An Expressive Trace Theory for LOTOS

S. Gallouzi[a], L. Logrippo[a], A. Obaid[b]

[a]University of Ottawa, Protocols Research Group, Department of Computer Science, Ottawa, Ontario, Canada, K1N 6N5

[b]Université du Québec à Hull, Département d'informatique, Case postale 1250, succursale B, Hull, Québec, Canada, J8X 3X7

Abstract

Trace theory concepts are added to LOTOS and appropriate composition functions on traces are defined. The result is a trace theory that is suitable for reasoning about communication sequences of LOTOS behavior expressions. This is illustrated by defining a trace set semantics and a trace-based proof system for LOTOS. An example, consisting of the proof of a property of traces of an unbounded reliable buffer is presented.

1 INTRODUCTION

The concepts of "trace" or "history", and invariant properties of traces and histories have had an important role in research on proof methods for concurrent processes. Proofs of processes based on these concepts have been developed in many papers, of which some of the best known are [HO83, Hoa85]. In LOTOS, instead, the most commonly used proof methods are based on the concept of bisimulation [Par81, Mil80, Mil89, Bri88].

This paper is a contribution towards a proof theory based on trace concepts for LOTOS. We start by defining traces and several useful operators on them, independently of concepts of behavior expressions. We elaborate on this point beyond what is needed for the rest of the paper because we want to show that LOTOS traces have many interesting properties that can be used in proofs. We then develop a trace theory for "basic LOTOS" behavior expressions. We end by providing an example, showing the proof of an invariant property of traces for an unbounded buffer process. The property proven is that at any time during the lifetime of the buffer, the number of outputs does not exceed the number of inputs. This property could not have been proven directly by using bisimulation: the typical way of proceeding by using this technique is to construct another simple process, obviously having this property, and to prove it equivalent to the given process.

An advantage of proof methods based on the idea of invariant properties is the compositionality they exhibit. Both [HO83] and [Hoa85] provide rules for proving properties

of a process on the basis of the properties of component processes. Compositionality is a very valuable characteristic of proof methods, especially to help formalizing the design process, by allowing the simultaneous construction of processes and proofs.

2 TRACE THEORY

2.1 Traces

A *trace* is a finite-length sequence of *symbols*. Symbols are denoted by identifiers and may represent atomic statements of a (concurrent) system, often referred to as actions or events. Each trace may then be interpreted as a sequence of (atomic) interactions that may take place between a process and its environment. A trace will be denoted as follows

$\langle\rangle$ The empty trace containing no symbols.

$\langle a \rangle$ A trace containing only the symbol a.

$\langle a, b \rangle$ A trace containing two symbols, a followed by b.

$\langle a.t \rangle$ A trace containing the symbol a followed by the trace t.

In other words, a and t, in the previous line, are the *head* and *tail* of the trace, respectively. This notation suggests that two traces are said to be *equal* if they are both empty or their heads and tails are equal.

Every trace is associated with some *alphabet*; a finite set of symbols. We interpret each symbol from the trace's alphabet as a possible interaction between a process and its environment. For each alphabet A, A^* denotes the set of all traces, including the empty trace $\langle\rangle$, which are formed from symbols in A. We will need to handle a special symbol denoted by δ, which is used in LOTOS to indicate the successful termination of a process which engages in it, via the **exit** behavior expression. Therefore this symbol can appear only at the end of a trace. We will thus let A_δ be the alphabet $A \cup \{\delta\}$ and A_δ^* be the set of all traces of symbols of A_δ, where δ may only occur at the end of a trace. Such traces are said to be *well-formed*. Note that the (well-applied) composition of well-formed traces using the functions and operations introduced in this chapter will always yield (a set of) well-formed traces.

In the remainder of this section we define the most important operators and composition functions on traces, and state their chief properties. This part has borrowed many ideas from [Hoa85, vdS85]. We shall omit the proofs of the properties. However, the reader may refer to [Gal89] for a more detailed treatment of these operators and their properties. We will also use the following conventions: a, b, c, ... stand for symbols, s, t, u, ... stand for traces, T, U, V, ... stand for sets of traces, and A, B, C, ... stand for alphabets.

2.2 Basic operators on traces

Projection

The *projection* of a trace t on an alphabet A, denoted by $t \lfloor A$, can be obtained from t by omitting all symbols outside A. For example $\langle a, b, c, d, a \rangle \lfloor \{a, b\} = \langle a, b, a \rangle$. It satisfies the following properties

P 2.1 $t\lfloor\{\} = \langle\rangle$, $t\lfloor A\lfloor B = t\lfloor(A \cap B) = t\lfloor B\lfloor A$, *and* $t \in A^* \equiv t = t\lfloor A$

Next we extend the definition of projection to operate on sets of traces as well. It is defined by $T\lfloor A = \{t\lfloor A | t \in T\}$. Projection of sets of traces is monotonic w.r.t. inclusion ordering ($\subseteq$) over sets

P 2.2 $U \subseteq V \Rightarrow U\lfloor A \subseteq V\lfloor A$

Length

The *length* of a trace t is denoted $|t|$. For example $|\langle a, b, c\rangle| = 3$ and $|\langle\rangle| = 0$. The number of occurrences of symbols from A in a trace t can be counted by $|t\lfloor A|$. Therefore, we define $t \downarrow A = |t\lfloor A|$. If A contains a single symbol a, the number of its occurrences in the trace t is denoted $t \downarrow a$ (instead of $t \downarrow \{a\}$). The following property is an immediate consequence of the above definition and P 2.1

P 2.3 $(t\lfloor A) \downarrow B = t \downarrow (A \cap B) = (t\lfloor B) \downarrow A$

We also have $t \downarrow (A \cup B) = t \downarrow A + t \downarrow B - t \downarrow (A \cap B)$

Concatenation

The *concatenation* operator, denoted by "$\frown$", constructs a trace by putting two traces, u and v, together in this order. For example $\langle a, b, c\rangle \frown \langle d, c\rangle = \langle a, b, c, d, c\rangle$ and $\langle a, b\rangle \frown \langle\rangle = \langle a, b\rangle$. Concatenation should always yield well-formed traces, and so it is undefined for traces associated with alphabets containing the special symbol δ in its first argument. Clearly it is associative but not commutative, and has $\langle\rangle$ as its unit. It also satisfies the following properties

P 2.4 $(u \frown v)\lfloor A = (u\lfloor A) \frown (v\lfloor A)$ *and* $(u \frown v) \downarrow A = (u \downarrow A) + (v \downarrow A)$

We can now define t^n, the n^{th} trace power of t where n is a natural number, as n copies of t concatenated to each other. We shall write t^* and t^+ to denote n or more copies of t concatenated to each other, where n is equal to 0 and 1 respectively.

Prefix

We write $u \preceq v$, to denote that u is an initial segment of v, often called *prefix*. This means that $|u| \leq |v|$ and the two traces are identical in their first $|u|$ symbols. For example $\langle a, b\rangle \preceq \langle a, b, a, c\rangle$. We call u a prefix of t if and only if $\exists v \cdot t = u \frown v$. If $v \neq \langle\rangle$ we can write $u \prec t$. Clearly $\preceq$ is a partial ordering relation. It follows that a function f that maps traces to traces is said to be *monotonic* if it preserves the ordering $\preceq$ ($f(u) \preceq f(v)$ whenever $u \preceq v$). For example projection is monotonic

P 2.5 $u \preceq v \Rightarrow (u\lfloor A) \preceq (v\lfloor A)$

Concatenation is monotonic in its second argument, keeping the first argument constant

P 2.6 $u \preceq v \Rightarrow (t \frown u) \preceq (t \frown v)$

The set of traces that contains all the prefixes of a trace t is called *prefix closure* of t, and is denoted by $pref(t)$. We have $pref(t) = \{u | u \preceq t\}$.

P 2.7 $u \preceq v \equiv pref(u) \subseteq pref(v)$

The prefix closure of a set of traces T is defined as $pref(T) = \bigcup_{t \in T} pref(t)$.

P 2.8 $T \subseteq pref(T)$ *and* $\{\langle\rangle\} \subseteq pref(T)$

A set T is called *prefix-closed* if $T = pref(T)$. The $pref$ of sets of traces is monotonic for $\subseteq$-ordering, in that

P 2.9 $U \subseteq V \equiv pref(U) \subseteq pref(V)$

The last two properties describe the distribution of projection through prefix closures

P 2.10 $pref(T)\lfloor A = pref(T\lfloor A)$

P 2.11 *The projection of a prefix-closed set on any alphabet is prefix-closed.*

Containment

The *contiguous containment* operator, denoted by "*in*", is used to express the fact that a trace u is a contiguous subsequence of a trace v (not necessarily initial). We write u *in* v if and only if $\exists s, t \cdot v = s \frown u \frown t$. Note that $u \preceq v \Rightarrow u$ *in* v. If u is a subsequence of v, not necessarily a contiguous one, we write $u \sqsubseteq v$; this is defined as follows $u \sqsubseteq v \equiv \exists u_1, \ldots, u_n, v_1, \ldots, v_{n+1} \cdot v = v_1 \frown u_1 \frown v_2 \frown \ldots \frown u_n \frown v_{n+1} \wedge u = u_1 \frown \ldots \frown u_n$, where $n \geq 1$. For example $\langle a, d\rangle \sqsubseteq \langle b, a, c, d, a\rangle$. Clearly both relations are also partial orderings, and their least element is $\langle\rangle$. Moreover, projection is monotonic for these orderings, since it distributes through concatenation; they both satisfy property P 2.5. Also concatenation is monotonic in all its arguments for the $\sqsubseteq$-ordering

P 2.12 $u \sqsubseteq v \Rightarrow (t \frown u) \sqsubseteq (t \frown v)$
$u \sqsubseteq v \Rightarrow (u \frown t) \sqsubseteq (v \frown t)$

The $\sqsubseteq$-ordering is a generalization of the *in*-ordering, therefore

P 2.13 u *in* $v \Rightarrow u \sqsubseteq v$

Renaming

Let F be a function mapping symbols in an alphabet A to symbols in an alphabet B such that $F(a) = \delta$ iff $a = \delta$. We define the postfixed *renaming* operation over traces, as follows $\langle\rangle[F] = \langle\rangle$ and $\langle a.t\rangle[F] = \langle F(a).t[F]\rangle$. Thus if $t \in A^*_\delta$ then $t[F] \in B^*_\delta$. For example, if $F : \{a, b, c, d\} \longrightarrow \{a, c, e, f\}$ given by $F(a) = a$, $F(b) = c$, $F(c) = e$ and $F(d) = f$ is a renaming, we then have $\langle a, b, c, d\rangle[F] = \langle a, c, e, f\rangle$. We shall use convenient abbreviations in writing renamings explicitly. Thus $b_1/a_1, \ldots, b_n/a_n$ (where $a_1, \ldots, a_n$ are distinct symbols) stands for the renaming $F : \{a_1, \ldots, a_n\} \longrightarrow \{b_1, \ldots, b_n\}$ given by $F(a_i) = b_i$ if $a_i \in \{a_1, \ldots, a_n\}$ and $F(a) = a$ if $a \notin \{a_1, \ldots, a_n\}$. So in place of $t[F]$ above, we write $t[c/b, e/c, f/d]$. Identical renamings may be omitted in this notation.

Renaming is also monotonic

P 2.14 $u \preceq v \Rightarrow u[F] \preceq v[F]$

The following properties are restricted to the case when the renaming function F is one-to-one (injection)

P 2.15 $(t\lfloor A)[F] = (t[F])\lfloor F(A)$ *if F is one-to-one.*

P 2.16 $t \downarrow A = (t[F]) \downarrow F(A)$ *if F is one-to-one.*

2.3 Sequential composition

If the successful termination symbol δ does not occur at the end of a trace u (in which case it does not occur in u at all), the *sequential composition* of u and v, denoted $u \gg v$, is u. If δ does occur at the end of u, it is removed and the result is concatenated to v. For example $\langle a, b, c\rangle \gg \langle a\rangle = \langle a, b, c\rangle$ and $\langle a, b, a, \delta\rangle \gg \langle b, d\rangle = \langle a, b, a, b, d\rangle$.

This composition operator is monotonic in all its arguments

P 2.17 $u \preceq v \Rightarrow ((t \gg u) \preceq (t \gg v))$
$u \preceq v \Rightarrow ((u \gg t) \preceq (v \gg t))$

This property is also satisfied by the $\sqsubseteq$-ordering. Moreover, $\gg$ is clearly associative, strict (maps the empty trace to the empty trace) in its first argument, and has $\langle\delta\rangle$ as its unit.

P 2.18 $t \gg (u \gg v) = (t \gg u) \gg v$

P 2.19 $\langle\rangle \gg t = \langle\rangle$

P 2.20 $\langle\delta\rangle \gg t = t \gg \langle\delta\rangle = t$

We conclude this section by extending the definition of sequential composition to operate on sets of traces $U \gg V = \{u \gg v | u \in U \wedge v \in V\}$.

2.4 Disruption

The sequential composition uses δ as a glue which sticks two traces u and v together. We define another composition function, called *disruption*, that does not need the glue to stick a prefix of u and v together, and if the glue occurs, v cannot stick. It is denoted by $u[> v$, and can be defined as follows $u[> v = \{t \frown s | t \preceq u \wedge (if\ \neg(\langle\delta\rangle\ in\ t)\ then\ s = v\ else\ s = \langle\rangle)\}$. For example $\langle a, b, \delta\rangle[> \langle a\rangle = \{\langle a\rangle, \langle a, a\rangle, \langle a, b, a\rangle, \langle a, b, \delta\rangle\}$. As we shall see, this operator is useful to describe a situation where a sequence of actions can be interrupted by another sequence. Obviously, if the first sequence exits, disruption is no longer possible.

Disruption is not symmetric. However it enjoys a number of simple properties, including

P 2.21 $t \in u[> v \Rightarrow |t| \leq |u| + |v|$

P 2.22 $u \preceq v \Rightarrow u[> t \subseteq v[> t$

P 2.23 $\langle\rangle[> t = \{t\}$ *and* $t[> \langle\rangle = pref(t)$

Its effect on a singleton trace containing δ in its first argument is obvious

P 2.24 $\langle\delta\rangle[> t = \{\langle\delta\rangle, t\}$

The next property describes the distribution of projection through disruption

P 2.25 $(u[> v)\lfloor B_\delta = u\lfloor B_\delta [> v\lfloor B_\delta$

It follows that

P 2.26 $t \in u[> v \Rightarrow t \downarrow B_\delta \leq (u \downarrow B_\delta + v \downarrow B_\delta)$

We conclude this section by extending the definition of disruption to operate on sets of traces and listing the deriving properties. We have $U[> V = \bigcup_{u\in U, v\in V} u[> v$. Clearly, disruption is associative, monotonic for $\subseteq$, and that $pref$ distributes through $[>$ and the composition of two prefix-closed sets yields a prefix-closed set.

2.5 Merging

Traces are very well suited to represent all possible communication patterns between a process and its environment. We would like to be able to express relations between compound processes as equations between the traces of their components. In this section we define a composition function, called *merging*, to describe the parallel composition of two or more processes. This includes their mutual communication embodied by the common actions specified in their "synchronization alphabet" and the successful termination action. Each communication requires the participation of both processes. Recall that two LOTOS processes P and Q composed by means of the parallel composition operator $|[S]|$, where S is a sequence of actions will mutually interleave for actions not in S, mutually synchronize for the actions in S, and mutually synchronize on the "exit" action δ. If at any point a required synchronization is not possible, a deadlock occurs. This leads to the following auxiliary definition. Let

$\mathcal{M}(v_1, A, v_2) = \{t | v_i \sqsubseteq t \wedge v_i\lfloor A_\delta = t\lfloor A_\delta \wedge |t| = \sum_j |v_j| - (v_i \downarrow A_\delta)\}$

where $i, j = 1, 2$. Furthermore, for a set of traces T, trace $t \in T$ is said to be "*longest*" in T if $|t'| \leq |t|$ for any $t' \in T$. The *merging* of two traces u_1 and u_2 in the order imposed by a synchronization alphabet A, denoted by $u_1|A|u_2$, is the set of longest traces t such that there exists two prefixes v_1 and v_2 of u_1 and u_2, respectively, for which $t \in \mathcal{M}(v_1, A, v_2)$. We illustrate this definition with some examples as follows

$\langle a, b, \delta\rangle|\{a\}|\langle a, b\rangle = \mathcal{M}(\langle a, b\rangle, \{a\}, \langle a, b\rangle) = \{\langle a, b, b\rangle\}$
since
$\mathcal{M}(t, \{a\}, \langle\rangle) = \mathcal{M}(\langle\rangle, \{a\}, t) = \{\langle\rangle\}$ where $t \preceq \langle a, b, \delta\rangle$
$\mathcal{M}(\langle a\rangle, \{a\}, \langle a\rangle) = \{\langle a\rangle\}$
$\mathcal{M}(\langle a, b\rangle, \{a\}, \langle a\rangle) = \mathcal{M}(\langle a\rangle, \{a\}, \langle a, b\rangle) = \{\langle a, b\rangle\}$ etc.
Similarly, we can show that
$\langle a, b, a\rangle|\{b\}|\langle b, c\rangle = \{\langle a, b, a, c\rangle, \langle a, b, c, a\rangle\}$
$\langle a, b, \delta\rangle|\{\}|\langle c, \delta\rangle = \{\langle a, b, c, \delta\rangle, \langle a, c, b, \delta\rangle, \langle c, a, b, \delta\rangle\}$
$\langle a, b\rangle|\{a, b\}|\langle b, a\rangle = \{\langle\rangle\}$
$\langle a, b, c\rangle|\{b, d\}|\langle d, e\rangle = \{\langle a\rangle\}$

Notice that in the case of two traces with an empty synchronization alphabet, merging does not amount to interleaving as it is defined in [Hoa85]. They still have to "synchronize" on the special "exit" symbol δ. This ensures that the merging of well-formed traces yields a set of well-formed traces. Also note that merging may not allow (in some cases) the combination of all symbols from u_1 and u_2 into the resulting trace t; it may be *blocked* before it terminates. Blocking may even occur at the very beginning if the heads of u_1 and u_2 are in A and are not equal. In case of blocking, no further symbols from u_1 and u_2 can be combined, in which case we can write $block(u_1, A, u_2)$. Therefore we can define $block(u_1, A, u_2) \equiv \forall t \in u_1|A|u_2 \cdot \neg(u_1 \sqsubseteq t \wedge u_2 \sqsubseteq t)$. An alternative definition of "block" can be formalized using projection $block(u_1, A, u_2) \equiv \forall t \in u_1|A|u_2 \cdot t\lfloor A_\delta \prec u_1\lfloor A_\delta \vee t\lfloor A_\delta \prec u_2\lfloor A_\delta$. The first, fourth and fifth examples above are examples of blocking. Note that in the first line the blocking results by the fact that the traces cannot synchronize on δ. The following property is a consequence of this definition

P 2.27 $t \in u_1|A|u_2 \Rightarrow \exists v_1, v_2 \cdot v_i \preceq u_i \wedge t \in v_1|A|v_2 \wedge \neg block(v_1, A, v_2)$

The following properties are obvious consequences of the definition of merging

P 2.28 $t \in u_1|A|u_2 \Rightarrow |t| \leq |u_1| + |u_2|$

P 2.29 $t \in u_1|A|u_2 \wedge \neg block(u_1, A, u_2) \Rightarrow |t| = |u_1| + |u_2| - (u_1 \downarrow A_\delta)$

P 2.30 $t \in u_1|A|u_2 \wedge \neg block(u_1, A, u_2) \Rightarrow t \downarrow A_\delta = u_i \downarrow A_\delta$ *for* $i = 1, 2$.

The next property describes the prefix closure of merging in terms of its operands

P 2.31 $pref(u_1|A|u_2) = \{t | \exists v_1, v_2 \cdot v_i \preceq u_i \wedge t \in \mathcal{M}(v_1, A, v_2)\}$

An obvious consequence of this property is that merging preserves the inclusion ordering of the prefix closures of the merged traces

P 2.32 $v_1 \preceq u_1 \wedge v_2 \preceq u_2 \Rightarrow pref(v_1|A|v_2) \subseteq pref(u_1|A|u_2)$

Next we describe the distribution of projection through merging

P 2.33 $(u_1|A|u_2)\lfloor B \subseteq u_1\lfloor B|A|u_2\lfloor B$

Consider the following example $(\langle a, b, a\rangle|\{b\}|\langle b, c\rangle)\lfloor\{a, c\} = \{\langle a, a, c\rangle, \langle a, c, a\rangle\}$. If the projection is applied on the operands, some of the symbols of the synchronization alphabet might be omitted, and in this case merging would yield a larger set of traces $\langle a, b, a\rangle\lfloor\{a, c\}|\{b\}|\langle b, c\rangle\lfloor\{a, c\} = \{\langle a, a, c\rangle, \langle a, c, a\rangle, \langle c, a, a\rangle\}$. If no such symbols are omitted, projection would distribute through merging as follows

P 2.34 $(u_1|A|u_2)\lfloor B = u_1\lfloor B|A|u_2\lfloor B \qquad if\ A \subseteq B$

For example $(\langle a, b, a\rangle|\{b\}|\langle b, c\rangle)\lfloor\{a, b\} = \langle a, b, a\rangle\lfloor\{a, b\}|\{b\}|\langle b, c\rangle\lfloor\{a, b\} = \{\langle a, b, a\rangle\}$. It follows that

P 2.35 $t \in u_1|A|u_2 \Rightarrow t \downarrow B \leq (u_1 \downarrow B + u_2 \downarrow B)$

P 2.36 $t \in u_1|A|u_2 \wedge \neg block(u_1, A, u_2) \Rightarrow t \downarrow B = u_1 \downarrow B + u_2 \downarrow B - u_1 \downarrow (A \cap B)$

Finally, we extend the definition of merging to operate on sets of traces. We have $U|A|V = \bigcup_{u \in U, v \in V} u|A|v$. This definition enjoys a number of interesting properties. It provides a means for describing the associativity of merging.

P 2.37 $(U|A|V)|A|T = U|A|(V|A|T)$

However, it is not in general true that $(U|A|V)|B|T = U|A|(V|B|T)$. As a counterexample let $U = \{\langle a \rangle\}$, $V = \{\langle a \rangle\}$, $T = \{\langle b \rangle\}$, $A = \{b\}$, and $B = \{a\}$. Merging is monotonic for the $\subseteq$-ordering of sets of traces

P 2.38 $T \subseteq U \Rightarrow T|A|V \subseteq U|A|V$

The last two properties are related to merging and prefix closure.

P 2.39 *The merging of prefix-closed sets of traces is prefix-closed.*

P 2.40 $pref(U|A|V) = pref(U)|A|pref(V)$

3 TRACE SEMANTICS OF PROCESSES

3.1 Traces of processes

A process definition describes the behavior pattern of a process, by defining the sequences of observable actions that may occur (be observed), over a finite set of actions. The latter is the *process alphabet*, denoted $\alpha(P)$ for a given process P. The behavior of a process up to some moment in time can be recorded, by its environment, as a finite-length sequence of observable actions in which the process has participated, which we called a *trace*.

Nothing can be observed or recorded before a process has engaged in an observable action. This is represented by the empty trace $\langle\rangle$, which every process has as its shortest possible trace. Therefore the complete set of all possible traces, often called the *trace set*, of any process contains at least the empty trace. Also note that a process may reach a point where it cannot offer anything to the environment; it may reach a deadlock state or engage in an unbounded sequence of unseen actions. Once again nothing can be observed and only the empty trace can be recorded. For example, the only trace of the inactive process **stop** is $\langle\rangle$, so its trace set is $\{\langle\rangle\}$.

The traces of the behavior of a process have finite length. However the trace set of a process can be infinite (i.e. recursive processes). Moreover not every action, in which a process is ready to engage at a given moment in time, will actually occur: the choice may depend on the environment in which the process is placed, or on the nondeterministic behavior of the process. Thus not every possible trace of a given process will actually be recorded every time the process is "executed". For example, a process that models an unreliable simplex buffer (USB) which accepts an input, and then nondeterministically either immediately outputs what was input or loses it

process USB [*input*, *output*] :=
    input; (*output*; USB [*input*, *output*]
        []
        i; USB [*input*, *output*])
endproc

Its trace set is { $\langle\rangle$, $\langle input\rangle$, $\langle input,\ output\rangle$, $\langle input,\ input\rangle$, ... }. Only one of the two traces of length two can actually occur in an execution. The choice between them is nondeterministic. This trace set can be characterized as follows

$$\{t | t \in \{input, output\}^* \wedge t \downarrow output \leq t \downarrow input \wedge \neg(\langle output,\ output\rangle\ in\ t)\}$$

Note that **i** (the unobservable action) models the loss of input, which cannot be observed or recorded in the trace model.

3.2 From labeled transitions to traces

The operational semantics of LOTOS [ISO88, BB87] enables us to derive actions, in which a process (behavior expression) can engage, from the structure of the expression itself. Consider the result(s) of the execution of a trace $\langle a_1, a_2, \ldots, a_n\rangle$ on B. The result may be any B' for which $B \xrightarrow{s} B'$, where $s = i^{k_0}a_1 i^{k_1} a_2 i^{k_2} \ldots a_n i^{k_n}$ ($k_i \geq 0$) and i^k denotes a sequence of k **i**-actions, that is, an arbitrary number of internal events may occur before, among and after the a_i. This leads to the definition of the *trace relation* '$\Longrightarrow$', whose purpose is to abstract from the invisible actions that are derived by the transition relation when applied to a behavior expression.

Definition 3.1 (Trace Relation) *Let t denote a trace $\langle a_1, a_2, \ldots, a_n\rangle$ then we have $B \stackrel{t}{\Longrightarrow} B'$ whenever there exists a sequence $i^{k_0}a_1 i^{k_1} a_2 i^{k_2} \ldots a_n i^{k_n}$ ($k_i \geq 0$), denoted by s, such that $B \xrightarrow{s} B'$.*

This implies that $B \stackrel{\langle\rangle}{\Longrightarrow} B'$ whenever $B \xrightarrow{i^k} B'$ ($k \geq 0$), and that, for any B, $B \stackrel{\langle\rangle}{\Longrightarrow} B$ (in this case $k = 0$).

If $B \stackrel{t}{\Longrightarrow} B'$, we say that B' is an element of B/t (B after t). In other words if t is a trace of B then B/t is a set of behavior expressions whose elements behave the same as B from the time after B has been engaged in all the actions recorded in the trace t.

Definition 3.2 *Let B be a behavior expression and t a possible trace of B, then $B/t = \{B' | B \stackrel{t}{\Longrightarrow} B'\}$.*

If t is not a possible trace of B then B/t is undefined. The set $B/\langle\rangle$ contains B, together with, possibly, other behavior expressions that do not behave like B, since B may engage in an arbitrary number of internal events to reach a point where it does not accept what it originally did. This operator differs from the one in [Hoa85], which yields a single process as opposite to a set of processes.

The trace set of a behavior expression B, denoted $traces(B)$, consists of the action sequences derived by means of the trace relation.

Definition 3.3 *Let S be a set of behavior expressions, B a behavior expression. We define the* trace set *by means of the following two rules*

$traces(B) = \{t | \exists B' \cdot B \stackrel{t}{\Longrightarrow} B'\}$

$traces(S) = \bigcup_{B \in S} traces(B)$

From this definition it follows that

P 3.1 *The trace set of every process is prefix-closed.*

The trace set of B/t, provided t is a possible trace of B, can be defined as follows

P 3.2 $traces(B/t) = \{u | t \frown u \in traces(B)\} \qquad if\ t \in traces(B)$

Our analysis of recursive processes will be helped by defining a means of approximating the behavior of processes. We adapt the following definition from [Hoa85].

Definition 3.4 *If P is a non-divergent process and n a natural number, we define $(P \uparrow n)$ as a process which behaves like P for its first n observable actions, and then becomes* **stop**.

This definition will be useful in the analysis of recursive constructions. More specifically, we will apply it to carry out proofs by induction on recursively defined processes that do not diverge. Note that such processes are obviously guarded in the sense of [Mil89] (also see section 3.3 on page 11). Moreover, it was shown in [Hoa85] that an equation defining a guarded process has a unique solution.

3.3 Trace set semantics

In this section, we present a *trace set semantics* for LOTOS which consists of a set of rules and axioms, presented in a denotational style. this semantics provides a means to systematically derive the set of all possible traces (trace set) that a behavior expression may perform from the structure of the expression itself. More precisely, given an expression B, we use the function $traces(B)$ (definition 3.3) to yield that set. The result is the interpretation of a LOTOS text in terms of a trace set. A similar approach was used to develop the trace set semantics of CSP [Hoa85]. It can be argued that modeling the behavior of a process in terms of traces does not fully determine all possible observable aspects of its behavior. However, we are primarily interested (in this paper) in illustrating the expressive prower of the trace combinators introduced in the previous section.

In the definitions below, when a set of traces T for a given behavior is defined as a function of sets of traces $T_1, \ldots, T_n$ of other behaviors, T will be undefined whenever any of $T_1, \ldots, T_n$ is undefined.

As mentioned earlier, **stop** has only one trace

T 3.1 $traces(\mathbf{stop}) = \{\langle\rangle\}$

It was also mentioned that the first and the only action of **exit** is successful termination, so it has only two traces

T 3.2 $traces(\mathbf{exit}) = \{\langle\rangle, \langle\delta\rangle\}$

The action prefix behavior expression $(a;\ B)$, where a is an observable action, is capable of performing a and then behaving like B. Because the environment may not participate in action a, it must have the empty trace, and every nonempty trace must have a as its head and a possible trace of B as its tail

T 3.3 $traces(a;\ B) = \{\langle\rangle\} \cup \{\langle a \cdot t\rangle | t \in traces(B)\}$

If B is prefixed with an internal action **i**, then nothing can be observed, or recorded, until B engages into an observable action

T 3.4 $traces(\mathbf{i};\ B) = traces(B)$ *if B does not diverge.*

We say that B *diverges* if $\forall k \geq 0 \cdot \exists B' \cdot B \xrightarrow{i^k} B'$. This means that B can progress invisibly by engaging in an unbounded sequence of internal actions, and refuse to respond to the requests of its environment, in which case its trace set is undefined. Another kind of behavior expressions that can have a similar behavior is one that involves hiding. The latter provides a means to transform observable actions of a process into unobservable ones; it introduces **i**-actions implicitly in a behavior expression. Therefore a behavior expression B may diverge immediately on hiding actions in A, where A is a sequence of observable actions, in which case we shall write $diverges(\{B\}, \{A\})$. Thus we can define $diverges(S, \{A\}) \equiv \forall n. \exists t \in traces(S) \cap \{A\}^* \cdot |t| > n$, where S is a set of behavior expressions. The trace set of (**hide** A **in** B) can be obtained from the trace set of B by simply removing all occurrences of the actions in A from its elements

T 3.5 $traces(\mathbf{hide}\ A\ \mathbf{in}\ B) = \{t \lfloor (\alpha(B) - \{A\}) | t \in traces(B)\}$
if $\forall t \in traces(B). \neg diverges(B/t, \{A\})$

However, this is undefined if $\exists t \in traces(B) \cdot diverges(B/t, \{A\})$.

The last two rules are restricted to the case where the process does not diverge. This is not regarded as a serious limitation, since divergence is never an intentional result in the attempted definition of a process. Therefore, in order to complete a proof whenever one of these rules is used, it is necessary to include lemmas concerning nondivergence.

A *process instantiation* "$P\ [a_1, \ldots, a_n]$" refers to a process definition that must exist somewhere in the specification, whose behavior is described in terms of a list of actions $[a'_1, \ldots, a'_n]$ (called *formal gates*), which corresponds to the process alphabet. The behavior of instantiation "$P\ [a_1, \ldots, a_n]$" is obtained from the corresponding process definition behavior by renaming a'_i to become a_i, for $i = 1, \ldots, n$. Therefore

T 3.6 *If "***process** $P\ [a'_1, \ldots, a'_n] := B_p$ **endproc***" is a process definition*
then $traces(P\ [a_1, \ldots, a_n]) = \{t[a_1/a'_1, \ldots, a_n/a'_n] | t \in traces(B_p)\}$

This rule can be applied to discover the trace set of a recursively *guarded* process [Mil89], since recursion in LOTOS is achieved by process instantiation. The behavior of a recursive process P that is unguarded may entail an infinite sequence of instantiations of P, in which case P diverges without even performing internal actions, and so its trace set is undefined.

The behavior expression $(B_1 | [A] | B_2)$, where A is a sequence of observable actions, describes a composition in which B_1 and B_2 must synchronize on the actions in A. It is able to perform any action not in A that either component is ready to perform, or any action in A or δ that both components are ready to perform. Therefore, every trace of $(B_1 | [A] | B_2)$ depends on the traces of B_1 and B_2, and on A, as expressed by the following rule

T 3.7 $traces(B_1 | [A] | B_2) = traces(B_1) | \{A\} | traces(B_2)$

The behavior expression $(B_1 \gg B_2)$ describes a composition, often referred to as the enabling operator, that enables the execution of B_2 if B_1 terminates successfully by performing the successful termination action via the **exit** process. The occurrence of this transition is hidden; δ is transformed into the internal action **i**. Therefore, every trace of $(B_1 \gg B_2)$ must be the result of the sequential composition (over traces) of a pair of possible traces of B_1 and B_2, and conversely

T 3.8 $traces(B_1 \gg B_2) = traces(B_1) \gg traces(B_2)$ *if* B_2 *does not diverge.*

This rule is also restricted to the case where the second argument of the sequential composition operation does not diverge, since the latter introduces implicitly unobservable actions into a behavior expression, and so it is potentially divergent. As an example consider the following behavior $B = \mathbf{exit} \gg B$, where B invokes itself in the second part of the expression, and, in this case, it engages into an unbounded sequence of internal actions. Obviously, if B_1 diverges, then so does $(B_1 \gg B_2)$. However, this is not mentioned in the rule above, because, in this case, the sequential composition of B_1 and B_2 does not, specifically, yield such behavior.

The behavior expression $(B_1[> B_2)$ describes another kind of composition, often referred to as the disabling operator, that allows the occurrence of an initial action of B_2 at each point of the execution of B_1, in which case control is transferred to B_2, leaving B_1. Once B_1 has terminated successfully, B_2 can no longer disrupt B_1. Hence every trace of $(B_1[> B_2)$ is either a trace of B_1 with δ at the end or a trace constructed from a pair of possible traces of B_1 and B_2, respectively, by simply putting them together in this order. More formally, since the trace set of every process is prefix-closed we can write

T 3.9 $traces(B_1[> B_2) = traces(B_1)[> traces(B_2)$

4 TRACE-BASED PROOF SYSTEM

4.1 Satisfaction relation

The required behavior of a process can be described in terms of some observable aspects of its behavior. The most relevant observation, that we have mentioned, is the trace of actions that occur at some moment in time. This description is then a (trace) predicate S containing a free variable t that stands for an arbitrary trace of the process being defined with, possibly, some other free variables. Such a predicate will be referred to as a *trace specification.*

A trace specification describes the characteristics of a process's trace set. Therefore, there is a need for a notion of a process P satisfying its (trace) specification S. In this context, P can be viewed as an implementation of S, since it gives a more structured and detailed description of the system specified in S. The fact that a process P meets a behavior requirement S, can be modeled as a binary relation "$\models$". We write $\models$ in an infix manner and $P \models S$ may be read "P *satisfies* S", which means that S is a property that is true for every possible observation of the behavior of P; or, more formally $\forall t \cdot t \in traces(P) \Rightarrow S(t)$. In other words, S is true if its variable t takes values observed from the process P. Following Hoare, we will sometimes write $S(t)$ to indicate that a

specification S contains a free variable t, whenever there is a need to show how t may be substituted by some more elaborated expression.

The satisfaction relation $\models$ is defined by means of the same rules governing Hoare's **sat** relation [Hoa85]. These rules provide the most general properties of the satisfaction relation. They apply to all kinds of processes and all kinds of specifications, and are based on the structure of the specification. Our objective is to design a framework for the verification of statements about the behavior of LOTOS processes, which requires additional rules based on their structure. Such rules should permit proof of the correctness of a compound process to be constructed from a proof of correctness of its parts. We shall give these rules in the next section and refer to them as the *proof rules*. This leads to a Hoare-style proof system for LOTOS.

4.2 Proof rules

The proof rules permit the use of formal reasoning to ensure that a LOTOS behavior expression B meets its specification S. They also provide other means of characterizing processes, in that the behavior of a process can be modeled by logical assertions on its traces and, possibly, some other observable aspects of that.

The reader should beware that, in the list of rules given below, different occurrences of the symbol t in the same rule may refer to different traces. Also recall that, as stated at the beginning of section 4.1, the variable t is always supposed to be universally quantified in such a context. This shorthand is due to Hoare [Hoa85].

As mentioned before, the process **stop** is completely inactive, and so the only possible observation of its behavior is the empty trace

S 4.1 $\mathbf{stop} \models (t = \langle\rangle)$

The **exit** process performs the successful termination action δ and then transforms into **stop**

S 4.2 $\mathbf{exit} \models (t = \langle\rangle \vee t = \langle\delta\rangle)$

Every trace of the expression $(a;\ B)$ is either empty or has a as its head and its tail is a possible trace of B. Therefore, the trace specification of B must describe its tail

S 4.3 *If* $B \models S(t)$ *then* $(a;\ B) \models (t = \langle\rangle \vee (t = \langle a.t'\rangle \wedge S(t')))$

All rules below assume that the behavior expressions involved in the following properties do not diverge.

The trace specification of B must describe every trace of $(\mathbf{i};\ B)$, since the **i**-action is not observable

S 4.4 *If* $B \models S$ *then* $(\mathbf{i};\ B) \models S$

The proof rule for hiding is also restricted to the case where the behavior expression B does not diverge immediately on hiding actions in a sequence A

S 4.5 *If* $B \models (\neg diverge(\{B\}, \{A\}) \wedge S(t))$
then $(\mathbf{hide}\ A\ \mathbf{in}\ B) \models (\exists u \cdot t = u \lfloor (\alpha(B) - \{A\}) \wedge S(u))$

Every trace of a process instantiation is the result of renaming a trace described by the trace specification of the behavior expression defining that process

S 4.6 *Let* "**process** $P\ [a'_1, \ldots, a'_n] := B_p$ **endproc**" *be a process definition*
If $B_p \models S(t)$ *then* $P\ [a_1, \ldots, a_n] \models (\exists u \in traces(B_p) \cdot t = u[a_1/a'_1, \ldots, a_n/a'_n] \wedge S(u))$

Any observation of the behavior $(B_1[]B_2)$ must be a possible observation of either B_1 or B_2, and so it must be described by at least one of their trace specifications

S 4.7 *If* $B_1 \models R$ *and* $B_2 \models S$ *then* $(B_1[]B_2) \models (R \vee S)$

The following describes the proof rules associated with the parallel composition sequential composition, and disruption. This description is based on the fact that every trace of such compound processes can be described as equations between the traces of their components

S 4.8 *If* $B_1 \models R(t)$ *and* $B_2 \models S(t)$ *then*
$(B_1|[A]|B_2) \models (\exists u, v \cdot t \in u|\{A\}|v \wedge R(u) \wedge S(v))$
$(B_1 \gg B_2) \models (\exists u, v \cdot t = u \gg v \wedge R(u) \wedge S(v))$
$(B_1[> B_2) \models (\exists u, v \cdot t \in u[> v \wedge R(u) \wedge S(v))$

Next, we give the rules governing the / and $\uparrow$ operators on processes. Every trace specification of a behavior expression B describes the trace $s \frown t$, whenever t is a possible trace of (B/s) and s a possible trace of B, since $s \frown t$ is also a trace of B

S 4.9 *If* $B \models S(t)$ *then* $\forall B' \in B/s \cdot (B' \models S(s \frown t))$

If n is a natural number, every trace of $(B \uparrow n)$ is also a trace of B, and therefore must be described by any trace specification which B satisfies

S 4.10 $B \models S(t) \equiv \forall n \geq 0 \cdot (B \uparrow n) \models S(t) \wedge |t| \leq n$

This is because the set of all traces of B is prefix-closed, thus the left-hand-side asserts a property of all such traces.

4.3 An example

Consider the behavior of an unbounded reliable buffer, that will be referred to as UB, which has two channels, one input and one output. It is bound to output (eventually) whatever was input, and the number of inputs performed is unbounded. It can be described in LOTOS using a one slot buffer SB

```
process UB[input, output] :=
        hide mid in
             (input; (UB[input, mid]
                      |[mid]|
                      SB[output, mid]))
endproc
```

where the one slot simplex buffer (SB) is defined as follows

process $SB\ [input, output] :=$
$\quad input; output; SB\ [input, output]$
endproc

We would like to prove that $UB[input, output] \models t \downarrow output \leq t \downarrow input$.

We first prove that the one slot buffer SB satisfies its behavior requirements (i.e. the number of input items does not exceed the number of output items by more than one).

Lemma $SB[input, output] \models (0 \leq t \downarrow input - t \downarrow output \leq 1)$

Proof. By induction on the length of t using S 4.3, S 4.6, and S 4.10. □

Coming now to the main theorem, by S 4.10 it suffices to prove that

$$\forall n \geq 0 \cdot UB[input, output]\uparrow n \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n$$

and since UB does not diverge, we can achieve this by induction on n. Let

$$A = \{input,\ output\}$$
$$B = UB[input, mid]\ |[mid]|\ SB[output, mid]$$
$$B' = UB[input, mid]\uparrow n\ |[mid]|\ SB[output, mid]\uparrow n$$

Proof.
Base case. Obvious.
Induction hypothesis.
$\quad UB[input, output]\uparrow n \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n$
Induction step.
First Note that by S 4.6 and lemma above, since the renaming is one-to-one, we have
$\quad SB[output, mid] \models 0 \leq (t \downarrow output - t \downarrow mid) \leq 1$
By S 4.8-1 it follows that
$\quad B' \models \exists u, v \cdot t \in u|\{mid\}|v \wedge 0 \leq u \downarrow mid \leq u \downarrow input$
$\qquad \wedge 0 \leq (v \downarrow output - v \downarrow mid) \leq 1 \wedge |u| \leq n \wedge |v| \leq n$
$\Rightarrow$ [P 2.27, P 2.28, P 2.30, P 2.36, and P 3.1]
$\quad B' \models \exists u, v \cdot t \in u|\{mid\}|v$
$\qquad \wedge 0 \leq (u \downarrow mid - v \downarrow mid + v \downarrow output) \leq (u \downarrow input + 1)$
$\qquad \wedge t \downarrow A = (u \downarrow A + v \downarrow A) \wedge (t \downarrow mid = u \downarrow mid = v \downarrow mid) \wedge |t| \leq (2 \times n)$
$\Rightarrow$ $[u \downarrow output = v \downarrow input = 0]$
$\quad B' \models 0 \leq t \downarrow output \leq (t \downarrow input + 1) \wedge |t| \leq n$
Note that by the definitions of merging, $\uparrow$, and T 3.7 we can show that
$\quad traces(B \uparrow n) \subseteq traces(B')$
Therefore, it follows that
$\quad B \uparrow n \models t \downarrow output \leq (t \downarrow input + 1) \wedge |t| \leq n$
$\Rightarrow$ [S 4.3]
$\quad (input;\ B \uparrow n) \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n + 1$
$\Rightarrow$ [S 4.5, since clearly $traces(B) \cap \{mid\}^* = \{\langle\rangle\}$]
$\quad$ **hide** mid **in** $(input;\ B \uparrow n) \models \exists u \cdot t = u \lfloor A \wedge u \downarrow output \leq u \downarrow input \wedge |u| \leq n + 1$
$\Rightarrow$ $[(u \lfloor A) \downarrow a = u \downarrow a$ if $a \in A$, and $(u \lfloor A) \downarrow a = 0$ if $a \notin A]$
$\quad$ **hide** mid **in** $(input;\ B \uparrow n) \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n + 1$
$\Rightarrow$ [S 4.10, and definition of $\uparrow$]
$\quad$ (**hide** mid **in** $(input;\ B)) \uparrow n + 1 \models t \downarrow output \leq t \downarrow input \wedge |t| \leq n + 1$ □

One should note that the unbounded buffer UB, described above, and an unbounded unreliable buffer UB' , that may lose inputs, satisfy the same trace specification, since their traces t also enjoy the property $t \downarrow output \leq t \downarrow input$. The LOTOS description of UB' is similar to that of UB except that we should replace the instantiation of SB by an instantiation of SB', given by

process SB' [$output$, $input$] :=
$output$; $input$; SB' [$output$, $input$]
[]
i; $input$; SB' [$output$, $input$]
endproc

The proof that UB' satisfies the above property can be achieved by proving, by induction, that $SB'[output, input] \models t \downarrow output \leq t \downarrow input + 1$. However, UB and UB' cannot be regarded as equivalent, even though we cannot distinguish their trace sets. This shows that in general, trace specifications cannot model all observable aspects of a process, and therefore their theory needs to be enriched, in such a way that $\models$ relation can provide means to distinguish such processes. Such means are introduced in [BHR84] (i.e. refusals and failures) and applied to LOTOS in [Gal89, GLO91].

5 CONCLUSION

We have developed the elements of a trace theory for "basic LOTOS" processes, and we have shown how this theory can be used to prove an invariant property of a simple LOTOS process. Although our proof method is more complex than bisimulation, the latter would not have been able to prove directly the desired property.

Based on this theory, we have implemented a "trace assertion checker" for basic LOTOS. Invariant properties of traces can be specified in an appropriate language, and then the set of traces for a basic LOTOS process is developed, at the same time checking whether the properties are still true up to the a given maximum trace length.

Work towards a similar tool designed for CSP was presented in [Kou87].

Another approach to the same problem could be developed using temporal logic. In [FGL90] a formal relation between LOTOS and linear temporal logic has been studied. Opposite to the temporal approach we treat infinite computations in terms of infinite sets of prefixes of traces.

Note that we can only treat safety properties in our model. Liveness properties can be treated using a failure-based model [GLO91].

Acknowledgment

This work was supported by the Natural Science Research Council of Canada, The Telecommunication Research Institute of Ontario, the Tunisian government, and the Canadian International Development Agency. The "trace assertion checker" was implemented by Brahim Ghribi.

References

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential. *JACM*, 31(3):560–599, 1984.

[Bri88] Ed Brinksma. *On the Design of Extended LOTOS*. PhD thesis, University of Twente, The Netherlands, 1988.

[FGL90] A. Fantechi, S. Gnesi, and C. Laneve. An expressive temporal logic for lotos. In *Formal Description Techniques*. North-Holland, 1990. FORTE 89.

[Gal89] Souheil Gallouzi. Trace analysis of LOTOS behaviours. Master's thesis, University of Ottawa, Canada, 1989.

[GLO91] S. Gallouzi, L. Logrippo, and A. Obaid. A Hoare style proof system for LOTOS. In J. Quemada, J. Manas, and E. Vazquez, editors, *Formal Description Techniques*. North-Holland, 1991. FORTE 90.

[HO83] Brent T. Hailpern and Susan S. Owicki. Modular verification of computer communication protocols. *IEEE Transactions on Communications*, COM-31(1):56–68, 1983.

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

[ISO88] ISO, IS 8807. *Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour*, May 1988.

[Kou87] D.G. Kourie. The design and use of a prolog trace generator for CSP. *Software-Practice and Experience*, 17:423–438, 1987.

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

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

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference*. LNCS volume 104, Springer-Verlag, 1981.

[vdS85] Jan L.A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Sringer-Verlag, 1985.