B4-43

Reference : Lo/WP1/T1.2/IEI/N0013/V3

LOTOSPHERE

ESPRIT Project 2304[1]

Title	:	**The Expressive Power of LOTOS Behaviour Expressions**
Status	:	Public
Type	:	Paper, Final
Editor	:	A. Fantechi, S. Gnesi, G. Mazzarini
Date	:	3/3/1992
Distribution	:	PM, WP1M, T1.2
Owner	:	
Note	:	Revised version of the paper presented at FORTE'90 Conference, Madrid, 5-8 November 1990

[1] The LOTOSPHERE consortium consists of the following companies/institutes/universities:

Alcatel Standard Electrica, Ascom Tech, British Telecommunications, C.N.R.-CNUCE, PTT-DNL, Consorzio Pisa Ricerche, Gesellschaft für Mathematik und Datenverarbeitung, C.N.R.- Istituto di Elaborazione della Informazione, Institut National de Recherche en Informatique et en Automatique, Laboratoire d'Automatique et d'Analyse des Systemes du CNRS, Océ Nederland, SYSECA Logiciel, TECSIEL, Technische Universität Berlin, Universidad Politécnica de Madrid, University of Stirling, University of Twente.

The Expressive Power of LOTOS Behaviour Expressions

Alessandro Fantechi◇, Stefania Gnesi◇, Gianluca Mazzarini○

◇*IEI-CNR, Via S. Maria 46, I 56100 Pisa, ITALY*

○*Consorzio Pisa Ricerche, Via Risorgimento 9, I 56100 Pisa, ITALY*

Abstract. It is a known result that, when in a process algebra like LOTOS we have both recursion and parallel composition operators, the finite-state behaviour of processes is in general lost. Actually, this happens also when other LOTOS operators, such as enabling or disabling, are used in a recursive context.

The purpose of our paper is to study the expressive power of Basic LOTOS in terms of the relations between the syntactical structure and the finite-state behaviour of processes. We then define some sufficient conditions on the syntactical shape of the specifications in order to preserve the finite-state nature, extending those already presented in the literature. Moreover, we define the expressive power of other LOTOS subsets, ranging from context-free processes to Turing-equivalent processes.

This work was supported in part by the CEC under ESPRIT project 2304 LOTOSPHERE.

1. Introduction

In the analysis of LOTOS specifications several methodologies and most of automatic tool support can be used only if the specifications considered describe finite-state machines. For example, an equivalence verifier is able to manipulate only specifications in this class; an attempt to verify the equivalence of two non finite-state processes may not terminate with an answer. On the other hand, even some simple protocol specifications can only be described (at least at a certain level of abstraction) by a non-finite state machine, so recently some attempts have been done to exploit more powerful verification techniques (see [FGL89, BaB90, DIN91]).

To know what can be *effectively* verified (e.g. equivalences, properties, and so on) on a generic LOTOS specification it is useful to understand what is the "expressive power" of different LOTOS operators: a subset of LOTOS has a greater *expressive power w.r.t. another if it is able to express a wider class of processes*.

If we consider a process as the set of its computations (maximal traces), an interesting classification of the different classes of processes can be done on the basis of the decidability of some classical problems of formal language theory (e.g. the equivalence of two processes, intended as having exactly the same set of computations, called *maximal traces*). In this respect, the maximal expressive power that a subset of LOTOS can exhibit is that of Turing Machines. In LOTOS the presence of values, conditional expressions and recursion is enough to reach the expressive power of Turing Machines. It turns out, however, that even without resorting to the use of values (that is, even using the so called "Basic LOTOS"), it is possible to express Turing Machines; actually, different expressivity results can be given for different LOTOS operators. This paper presents an analysis of the expressive power of Basic LOTOS process operators: the possibility of using Basic LOTOS expressions as describing traces of actions can be directly used to inherit known results from the theory of formal languages.

These results cannot be immediately extended to full LOTOS, but we consider that they are useful in several respects:

1) Several analysis prototype tools and methodologies work only for Basic LOTOS, so that a reduction to Basic LOTOS is needed in order to analyse a LOTOS specification.

2) The Basic LOTOS operators identify the general structure of a specification, hence negative limits on the decidability of problems extend to full LOTOS specifications with the same structure.

3) Some results obtained for Basic LOTOS can as well hold for full LOTOS specifications: whenever we will show, for example, that a particular combination of Basic LOTOS regular processes is also regular, this will hold also if the component regular processes are full LOTOS

ones.

Consider, for example, a specification written in a *constraint oriented* style [VSS88]: the specification is achieved by the "top-level" parallel composition of a number of processes acting as constraints. Since it is shown that "static" parallel composition of regular processes is still regular, we can deduce that if all the constraints are regular, so is the complete specification. On the other hand, a *resource oriented* specification in which an unbounded number of processes are dynamically allocated is not, in general, regular.

Our analysis shows first how LOTOS can express Turing Machines, giving two different minimal subsets of Basic LOTOS which are both able to reach the expressive power of Turing machines (Section 3).

We then recall the reasons why it is impossible to find necessary conditions on the syntax for restricting to finite-state processes (Section 4).

Starting from the simplest subset of LOTOS terms which generate finite-state process, we proceed in our analysis by adding operators (Section 5), finding different subsets of expressive power ranging from finite-state processes, to context free processes, to Turing equivalent processes. For each of this subset it is briefly discussed which verification problems are decidable and which are not. The results in this sense are summarized in Appendix A.

Appendix B collects conditions on the syntax of finite state processes, which are individually presented throughout the analysis of the different operators in section 5. These conditions can be checked by automated analysis tools, as equivalence verifiers, to avoid infinite loops. Some of these conditions can be found (derived with different approaches) also in [Ail86], [GaN89] for LOTOS and in [MaV90], [Tau89] for generic process algebras; actually, we obtain also some finer results on the parallel composition operator. This gives the opportunity to enrich the set of processes accepted by tools, as AUTO [MaV89], which already check finite-state conditions on input processes.

2. Basic LOTOS

The considered language is Basic LOTOS, as defined in [BoB87], which only describes process synchronization, while full LOTOS also describes interprocess value communication. Basic LOTOS employs a finite alphabet of observable actions. Actions occur at *gates*; since no value communication is present, we can identify the observable actions with the gate at which they occur. The alphabet of actions include also an unobservable (or internal) action i and an action d used to give semantics to the enabling operator (see Appendix C). In Table 1 we present the syntax of the operators of Basic LOTOS; the table presents also the subset formed by **stop**, action prefix ';' choice '[]' and process instantiation (both recursive and renaming): we call it DELOTOS (DEcidable LOTOS, since, as we will see, all problems are decidable on this subset), which is able to specify every sequential non-communicating process. The operational semantics of Basic LOTOS and the definitions of some equivalence relations on it are presented in Appendix C.

It is necessary to point out that, both in DELOTOS and in the other subsets, we consider *guarded* recursion, i.e. all the recursive calls must be preceded by an action of the alphabet.

In formalizing recursion, Basic LOTOS makes use of the mechanism of process instantiation. However, through process instantiation it is possible not only to model recursion, but also to perform a sort of *renaming* of the actions, by passing actual parameters which are different from the formal ones. Renaming can be simulated as a syntactical substitution of gates when no operator acting on gates, such as hiding or parallel composition, is involved; otherwise, renaming becomes a semantic substitution (changing gates in the computations) analogous to CCS relabelling [Mil80]. For this reason, we will call Basic LOTOS* the subset of Basic LOTOS where process instantiation is admitted only without renaming; we will use also the notation DELOTOS* with the same meaning.

DELOTOS *BASIC LOTOS*

operator		syntax
Inaction		stop
Unobservable action		i;B
Observable action		g;B
Choice		B1[]B2
Successful termination		exit
Sequential composition	(enabling)	B1>>B2
Disabling		B1[>B2
Synchronization		B1||B2
Interleaving		B1|||B2
Parallel composition	(general form)	B1|S|B2
Hiding		hide S in B
Process instantiation		pid[g1,...gn]

Table 1. Basic LOTOS operators

3. Basic LOTOS and Turing Machines

Since LOTOS is a formalism which completely covers the description of concurrent systems, we may wonder whether Basic LOTOS has the full power of computation of Turing Machines: if so, we should manage to describe Turing Machines by means of a Basic LOTOS specification.

Indeed, this result can be achieved formalizing a two-counter, which is known to be able to simulate Turing Machines [JLL77]. A *two-counter* is a 6-tuple:
$C = <Q, q_0, q_F, I, C_1, C_2>$
where Q is a finite set of states, $q_0 \in Q$ is the initial state, $q_F \in Q$ is the halting state, I is a finite set of instructions, and C_1, C_2 are counters able to keep a non-negative integer.

At the beginning, the counters are set to zero; the possible instructions are (the states q, q*, r, s belong to Q):

a) (q, D_i, q^*): being in the state q, decrease the counter C_i and go to the state q*.
b) (q, I_i, q^*): being in the state q, increase the counter C_i and go to the state q*.
c) (q, T_i, r, s): being in the state q, check whether C_i is set to zero: if so, go to r, otherwise go to s.

The control program for this two-counter can be represented as a finite state automaton (FSA) and so as a Basic LOTOS process making use only of choice, action prefix operators and recursive process instantiation.

The following Basic LOTOS process defines a zero-counter, which is the formalization of one of the two single counters constituting the two-counter; it is the Basic LOTOS transposition of the corresponding CCS process given in [GoM84]:

```
process zero-counter :=
z [inc ,dec, tst]
where
    process c [a, inc, dec] := (inc; hide b in
    (c [b, inc, dec] | b | (b; c [a, inc, dec])) []
    (dec; a; stop))
    endproc
    process z [inc, dec, tst] := (inc; hide b in
    (c [b, inc, dec] | b | (b; z [inc, dec, tst])) []
    (tst; z [inc, dec, tst]))
    endproc
endproc
```

Now, to specify the *two-counter* it is simply required to write two distinct zero-counters, C_1 and C_2, each one with indexed instructions, to put them in interleaving and to compose them in parallel on the appropriate gates with the control program, as shown in Fig. 1.

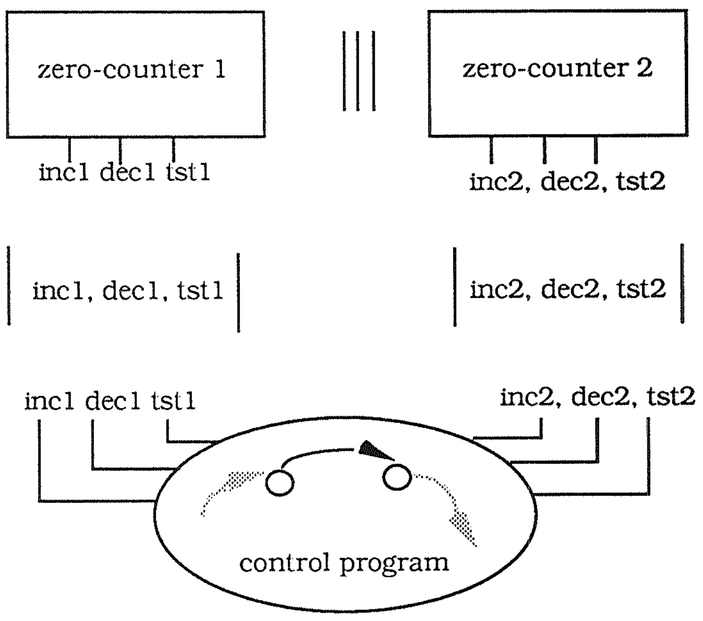


Figure 1

We can observe that in describing the two-counter we have used all Basic LOTOS operators except *enabling* >> and *disabling* [>; thus we can already state the following result:

Proposition 1: *Basic LOTOS without enabling and disabling can express Turing Machines.*

Alternatively, we can define the two-counter in Basic LOTOS*, without using hiding: in fact, adapting to Basic LOTOS* a result from [BeK84], it is possible to write a zero-counter using process instantiation (without renaming), enabling and choice:

```
process zero-counter-bis :=
c[tst,ins,dec]
where
   process c [tst,inc,dec] := (tst; exit [] inc; h [dec,inc]) >> c [tst,inc,dec] endproc
   process h [dec,inc] := dec; exit [] (inc; h [dec,inc]) >> h [dec,inc] endproc
endproc
```

As in the previous example, we can interleave two instances of this counter and compose them appropriately with a control section, again obtaining a representation of Turing Machines.
In this way, we have shown the following statement:

Proposition 2: *Basic LOTOS* without hiding and disabling can express Turing Machines.*

4. The Finite State Constraints on Basic LOTOS

Several existing verification tools (AUTO [MaV89], Squiggles [BoC89], CÆSAR [GaS90]...) work correctly on Basic LOTOS processes which can be represented by a finite state automaton (FSA), i.e. processes generating *regular* maximal traces, on which most of the problems are decidable.

It would be desirable to have both necessary and sufficient conditions on the syntactical shape of Basic LOTOS behaviour expressions in order to distinguish the regular ones, so that one can understand which processes can be given safely as input to the verification tools, avoiding in this way the risk of wrong and/or incomplete answers and of infinite loops. In order to establish such conditions, we match every Basic LOTOS process with a "characteristic language", that is represented by the set of maximal traces on the set of the actions. In this way, it is possible to make use of the known results of the language theory in the examination of the expressive power of Basic LOTOS processes[1].

Unfortunately, as it has already been noted in [MaV89], it is not possible to establish general necessary conditions which permit to state whether a Basic LOTOS process gives rise to regular maximal traces:

Proposition 3: *Given a Basic LOTOS process P, it is undecidable to say whether it generates regular maximal traces.*

Proof: We have shown above that the whole Basic LOTOS has the full expressive power of Turing Machines. This means, reasoning in terms of maximal traces, that the class of Basic LOTOS processes coincides with the class of phrase structure grammars: but, for this class, the problem of establishing whether the language generated by a particular grammar is regular is known to be undecidable. Thus, it is equally unsolvable the problem of deciding whether, given

[1] Note that we will consider only (maximal) traces as models of LOTOS computations; this implies that we will refer to the maximal trace equivalence. It is not obvious that all the results obtained for maximal trace equivalence can be extended to bisimulation equivalence, because it takes into account the branching structure of processes.
As a reference for known results of language theory see, for example, [HoU69].

a Basic LOTOS process, the set of its maximal traces represents a regular language. ♦

As a consequence of this result, it is not possible to provide necessary conditions on the syntax of processes for regularity; we can only define some sufficient conditions by means of:
i) searching for Basic LOTOS subsets of operators which describe only regular languages,
ii) detecting the upper bounds in the expressive power of other subsets in order to recognize when regularity is lost and trying, at the same time, to give suitable restrictions on the use of some operators in order to obtain regular behaviours.

The results we obtain about regularity are scattered throughout the analysis of the different operators in the next section and are grouped in a table in Appendix B.

We remark that the conditions we give are on the syntax of the process, and therefore can be checked by a textual analysis of the process; other techniques are available, like the interpretative technique proposed in [DeI91], which are able to decide the finiteness of a process in some cases not captured by the syntactic criteria. In this paper we restrict ourselves to what can be obtained by a textual analysis.

5. The Expressive Power of Basic LOTOS Operators

The first Basic LOTOS subset that can be considered is *finite Basic LOTOS*. In this subset there are all Basic LOTOS operators, apart from process instantiation used as recursion. Only finite processes (thus finite computations) can be described: therefore, being in a finite context, every problem can be decided on this class of processes. Of course, admitting finite processes only is of little interest in practical cases, therefore it will not be further considered.

Let us now analyze the subset DELOTOS; DELOTOS processes map directly in regular expressions, thus we can immediately state the following result[2]:

Proposition 4: DELOTOS has the same expressive power of regular languages.

Hence, any problem which is decidable on regular grammars is still decidable on DELOTOS processes: in particular, it is decidable whether a computation is admissible for a process and whether two given processes are maximal trace equivalent (producing the same set of maximal traces). In this last case, it is sufficient to build the minimal automata corresponding to each process and then verify their equivalence. Also bisimulation equivalence is known to be decidable on the automata corresponding to regular expressions.

[2] The results obtained for the regular languages can be easily extended to the class of ω (or ∞)-regular languages using the patterns described in [Par81]; this extension may not necessarily be obvious for the other classes of languages (see [Niw84]), so we will limit ourselves to the finite case.

It is also easy to see that the expressive power of DELOTOS and of DELOTOS* (DELOTOS with no renaming process instantiation) coincide: renaming process instantiation in this case can be simulated syntactically, since no operator acting on gates is present.

We now proceed to analyze what actually happens when DELOTOS is extended with the other operators.

5.1 *Enabling*

First, let us add the *enabling* operator >> to DELOTOS; obviously, it is necessary to introduce also the **exit** operator when using enabling: we will name the obtained subset DELOTOS + >> (the presence of **exit** is implicit).

When the enabling is used within recursion, DELOTOS + >> is as expressive as context-free languages; more precisely, this result holds if we discard the unobservable actions raising from the **exit**, according to the usual concept of *observational equivalence*:

Proposition 5: *Each process of DELOTOS + >> modulo observational equivalence can be written as a context-free grammar and viceversa.*

Proof: The proof follows in part the lines given for another process language, the Basic Process Algebra in [BBK88], in which systems of recursive equations are used.
Let us see first how to translate a context-free grammar (CFG) in a DELOTOS + >> process (assuming to discard unobservable actions). We recall that each CFG can be reduced to a normal form, the so called *Greibach normal form* (GNF); without losing generality, we can assume that each CFG considered here has no ε-productions, i.e. productions of the form A —> ε, where ε is the empty string, as it can be shown that any CFG with ε-productions is equivalent to a CFG without them; with this assumption, a stronger result holds: each CFG can be put in *reduced* GNF, that is a CFG with productions of the form:

A—>aα with "a" a terminal symbol, α string of non-terminal symbols with $|\text{length of } \alpha| \leq 2$.
Given such a grammar, it is possible to write a corresponding DELOTOS + >> process using these rules:

a) For each nonterminal symbol introduce a process name.
b) If, for this symbol, there are alternative productions, these will be represented with a nondeterministic choice [].
c) Any production like

   A—>aBC (with B, C possibly equal to A)

   will be translated in the body of the process A using the expression a;B>>C.

   Any production of the form

   A—>aB

will obviously be put in the form a; B and productions like

A—>a

will be translated as a; **exit**.

It happens that in each finite maximal trace a 'd' action (coming from the final "**exit**") remains at the end: to avoid this, each process can be put in the form

P >> **stop**

so that the unobservable actions produced by the enabling can be discarded.

As for viceversa, we show first that each process of DELOTOS + >> , not using the **stop** operator, can be reduced to a system of recursive equations [BBK88]. A system of recursive equations is a pair (X_0, E), where X_0 is a recursion variable (the "root" variable) and E is a finite set of recursive equations $\{X_i = s_i(X_0,...X_n) \mid i = 0,...n\}$. The $s_i(X_0,...X_n)$ are process expressions on the algebra $(+, \cdot)$, where '+' stands for the union and '$\cdot$' for the concatenation, possibly containing occurrences of the recursion variables $X_0,...X_n$.

In [BBK88] it is shown that, naming H a system of recursive equations and denoting with H^t the CFG obtained by replacing '+' by '|' and '=' by '—>' (the start symbol being the root variable of H), the solution of H is the context-free language generated by H^t. For this reason, the reduction of a DELOTOS + >> behaviour expression to a system of recursive equations is enough to show our assumption. This reduction is routine: the choice will be represented as '+', the action prefix and the enabling can be represented by a concatenation of the symbols in the recursive system; moreover, each process in the behaviour expression will be transformed in a recursion variable (the "root" variable will be the outermost process). The process "**exit**" does not appear in the system of recursive equations when the reduction has been performed. Also in this case, process instantiation involving renaming can be treated syntactically.

In the solution of the obtained system of recursive equations the 'd' actions, coming from the final "**exit**", do not appear : to avoid this, we can add to the system the equation: $Y=X_0.d$.

When also the **stop** operator is used, it is not possible to build as before the corresponding system of recursive equation; however, we can derive a context-free grammar with the following procedure:

a) starting from a process P a grammar is derived as before, mantaining **stop** as terminal symbol. For example, from **process** P [a, b, c] := a; **stop** [] b; P >> (P [] c;**exit**) **endproc** we derive the grammar:

P—> a stop | b P Q

Q—> P | c

b) from this grammar we derive the corresponding pushdown automa. Obviously it does not recognize the maximal traces generated by P. Let us modify the automa by eliminating the moves involving the terminal symbol stop and substituting them by ε-moves to a new final state. It is easy to see that this automa recognizes exactly the maximal traces of P. Now, the context free grammar for this language can be derived from the automa in a standard way [HoU69]. ♦

We have thus seen that adding enabling to DELOTOS causes an increase of expressive power, from regular to context-free languages: as a consequence, in the subset DELOTOS + >>, even if the problem of establishing whether a trace is produced by a process remains solvable, the problem of the maximal trace equivalence of two processes becomes unsolvable. Anyway, the undecidability result for maximal trace equivalence on languages does not extend in this case to bisimulation equivalence on processes, as a result from Baeten, Bergstra and Klop shows: *Bisimulation equivalence may be decidable on processes expressing context-free languages.* The proof of this statement (see [BBK88]) is based on the regularity properties of the computation trees, starting form the system of recursive equations defining such processes. DELOTOS + >> processes, without the **stop** operator, as we have seen, can be expressed as a system of recursive equations, and therefore share the decidability of bisimulation equivalence. A better algorithm for this problem, based on the construction of semantic tableaus, has later been presented by Hüttel and Stirling [HüS91].

Two possible restrictions can be made on DELOTOS + >> processes to achieve regularity:
a) If enabling is never involved within recursive calls in a process of DELOTOS + >> and the sequentially composed processes are regular then also their sequential composition is regular.
To see this, it is necessary to get the finite state automata corresponding to the arguments of the enabling and then connect with an arc marked with 'i' the final nodes of the right argument automaton (corresponding to the **exit**) to the initial node of the left argument automaton. The obtained finite state automaton corresponds to the sequential composition which is hence regular.
b) If enabling is present within recursion, it is sufficient that at least its left argument is a finite-state process, not containing the recursive call.
In this case it is possible to derive a finite-state automaton (FSA) for the process, i.e. given A*, the FSA for the left argument of enabling, we proceed as in case a) and, at any time we meet a recursive call in the right argument, we simply add an arc back to the initial node (coinciding with the initial node of A*).

5.2 Disabling

Using disabling instead of enabling in extending DELOTOS, we can expect to achieve similar results, because the maximal traces of a process of the form P[>Q are prefixes of the traces obtained by concatenating the maximal traces of P to those of Q, that is, of the maximal traces of P>> Q (discarding the 'i' actions produced by the **exit**).

Actually, it can be proved that *not all* the context-free grammars can be represented by a DELOTOS + [> process.

Proposition 6: *The language DELOTOS + [> is strictly more expressive than regular grammars, but also strictly less expressive than context-free grammars.*

Proof: This proposition is proved in three steps:

a) produce a DELOTOS + [> description of a context-free grammar;
b) exhibit a context-free grammar that cannot be represented as a DELOTOS + [> process;
c) prove that no more than context-free grammars can be expressed, i.e. DELOTOS + [> cannot describe context-sensitive languages.

a) Let us consider the process:

```
process rec-dis [a, b] := a; (rec-dis [a, b] [> (b; stop))
endproc
```

According to the operational semantics of disabling, its set of maximal traces is $CF=\{a^n b^m \mid n\geq 1, 1\leq m\leq n\}$. The grammar generating the language CF is:

$$S \longrightarrow aSb \mid aS \mid ab$$

which is a context-free grammar.

b) Let us consider the context-free language $CF' = \{a^n b^n \mid n\geq 1\}$. No DELOTOS + [> process can admit CF' as the set of its maximal traces. In fact, it cannot be produced using disabling outside a recursive call: as we will see, in this way we still obtain regular processes. Thus, it should be produced by a process syntactically similar to the above one. But, in this case, for each computation produced by disabling we have also the partial traces of the left argument (of disabling) joined with the maximal traces of the right one: for this reason we cannot find a DELOTOS + [> process whose maximal traces are only the ones belonging to CF', which have an exactly equal number of different symbols.

c) Given a process P[>Q, we assume, for induction, that the maximal traces of P and those of Q are two context-free languages. Now, the maximal traces of P[>Q are the prefixes of the maximal traces of P, concatenated with the maximal traces of Q. It is known that the language of prefixes of a context-free language is also context-free; since the class of context-free languages is closed for concatenation, P[>Q is a context-free language. ♦

The example at point a) in the proof also shows that the context-free grammars identified by DELOTOS + [> processes are not "simple".

A CFG is said to be *simple* if there is no pair of different productions $A \longrightarrow a\alpha$, $A \longrightarrow a\beta$: this means that, at each step of derivation, the choice of the production to apply is deterministic.

It has been shown that, for this subclass of context-free grammars, the problem of equivalence is decidable. As DELOTOS + [> describes context-free languages that do not belong to this "simple" subclass, the problem of maximal trace equivalence is not, in general, decidable.

For what concerns restrictions on DELOTOS + [> processes to fall into the regular class, the same observations made about enabling hold:

a) When the disabling is the outermost operator and its arguments are regular processes, then the whole behaviour expression is regular.
By merging the FSA of P and that of Q, so that each node of the FSA corresponding to P is merged to the initial node of the FSA corresponding to Q, we obtain a FSA for P[>Q. This proves the statement.

b) If disabling is involved within a recursive call and at least its left argument is finite-state, not containing the recursive call, the process still has regular maximal traces.
To see this, we first build A*, the FSA for the left argument of disabling, and thereafter a "special" FSA for the right argument, where there is a node marked with "x" for any recursive call.
We then make use of the merging technique seen for point a), being careful to substitute the "x" marked node with the initial node of A*: in this way, we have built a FSA for the result process.

To close our analysis about DELOTOS enriched with enabling and/or disabling, the case is left in which enabling and disabling are added together to DELOTOS: in this case the expressive power is not greater than DELOTOS + >>.

Proposition 7: *The subset DELOTOS + >> + [> has exactly the same expressive power of DELOTOS + >>.*

Proof: The exit operator alone does not add expressivity to disabling; in fact, if we have an **exit** operator in the subprocess P of P[>Q (the other case does not hurt), the maximal traces of P are added to the set of already known maximal traces of P[>Q. Since, for induction, the maximal traces of P form a context-free language, and the class of context-free languages is closed under union, then the maximal traces of P[>Q form a context-free language.
For the conditions of regularity given for enabling and disabling, therefore the cases of interest are those processes in which enabling and disabling interact recursively, for instance processes like:

process rec.en.dis[a,b,c] := ((a; rec.en.dis[a, b, c]) [> (b; **exit**)) >> c; **exit endproc**

Now, a process of the form (P [> Q) >> R, even containing recursion, has as maximal traces those of P [> Q linked with the maximal traces of R. As the class of context-free languages is closed under concatenation, this is sufficient to prove the proposition. The symmetric case, i.e.

the processes of the form (P >> Q) [> R, can be solved analogously. ♦

For DELOTOS + >> + [>, the restrictions to get regular processes are of course represented by the union of the constraints required in the two previous cases. For this class the algorithms for deciding bisimulation equivalence cannot in general be applied.

5.3 *Parallel composition operators*

Let us now consider DELOTOS enriched with the parallel composition operator; first, we analyze the two degenerate cases of parallel composition:

- *synchronization* (the operator ||), is used when two processes wish to proceed together and synchronize on all the gates,
- *interleaving* (the operator |||), is used when two processes wish to proceed in a completely independent way and never synchronize.

In treating parallel composition, we must point out that up to now we have considered process instantiation used also as renaming. Since parallel composition acts on the gates, renaming has a semantic side effect, so using DELOTOS instead of DELOTOS* may cause an increase in the expressive power of the subset.

5.3.1 *Synchronization*

As follows from the operational semantics, synchronization is insensitive to 'i' actions, i.e. it is not possible to synchronize on unobservable actions. So the 'i' actions present in two synchronized processes are interleaved, rather than performed together. In this sense, synchronization does not perform differently from general parallel composition, in which some actions are synchronized and some are interleaved.
Since we are interested in distinguishing synchronization from general parallelism, it is implicitly assumed that no unobservable action is present in the processes to be synchronized.

Under this assumption, it can be proved that DELOTOS* + || is as expressive as regular languages:

Proposition 8: The languages described by DELOTOS + // processes are regular.*

Proof: As the interesting cases regard the interaction between synchronization and recursion, we need to investigate only two interesting cases:

a) the synchronization is outside a recursive call, i.e.

```
process syn_out :=
x || y
where
  process x := P(x) endproc
  process y := Q(y) endproc
endproc
```

b) the synchronization is inside a recursive call, i.e.

```
process  x := P(x) || Q(x)
endproc
```

a) Let us make the assumption that both x and y express regular languages. If so, it is possible to build the finite state deterministic automaton (FSDA) that recognizes the traces of x and y.
It is easy to build the FSDA recognizing the traces of x || y: the FSDA for x || y is built starting from the FSDA's for x and y: if they do not have two arcs with the same labels outgoing from the initial node, the new automaton will collapse to one node with no outgoing arcs; otherwise, the automaton to be built will have this arc from the initial node. Now, we can repeat this procedure for the nodes reached in the original FSDA's for x and for y (unless they are, in the original automata, the initial nodes, in which case we can stop): if there are no arcs satisfying the conditions, the corresponding node in the new automaton will be final; if we reach a final node in one automaton, this will correspond to a final node in the automaton for x || y.
As a consequence, x || y is regular if x and y are regular; an inductive reasoning hence suffices to prove the proposition for point a).

b) This is the most interesting case, particularly when the recursion call appears in both processes that synchronize, such as in

```
process
   sync[a,b,c] := (a;b;sync[a,b,c] [] a;c;sync[a,b,c]) || a;b;sync[a,b,c] [] a;stop
endproc
```

which has the same maximal traces of:

```
process sync1[a,b,c] := a;b;sync1[a,b,c] [] a;stop endproc.
```

which is a regular process.

By structural induction, we can assume that P(x) and Q(x) in case b) are regular processes. Hence, the maximal traces of the two processes P(x) and Q(x) are the same of the processes P'(x) and Q'(x) respectively, defined as follows:

$$P'(x) = s_1;x \ [] ... [] \ s_n;x \ [] \ f$$

$$Q'(x) = t_1;x \ [] ... [] \ t_m;x \ [] \ f^*$$

where f and f* do not contain x and we have liberally extended the action prefix operator to sequences of actions.

Let us now consider the synchronization P(x) || Q(x): it is possible to distinguish two cases within the synchronization of the subterms $s_i;x \parallel t_j;x$ (the synchronizations involving f or f* produce only finite computations):

i) if no s_i matches completely with any t_j, then we obtain only finite strings.

ii) The most general case is left where s_i is a substring of t_j or viceversa, i.e. the synchronizations are of the form:

$$s_i;(x \parallel q_j;x) \text{ where } t_j = s_i \cdot q_j \quad \text{or}$$

$$t_j;(x \parallel q_i;x) \text{ where } s_i = t_j \cdot q_i \quad (q_j, q_i \text{ possibly empty})$$

From this assumption we obtain the following general form:

$$P'(x) \parallel Q'(x) = \underset{h}{[]} \ Fin_h \ [] \ \underset{k}{[]} \ p_k;(x \parallel r_k;x)$$

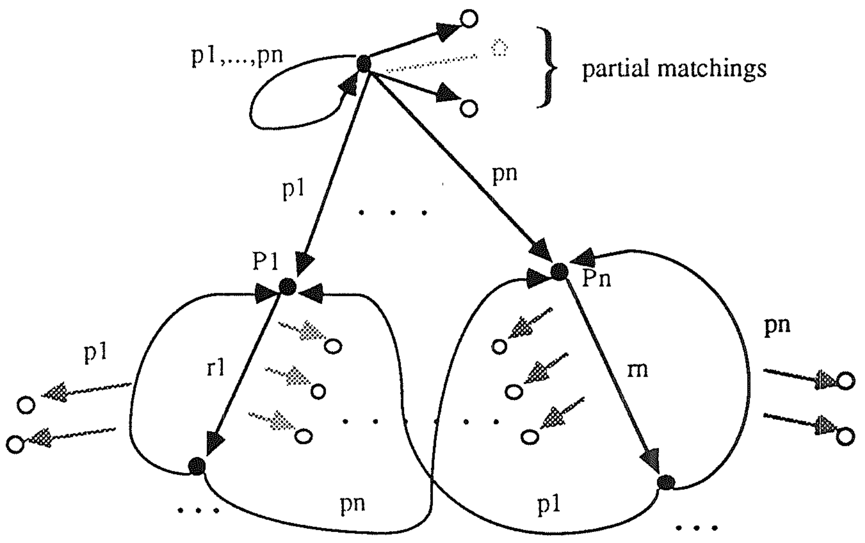
where Fin_h stands for the finite strings deriving from the partial matches of s_i's and t_j's.

To show that the language of the maximal traces is regular, we provide a procedure to build the FSA corresponding to the process P'(x) || Q'(x) [3].

From the initial node of the automaton there will be, possibly, a number of arcs going into final nodes, corresponding to the finite traces Fin_h, and outgoing arcs marked with the p_k's satisfying the above matching condition (by arc marked with p_k we intend here a sequence of arcs marked with the actions of p_k). There are now two cases: if the maximal traces of $x \parallel r_k;x$ are the same of the starting process x, then the arc marked with p_k will sink back into the initial node; otherwise, we introduce a new node P_k in which this arc sinks.

From this node, the traces will follow the pattern of r_k and could possibly be blocked at any actions of r_k, due to the semantics of synchronization, giving rise, in the automaton, to branches towards final states. When r_k is completed, the automaton has to proceed with the initial parts of the maximal traces of x itself, so there will be the same outgoing arcs of the initial node: the ones corresponding to the partial matches will sink into final states, the other marked with the p_k's will go back to the nodes P_k's, since after p_k only r_k is possible (see Fig. 2). ♦

[3] Considering only maximal traces does not, in general, preserve bisimulation, since the branching structure of the synchronized processes is lost.



Note: ○ are the final states

Figure 2

5.3.2 *Interleaving*

Intuitively, interleaving is rather powerful, meaning that it shuffles in any possible way the traces of the two processes to which it is applied; the following proposition formalizes this intuition:

Proposition 9: *The Basic LOTOS subset DELOTOS + ||| can express (some) context sensitive languages.*

Proof: To show the assumption, we refer to the results on process algebras obtained by Bergstra and Klop [BeK84].
Let D be a finite set of actions, which we can consider as data values, and $\underline{D} = \{\underline{d} \mid d \in D\}$. Over D we can define a *bag*, that is a set of values on which two operations (per value) are possible:

d: put d into the bag
$\underline{d}$: take d from the bag.

It is easy to specify the behaviour of a bag by means of a process in DELOTOS+|||:

process bag $[D \cup \underline{D}] := [\,]_{d \in D}\ d;(\underline{d}\ |||\ \text{bag}\ [D \cup \underline{D}])$ **endproc**

It can be shown [BeK84] that a bag whose domain contains at least two elements cannot be recursively defined in the algebra with signature (+, ·). This algebra corresponds to our subset DELOTOS + >>, which, as we already saw, is equivalent to the context-free grammars. Thus, DELOTOS + ||| is strictly more expressive than context-free languages.

On the other hand, the language CS = $\{a^nb^nc^n \mid n\geq 1\}$ cannot be produced as the set of maximal traces of a DELOTOS + ||| process, because, due to the definition of the operational semantics of interleaving, if in the set of maximal traces of a DELOTOS + ||| process there are strings of the language CS, there must be also some permutations of theirs: so DELOTOS + ||| is not able to represent all the context-sensitive languages.
Moreover, if we consider the word problem for a language defined by a DELOTOS + ||| process, it is shown to be decidable: in fact, all the computations of a process of length at most n can be found unfolding the recursion until we have traces longer than n, because the interleaving behaves in a monotonic way and only longer traces can be obtained by unfolding. This can be considered as a decision procedure for the word problem.

In this way we have also shown that DELOTOS + ||| is not able to describe phrase structure languages, for which the word problem is undecidable. ♦

5.3.2 *General parallel composition*

The former result regarding context sensitivity can be extended to parallel composition, as interleaving ||| is a particular case of | G | (G = ∅): actually, we are able to give a more precise result:

Proposition 10: *DELOTOS* + | G | is strictly more expressive than DELOTOS* + |||.*

Proof: Again, we use here some results from [BeK84], where it is shown that, supposing X a not finite process recursively defined on the signature +, ·, |||, X must have an infinite regular (eventually periodic) trace.
If we substitute the composition '·' with its restricted form action prefix ';' (as the first member must be a single action), it is clear that the previous result still holds and, at the same time, it is not difficult to see that the algebra defined on the signature +, ;, ||| corresponds exactly to DELOTOS+|||.
Let us now examine the maximal traces of the DELOTOS* + |G| process:

process par [a,b] := (a; b; par [a,b] | b | a; b; par [a,b]) [] **exit endproc**

The maximal traces of this process are of the form:

aabd
aabaaaabd
$a^2ba^4 \ldots a^{2n}bd$

If we call p_1 the first trace aabd, the n-th trace p_n is defined as $\{p_{n-1}$ - 'd'$\}a^{2n}bd$. Using the so called "uvwxy" theorem, it is easy to show that this language cannot be context-free.
On the other hand, the infinite trace clearly is not eventually periodic: this fact, together with the previous observation, shows that par [a,b] cannot be described using only interleaving without parallel composition. ♦

For what concerns the restrictions on DELOTOS* + |G| to represent only regular processes, we have to mention an interesting result from Park [Par81], originally given for the *fair merge* operator but easily extendable to interleaving:

a) When interleaving is the outermost operator in a behaviour expression and there are not mutual recursive calls between the arguments (which describe regular languages), DELOTOS + ||| still describes regular languages.

This result and the analogous one relative to synchronization (see above) can indeed be extended to DELOTOS* + | G |.

b) A process with a parallel composition at the "top level" and not involved into recursive calls of the regular arguments has regularly behaving maximal traces.
In fact, to build the FSDA corresponding to parallel composition, when in both the processes we meet a gate in G we use the building technique previously exploited for synchronization, otherwise we proceed using the technique introduced by Park for the fair merge, suitably adapted for interleaving.

5.3.4 *Parallel composition used as enabling*

For parallel composition it is not possible in general, as it happens for enabling and disabling, to have a recursive call in one of the two arguments without losing regularity. It happens, however, that some particular forms of parallel composition with recursive calls in one of the two arguments maintain regularity: these are the forms which have a behaviour similar to that of the enabling operator. Actually, the following equivalence holds:

a; b; c; **exit** >> Q ≈ **hide** t **in** (a; b; c; t; **stop** |[t]| t; Q)

Where the action on gate t synchronizes the end of the left process with the beginning of the right one, and acts as the fictitious d gate in the enabling.

Hence we can use the regularity conditions for the enabling to enlarge the cases of regular processes involving parallel composition; in particular, the interested condition is:
"*If enabling is present into recursion, it is sufficient that at least its left argument is a finite-state process, not containing the recursive call*".

It is now straightforward to give the following condition:

a process of the form:

process x := P | G | Q(x) **endproc**

is regular if all the following hold:

i) P is a regular process, not containing the recursive call x;

ii) G is the set of actions which are performed as last actions by P, and actions in G are not performed before by P (that is, the gates in G appear only before a **stop** operator in P);

iii) The set of first actions performed by Q(x) is contained in G.

It can be noted that the last point can be weakened in the following way:

iii') Any possible execution of Q(x) has a finite prefix of actions not in G, followed by an action in G, i.e., the synchronization action is preceded by a finite number of actions not in G.

Therefore, a process like the following:

process R := (Q [] a;b; **stop**) | [b,c] | (e; f; b; (R [] f; e; R) [] f; e; c; (R [] e; f; R))
where process Q := (d; Q [] a;c; **stop**) **endproc**
endproc

is regular.

Note that the conditions above can be expressed syntactically and can be checked by a textual analysis.

The same reasoning can be applied to define a class of process definitions using the parallel composition operator, which generate context free languages: let us define the subclass DPCLOTOS (for *Decidable Parallel Composition LOTOS*) of DELOTOS+|G|, where parallel composition appear only in the form:

process x := P(x) | G | Q(x) **endproc**

where:

- P(x), Q(x) are themselves DPCLOTOS processes, calling recursively x;
- G is the set of actions which are performed as last actions by P(x), and actions in G are not performed before by Bexp1 (that is, the gates in G appear only before a **stop** operator in P(x));
- The set of first actions performed by Q(x) is contained in G

Proposition 11. *DPCLOTOS* processes generate context-free languages.

Proof: Obvious, by translating the parallel composition into the corresponding enabling expression, and by Proposition 5.

For the same reason, processes in the classes DPCLOTOS+>>, DPCLOTOS+[>,

DPCLOTOS+>>+[> generate context-free languages.

5.4 *Hiding*

About the hiding operator, we can observe that it can be seen as a particular case of "renaming" and therefore behaves as a syntactical substitution when no operator acting on gates is present in its scope and as a semantic substitution otherwise.
Hence we can conclude that adding hiding to Basic LOTOS subsets which do not contain synchronization, parallel composition or process instantiation used as renaming does not change the expressive power, while adding hiding to all the other subsets may generally increase it.

6. CONCLUSIONS

The results obtained prove that all the LOTOS operators, except enabling and disabling, are needed to reach the full power of Turing Machines. On the other hand, enabling can take the place of hiding and of the renaming functionality of process instantiation in achieving the power of Turing Machines.
Thereafter, we have pointed out a regular subset, called DELOTOS, and, starting from this, we have studied the expressive power of the subsets obtained by enriching DELOTOS with the other operators (see the schema in Appendix A).
As most verification tools are designed for finite-state specifications, we have provided some sufficient conditions on the static structure of the processes (see the schema in Appendix B) in order to preserve regularity: it is in fact impossible to give general necessary conditions.
We believe that designers and users of verification tools can take advantage from our work: on one hand, we give precise indications on the expressive power of Basic LOTOS subsets and, thus, on the problems decidable on them, on the other hand we individuate a wide range of regular processes which can be submitted safely as input to the tools working on finite-state objects.

ACKNOWLEDGEMENTS

We would like to thank Gioia Ristori for helpful discussions about this work.

REFERENCES

[Ail86] Ailloud, G.: Verification in Ecrins of LOTOS Programs, in Esprit/Sedos Project Report N.

89.2: Toward practical Verification of LOTOS Specifications, 1986.

[BoB87] Bolognesi, T., Brinskma, E.: Introduction to the ISO Specification Language LOTOS, *Computer Networks & ISDN Systems*, 14, 1, 25-29 (1987).

[BaB90] Barbeau, M., Bochmann, G. V.: Extension of the Karp and Miller Procedure to LOTOS Specifications, *Proc. 2nd Workshop on Computer-Aided Verification,* Vol. 1, 1990.

[BBK88] Baeten, J. C. M., Bergstra, J. A., Klop, J. W.: Decidability of Bisimulation Equivalence for Processes generating Context-Free Languages, *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Model for Concurrency*, Noordwijkerhout, 1988.

[BoC89] Bolognesi, T., Caneve, M.: Equivalence Verification: Theory, Algorithms and a Tool, in van Eijk, P.H.J., Vissers C.A., Diaz M., eds., *The Formal Description Technique LOTOS*, pp. 303-326, North-Holland, 1989.

[BeK84] Bergstra, J. A.,. Klop, J. W: The Algebra of Recursively Defined Processes and the Algebra of Regular Processes, *ICALP '84, LNCS vol. 172*, pp. 82-94, 1984.

[DeI91] De Francesco, N., Inverardi, P.: A Semantics Driven Method to Check the Finiteness of CCS Processes, *Proc. 3rd Workshop on Computer-Aided Verification*, pp.342-354, 1991.

[DIN91] De Nicola, R., Inverardi, P., Nesi, M.: Equational Reasoning about LOTOS Specifications: A Rewriting Approach", *Proc. Sixth Workshop on Software Specification and Design*, pp. 148-155 IEEE, 1991.

[FGL89] Fantechi, A., Gnesi, S., Laneve, C.: An Expressive Temporal Logic for Basic LOTOS, *Formal Description Techniques - II*, pp. 261-276, North-Holland, 1989.

[GoM84] Goltz, U., Mycroft, S.: On the Relationships of CCS and Petri Nets, *ICALP '84, LNCS vol. 172*, pp. 196-208, 1984

[GaN89] Garavel, H., Najm, E.: TILT: from LOTOS to Labelled Transition Systems, in van Eijk, P.H.J., Vissers C.A., Diaz M., eds., *The Formal Description Technique LOTOS*, pp. 327-336, North-Holland, 1989.

[GaS90] Garavel, H., Sifakis, J.: Compilation and Verification of LOTOS Specification, *Protocol Specification, Testing and Verification, X*, pp. 379-394, North-Holland , 1990.

[HüS91] Hüttel, H., Stirling, C.: Actions Speak Louder than Words: Proving Bisimilarity for Context-free Processes, *Proc. LICS 91*, Computer Society Press, 1991.

[HoU69] Hopcroft, J. E., Ullman, J. D.: Formal Languages and their Relation to Automata, Addison-Wensley, 1969.

[JLL77] Jones, N. D., Landweber, L. H., Lien, Y. E.: Complexity of Some Problems in Petri Nets, Theor. Comp. Sci. 4, 295-297 (1977).

[Mil80] Milner, R.: A Calculus of Communicating Systems , *LNCS vol. 92*, 1980.

[MaV89] Madeleine, E., Vergamini, D.: AUTO: A Verification Tool for Distributed Systems Using Reduction of Finite Automata Networks, *Formal Description Techniques - II*, pp.61-66, North-Holland, 1989.

[MaV90] Madeleine, E., Vergamini, D.: Finiteness Conditions and Structural Construction of Automata for all Process Algebras, *Proc. 2nd Workshop on Computer-Aided Verification*, Vol. 1, 1990.

[Niw84] Niwinski, D.: Fixed-point Characterization of Context-free ω-languages", *Information and Control* 61, 247-276 (1984).

[Par81] D. Park, D.: Concurrency and automata on infinite sequences, *Proc. 5th GI Conf., LNCS vol. 104*, pp. 167-183, 1981.

[Tau89] Taubner, D.: Finite Representations of CCS and TCSP Programs by Automata and Petri Nets, *LNCS vol. 369*, 1989.

[VSS88] Vissers, C. A., Scollo, G., van Sinderen, M.:Architecture and Specification Styles in Formal Descriptions of Distributed Systems, *Protocol Specification, Testing and Verification, VIII*, pp.189-204, North-Holland, 1988.

APPENDIX A

Here is shown a table which summarizes the results obtained about the expressive power of Basic LOTOS subsets, in relation with the classes of formal languages; the decidability of some problems on these classes is also reported.

LOTOS SUBSETS		EXPRESSIVE POWER				
		Regular (finite state)	Context Free	Context Sensitive	Turing Machines	
DELOTOS*	DELOTOS +hiding	●				
+enabling	+enabling		●			
+disabling	+disabling		● 1)			
+synchroniz.		● (- i)				
+interleaving	+interleaving			● 3)		
+parallel composition				● 2)		
+enabling +par. comp.	+enabling +par. comp.				●	
	+parallel composition				●	
	DPCLOTOS		●			
		Decidable	*Undecidable*	*Undecidable*	*Undecidable*	Max. trace equivalence
		Decidable	*Decidable (enabling)*	*Undecidable?*	*Undecidable*	Bisimulation
		Decidable	*Decidable*	*Decidable*	*Undecidable*	Admitted computation
						PROBLEMS

Notes:

1) a subclass of Context-Free Languages

2) a class containing some Context-Sensitive Languages

3) a subclass of the previous one, still containing some Context-Sensitive Languages

4) (-i) with only observable actions

Table 2. The expressive Power of LOTOS subsets

APPENDIX B

The following definition of *regular process* summarizes the syntactical restrictions to achieve a finite-state behaviour.

A regular behaviour expression has the following syntax:

stop | **exit** | P | x[]y | x>>y | x[>y | x|G|y

where x and y are regular behaviour expressions and P is the identifier of a regular process, or a free identifier.
We will indicate by RBE a regular behaviour expression with no free identifier and by RBE(X) a regular behaviour expression with occurrences of free identifier X.

A regular process RP is a process defined by one of the following definitions:

process RP := RBE (RP) **endproc**

process RP := RBE1 >> RBE2 (RP) **endproc**

process RP := RBE1 [> RBE2 (RP) **endproc**

process RP := RBE1 (RP) || RBE2 (RP) **endproc** where RBE1 and RBE2 do not perform internal actions

process RP := RBE1 |G| RBE2 (RP) **endproc** where G is the set of actions performed as last actions by RBE1, and actions in G are not performed before by RBE1 (that is, the gates in G appear only before a **stop** operator in RBE1). Any possible execution of RBE1 has a finite prefix of actions not in G, followed by an action in G

APPENDIX C

The semantics of Basic LOTOS is based on the concept of "Labelled Transition Systems" (LTS in the following). A LTS is a 4-uple $(S, Act, \{R_x, x \in Act\}, s_o)$ such that S is a set of states, Act is a set of actions, $R_x \subseteq S \times S$, $s_o \in S$ is the initial state.
We will use the notation B_1-g->B_2 to mean that $(B_1,B_2) \in R_g$ and we will say that the system in the state B_1 is able to perform action 'g' and transform in the state B_2.
In Table 1 we present the actions and operators of Basic LOTOS and their related operational semantics. From the operational semantics we can observe that the transitions of programs are labelled by the observable actions (marked as o) and by the unobservable (i) actions. We also distinguish transitions which involve the successful termination action 'd'. Since observable actions occur at gates, we can define the set Act of actions for Basic LOTOS as $Gates \cup \{i\} \cup \{d\}$.
On LTSs several equivalence relations are defined; among them we consider the ***bisimulation*** the *maximal trace* equivalences.
We will consider only "strong" equivalences, which do not distinguish between observable and unobservable actions: it is the same as considering processes as only performing "concrete" or observable actions. The corresponding weak equivalences, which forget unobservable action, can be found in literature.

Definition C.1: $\sigma \in Act^* \cup Act^\omega$ is a *maximal trace* of a process $p \in \mathcal{P}$, if the length of σ is infinite, or there exist actions $a_1, a_2, ..., a_n, ... \in$ Act and processes $p_1, p_2, ..., p_n, ... \in \mathcal{P}$ such that $\sigma = a_1 \cdot a_2 \cdot ... \cdot a_n$, and p --$a_1$-->$p_1$ --a_2-->p_2 ....--a_n-->p_n and do not exist $a \in Act$, $q \in \mathcal{P}$ such that $p_n \xrightarrow{a} q$.

Definition C.2: Two processes p and q are said *maximal trace equivalence* if the related sets of maximal traces are equal, and we write p~q.

Definition C.3: A *bisimulation* R is a binary relation on $\mathcal{P}$ such that whenever pRq and $a \in \Sigma$ then:
i) $p \xrightarrow{a} p' \Rightarrow \exists q'.\ q \xrightarrow{a} q'$ and p'Rq',
ii) $q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p'$ and q'Rp'.

Definition C.4: Two processes p and q are said *bisimulation equivalent* if and only if there exists a bisimulation R with pRq, and we write $p \approx q$.

operator	syntax	operational semantics	informal meaning
Inaction	**stop**		denotes a process which cannot perform any action.
Unobservable action	i;B	i;B -i->B	models an event internal to the process.
Observable action	g;B	g;B -g->B	models a process which can perform the transition g.
Choice	B1[]B2	B1-odi->B1' *implies* B1[]B2 -odi->B1' B2-odi->B2' *implies* B1[]B2 -odi->B2'	the actions of the process are the set of possible actions of B1 and B2.
Parallel composition	B1|S|B2	B1-oi->B1' *and* gate(oi)$\notin$ S *implies* B1|S|B2-oi-> B1'|S|B2 B2-oi->B2' *and* gate(oi)$\notin$ S *implies* B1|S|B2-oi-> B1|S|B2' B1-od->B1' *and* B2-od->B2' *and* gate(od)$\in$ S *implies* B1|S|B2-od-> B1'|S|B2'	the parallel composition forces the subprocesses to interact at every gate in the set S.
Successful termination	**exit**	**exit** -d->**stop**	this operator models the process able to emit a successful termination signal
Sequential composition (enabling)	B1>>B2	B1 -d->B1' *implies* B1>>B2 -i->B2 B1 -oi->B1' *implies* B1>>B2 -oi->B1'>>B2	this operator allows sequential process composition.
Disabling	B1[>B2	B1 -d->B1' *implies* B1[>B2 -d->B1' B1-oi->B1' *implies* B1[>B2 -oi->B1'[>B2 B2-odi->B2' *implies* B1[>B2 -odi-> B2'	this operator allows process B1 to be disabled by process B2.
Hiding	**hide** S **in** B	B-odi->B' *and* gate(odi)$\notin$ S *implies* **hide** S **in** B-odi-> **hide** S **in** B' B-o->B' *and* gate(o) $\in$ S *implies* **hide** S **in** B-i-> **hide** S **in** B'	this operator allows transitions at gates in S to be internalized.
Process instantiation	p[g_1,...g_n]	Bp[g_1/h_1,...g_n/h_n] -odi->B' *implies* p[g_1,...g_n] -odi->B'	the transitions of a process instantiation are those of the body of the process declaration (Bp) which substitute formal parameters (h_i) with actual ones.

Table 3. Syntax and semantics of Basic LOTOS