, abstract data types, which were found inappropriate by most users, have been replaced with constructive types borrowed from functional languages and functions defined in a standard, imperative style.

– The behaviour part of Lotos was also simplified by removing its most confusing features for beginners: the functional style for dealing with variables was replaced with an imperative one, thus allowing state variables to be modelled directly; the two different sequential composition operators (action prefix and enabling) have been merged into a single one; the binary parallel operator was extended to a simpler, more general one; etc.

From our teaching experience, we notice that "lazy" students, who do not take time to learn Lotos properly, spontaneously tend to write Lotos NT descriptions when asked to produce Lotos ones! We consider this situation as a strong indication that our approach is pertinent. In an ideal Formal Description Technique, only difficult concepts (such as non-determinism, synchronization, communication, real-time, etc.) would require explanations, the remaining constructs (conditionals, assignments, loops, etc.) being standard.

# Acknowledgments

The authors are grateful to their colleagues in the E-Lotos standardization Committee, to the European Commission for supporting the Cost 247 project, which played an significant role in the elaboration of E-Lotos, to their colleagues in the Cost 247 project, to Didier Bert, Rance Cleaveland, Radu Mateescu and Charles Pecheur for their judicious comments and suggestions.

# References

[Ard97] Mark Ardis. Formal Methods for Telecommunication System Requirements: a Survey of Standardized Languages. *Annals of Software Engineering*, 3:157–187, 1997.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1988.

[BCL97] G. Bhat, R. Cleaveland, and G. Lüttgen. Dynamic Priorities for Modeling Real-Time. In *Proceedings of the 17th Joint Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE X/PSTV XVII '97), Osaka, Japan*, pages 321–336. Chapman and Hall, November 1997.

[BE86] D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proceedings of ESOP'86*, volume 213 of *Lecture Notes in Computer Science*, pages 119–132, Berlin, 1986. Springer Verlag.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BK84] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60:109–137, 1984.

[BL94] B. Berthomieu and T. Le Sergent. Programming with Behaviors in an ML Framework - The Syntax and Semantics of LCS. In *Proceedings of the 5th European Symposium on Programming, ESOP'94, Edinburgh, U.K.*, 1994.

[BL95] E. Brinksma and G. Leih. *Enhancements of LOTOS*. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 453–466. Kluwer Academic Publishers, 1995.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[Bol90] T. Bolognesi. A Graphical Composition Theorem for Networks of Lotos Processes. In IEEE Computer Society, editor, *Proc. 10th International Conference on Distributed Computing Systems, Washington, USA*, pages 88–95. IEEE, May 1990.

[Bri88] Ed Brinksma. *On the Design of Extended LOTOS, a Specification Language for Open Distributed Systems*. PhD thesis, University of Twente, November 1988.

[Cam89] J. Camillieri. An Operational Semantics for OCCAM. *International Journal of Parallel Programming*, 18(5):149–167, October 1989.

[Cd95] J.P. Courtiat and R.C. de Oliveira. A Reachability Analysis of RT-LOTOS Specifications. Technical Report 95159, LAAS, May 1995.

[CLN97] R. Cleaveland, G. Lüttgen, and V. Natarajan. A Process Algebra with Distributed Priorities. *Theoretical Computer Science*, 1997. To Appear.

[dS85] Robert de Simone. Higher-level synchronising devices in Meije–SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[Exn94] Jürgen Exner. The OPAL Tutorial. Technical Report 94-9, Technische Universität Berlin, Berlin, May 1994.

[FGGP93] A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. Algebraic Programming in OPAL. *Bulletin EATCS*, (50):171–181, June 1993.

[FNLL96] A. Février, E. Najm, G. Leduc, and L. Léonard. Compositional Specification of ODP Binding Objects. In *Proceeding of the 6th IFIP/ICCC Conference on Information Network and Data Communication, INDC'96, Trondheim, Norway*, June 1996.

[For97] Formal Systems (Europe) Ltd. *Failure-Divergence Raffinement – FDR2 User Manual*, October 1997.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[Gar95a] Hubert Garavel. On the Introduction of Gate Typing in E-LOTOS. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP International Workshop on Protocol Specification, Testing and Verification (Warsaw, Poland)*. IFIP, Chapman & Hall, June 1995.

[Gar95b] Hubert Garavel. A Wish List for the Behaviour Part of E-LOTOS. Rapport SPECTRE 95-21, VERIMAG, Grenoble, December 1995. Input document [LG5] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

[GH94] Hubert Garavel and René-Pierre Hautbois. *Experimenting LOTOS in Aerospace Industry.* In Teodor Rus and Charles Rattray, editors, *Theories and Experiences for Real-Time System Development*, volume 2 of *Amast Series in Computing*, chapter 11. World Scientific, 1994.

[GHG+93] J. Guttag, J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Text and Monographs in Computer Science. Springer Verlag, 1993.

[GM96] Hubert Garavel and Radu Mateescu. French-Romanian Proposal for Capture of Requirements and Expression of Properties in E-LOTOS Modules. Rapport SPECTRE 96-04, VERIMAG, Grenoble, May 1996. Input document [KC4] 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.

[GP95] J. F. Groote and A. Ponse. Syntax and semantics of $\mu$-CRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes, Workshops in Computing*, Berlin, 1995. Springer Verlag.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[GS96] Hubert Garavel and Mihaela Sighireanu. On the Introduction of Exceptions in LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 469–484. IFIP, Chapman & Hall, October 1996.

[GV92] J. F. Groote and F. W. Vaandrager. Structured Operational Semantics and Bisimulation as a Congruence. *Information and Computation*, 100(2):202–260, October 1992.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub., 1993.

[Her98] Christian Hernalsteen. *Specification, Validation and Verification of Real-Time Systems using ET-LOTOS*. Thèse de Doctorat, Université Libre de Bruxelles, June 1998.

[HF95] C. Hernalsteen and A. Février. A suspend/resume operator for ET-LOTOS. Input document [LG9] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège, Belgium, December, 19–21, 1995, December 1995.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall, 1991.

[ISO88a] ISO/IEC. ESTELLE — A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[ISO88b] 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, Genève, September 1988.

[ISO96] ISO/IEC. Interface Definition Language. Technical Report, International Organization for Standardization — Open Distributed Processing, 1996.

[IT92] ITU-T. Specification and Description Language (SDL). ITU-T Recommendation Z.100, International Telecommunication Union, Genève, 1992.

[JGL+95] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21 N10108 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[Kar94] Pim Kars. Representation of Process-Gate Nets in LOTOS and Verification of LOTOS Laws: the Boolean Algebra Approach. In Dieter Hogrefe and Stefan Leue, editors, *Proceedings of the 7th International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols FORTE'94 (Bern, Switzerland)*, October 1994.

[Kar97] P. Kars. *Process-Algebraic Transformations in Context*. PhD thesis, University of Twente, June 1997.

[KST97] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, '997.

[Leo97] Luc Leonard. *An Extended LOTOS for the Design of Time-Sensitive Systems*. PhD thesis, University of Liège, September 1997. Collection of Publications of the Faculty of Applied Sciences, Nr 177.

[LL91] E. Lallemand and G. Leduc. A LOTOS Data Facility Compiler (DAFY). In Kenneth R. Parker and Gordon A. Rose, editors, *Proceeding of 4th International Conference on Formal Description Techniques FORTE'91*. North-Holland, 1991.

[LL97] Luc Leonard and Guy Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29:271–292, September 1997.

[LvKBO87] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe. *Anna, a Language for Annotating Ada Programs: Reference Manual*, volume 260 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Mun91] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[NV90] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems*. In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[Pec93] Charles Pecheur. VLib: Infinite Virtual Libraries for LOTOS. In A. Danthine, G. Leduc, and P. Wolper, editors, *Proceedings of the 13th IFIP International Workshop on Protocol Specification, Testing and Verification (Liège, Belgium)*, pages 1–16. IFIP, North-Holland, May 1993.

[Pec96] Charles Pecheur. *Improving the Specification of Data Types in* LOTOS. Doctorate thesis, University of Liège, November 1996. Collection of Publications of the Faculty of Applied Sciences, Nr 171.

[Pep94] P. Pepper. The Programming Language Opal — Implementation Language. Technical Report, Fachbereich Informatik, Technische Universität Berlin, February 1994.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. DAIMI FN-19 FN-19, Computer Science Department, Aarhus University, 1981.

[QA92] J. Quemada and A. Azcorra. Structuring Protocols with Exception in a LOTOS Extension. In *Proceedings of the 12th IFIP International Workshop on Protocol Specification, Testing and Verification (Orlando, Florida, USA)*. IFIP, North-Holland, June 1992.

[Que98] Juan Quemada, editor. Committee Draft on Enhancements to LOTOS (E-LOTOS). ISO/IEC FCD 15437, April 1998.

[RdMS95] R. Roth, J. de Meer, and S. Storp. *Data specifications in modular LOTOS*. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 467–479. Kluwer Academic Publishers, 1995.

[SBL+91] R. Strom, D. Bacon, A. Lowry, A. Goldberg, D. Yellin, and S. Yemini. *Hermes: A Language for Distributed Computing*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, USA, 1991. ISBN 0-13-389537-8.

[Sco86] G. Scollo. Some Facilities for Concise Data Types in LOTOS. Technical Report, EspritP410 SEDOS/C1/WP13/T, May 1986.

[SG96] Mihaela Sighireanu and Hubert Garavel. On the Definition of Modular E-LOTOS. VASY Report, INRIA, December 1996. Input document [GR2] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Grenoble, France, December, 9–11, 1996.

[SG97] Mihaela Sighireanu and Hubert Garavel. A Proposal for Coroutines in E-LOTOS. VASY Report, INRIA, July 1997. Input document [HEL2] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Helsinki, Finland, July, 14–18, 1997.

[SM97] Mihaela Sighireanu and Radu Mateescu. Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, June 1997. Full version available as INRIA Research Report RR-3172.

[Sto91] Silke Storp. Integration of Standard Data Types into LOTOS. Master's thesis, Offene Kommunikations Systeme (OKS), Berlin, June 1991.

[Sun96] Sun Microsystems Inc. *Java Language Specification*. Addison Wesley, first edition, 1996.

[Wir83] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, 1983.

[WWF87] D. Watt, B. Wichmann, and W. Findlay. *ADA Language and Methodology*. Prentice-Hall, 1987.