IST. EL. ...
BIB...
Posiz. ARCHIVIO
B4-40

A TUTORIAL ON LOTOS

Tommaso Bolognesi - CNUCE / C.N.R. - Pisa

Rocco De Nicola - I.E.I. / C.N.R. - Pisa

Rapporto interno C86-7

Consiglio Nazionale delle Ricerche

CNUCE

PISA

A TUTORIAL ON LOTOS

Tommaso Bolognesi
CNUCE – C.N.R. – Pisa

Rocco De Nicola
I.E.I. – C.N.R. – Pisa

Riassunto. Viene illustrato LOTOS, un linguaggio per la specifica formale di sistemi distribuiti e concorrenti. Un sistema viene visto come un insieme di processi che interagiscono fra loro, si scambiano dati ed offrono potenziali interazioni con l'ambiente. LOTOS permette la descrizione modulare della struttura interna, anche dinamicamente variabile, di un sistema, sebbene la semantica di una specifica si riferisca soltanto al comportamento temporale del sistema specificato cosi' come lo si puo' osservare dall'ambiente esterno. E' previsto che LOTOS diventi uno Standard Internazionale ISO per il 1988.

Abstract. We illustrate LOTOS, a language for the formal specification of distributed, concurrent systems. A system is seen as a set of processes which interact with each other, exchange data, and offer potential interactions with the environment. LOTOS supports the modular description of the internal, dynamically variable structure of a system, although the semantics of a specification only refers to the temporal behaviour of the specified system as observable from the external environment. LOTOS is expected to become an ISO International Standard by 1988.

Key Words and Phrases: concurrent languages, formal description techniques, protocol specification, specification languages.

0. Introduction

LOTOS (Language of Temporal Ordering Specification) is one of the two [12, 13] specification languages under development for standardization within International Standards Organization (ISO). The main motivation for starting this effort was the need of *formally* describing the OSI (Open Systems Interconnection) network architecture. The aim of formal specifications in general is twofold:

i) to provide unambiguous, clear and, possibly, concise descriptions of the system being designed, and

ii) to support rigorous analysis and verification of the design *before* its implementation.

These aspects are particularly important of a distributed, standard architecture, such as OSI. Machines must communicate and cooperate with each other, and ambiguous specifications of the related software could lead to incompatible implementations. And OSI is having a large diffusion: any error which slips out of the analysis in the early stages of the development cycle is going to proliferate in all implementations.

The OSI architecture is logically partitioned into seven functional layers. By making use of the functionality provided by layer N-1, or **(N-1)-service**, the **entities** at layer i cooperate with each other according to the **N-protocol** to provide an enhanced **N-service**, with more powerful primitives, to layer N+1. At the bottom of the layered architecture sits the physical communication medium. A general description of the concepts of OSI can be found in [14] and [24]; see [25] for a quick overview. The standardization process of the OSI architecture proceeds layer by layer, but the services and protocols standardized so far are described in a mixture of natural language, graphical representations, ASN.1 [11], and state tables. The tendency within ISO today is to complement these semi-formal descriptions with formalized ones. Work in this direction has already started. LOTOS itself is expected to become ISO international standard by 1988.

The notion of N-service allows one to **abstract** from the structure of layers 0 through N and from the flow of data between them. The description of an N-service is, in principle, an **extensional** description of the **observable behaviour** of a black box, in terms of feasible sequences of service primitives exchanged at its so called 'service access points'. An N-protocol, conversely, gives an **intensional** description of the layer, in terms of N-entities and (N-1)-service. Protocol entities **synchronize** and **communicate** with each other, and operate **concurrently** [24].

The **expressive power** of a language, in general, is the ability of its constructs to capture in a natural way the relevant aspects of the objects being specified; and indeed all the aspects of OSI mentioned above can be naturally expressed in LOTOS. The language is meant to allow abstract specifications, which are independent from implementation details. It allows both intensional and extensional descriptions. It allows, for instance, to hide internal events, making them unobservable from outside. It supports hierarchic and modular design. And, of course, it provides constructs for expressing synchronization, communication and concurrency. Needless to say, such features suit the needs of formal specification well beyond the scope of OSI.

The **analytical power** of a language is determined by the possibility to formally verify properties of the specified objects. In this respect, the existence of a **formal semantics** for the language is crucial. It guarantees that the meaning of a LOTOS expression does not depend on subjective interpretations, but is a formal object uniquely associated to it. Not only does this avoid misinterpretations, typically by implementors, but it also provides a solid basis for automated analysis of a specification, and for proving that two syntactically different expressions have the same meaning. Formally specified requirements and implementations, for instance, can be proved **equivalent** in a rigorously defined sense. Or, an N-protocol composed with an (N-1)-service can be shown to behave as required by the N-service.

Unfortunately, expressive power and analytical power are often conflicting needs: the more expressive a language, the more difficult the analysis of its programs. It is too early to evaluate LOTOS with respect to this tradeoff. While some fairly complete specifications of OSI services and protocols have already appeared [5, 22], which allow to appreciate the expressive power of the language, the practical analyzability of such specifications via automated tools is still to be assessed, and a number of national and international projects are devoted to this challenging task (e.g., the ESPRIT/SEDOS project of the European Community).

LOTOS has two components. **Processes** are defined and combined using an extension of the Calculus of Communicating Systems, **CCS**, developed at the University of Edinburgh mainly by R. Milner [10,16], although some elements have been borrowed from the Theory of Communicating Sequential Processes (CSP) [2] and CIRCAL [15]. CCS provides a small set of operators and an algebraic framework for describing and analyzing concurrent systems. **Abstract Data types** can be defined in **ACT-ONE**, an algebraic specification language developed by the ACT-group of the Technical University of Berlin [6, 7]. Historically, the two basic components of LOTOS have been dealt with independently from each other, and their presentations can be easily separated. In fact this paper concentrates on the process (or control) component of the language, for various reasons. First, this is the component which has received more attention, so far, within ISO.

Second, this component departs from its inspiring theories more than the other one. Third, the theory of abstract data types is today more settled and less debated than the various approaches to a theory of concurrency. The reader interested in the specification of abstract data types in general may refer to [8] and [9]. Another tutorial presentation of LOTOS is found in [26].

The paper is organized as follows. Section 1 is meant to informally introduce the basic concepts of LOTOS, essentially composition, synchronization and abstraction of processes, in order to provide an intuitive support for their formal treatment in the rest of the paper. Our first approach to this goal is musical: we hope it could 'sound' better or, at least, less tedious than the usual intuitive modelization in terms of black boxes emitting characters. An actual, simple LOTOS specification is presented in Section 2. It illustrates how the language can support modular design and the use of abstraction levels. It also contains the specification of an elementary data type.

Section 3 provides syntax and semantics of LOTOS behaviour expressions. For every expression we give the axioms and the inference rules of the operational semantics, which allow to precisely derive the potential behaviours of the process denoted by the expression. Simple examples are given both of the expressions and of the derivation of their behaviours.

Section 4 introduces the notion of equivalence between LOTOS processes (or expressions). This notion, together with the operational semantics, is essential for giving a precise meaning to propositions such as "these two syntactically different expressions have the same meaning". Notions of equivalence are needed for proving that a given formal specification is met by some formally specified implementation, and for safely substituting subparts of a system without affecting its overall behaviour. Section 5 contains some concluding remarks on possible future developments and improvements of the language.

1. Composing processes

1.1 Music

The first mechanical device for composing nondeterministic music is probably the "arca musarithmica" shown in Figure 1.1, described by jesuit father Athanasius Kircher in his book "Musurgia Universalis" (Rome, 1660) [20]. The "score" appearing on the upper side of the box was obtained by randomly rotating internal elements of the device. In the second half of the 18th

century several books for nondeterministic composition were published in Europe, some of them related to illustrious names such as Mozart and Haydn. All these manuals were based on a method of casting dice to impose a random temporal ordering on a set of short musical episodes pre-composed by the author.

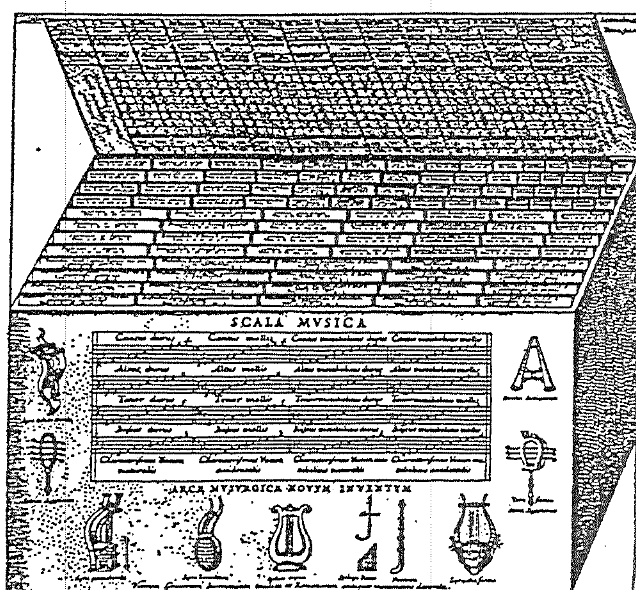


Figure 1.2 - A device for nondeterministic sequential composition

The basic idea was **sequential composition**: building blocks were composed together only on a one-after-the-other basis. However, the recent discovery of a correspondence between Maximilian Stadler and a minor musical editor, dating the late 18th century, seems to reveal that an attempt was made also in the direction of **concurrent and synchronized composition**, where separate sequential streams of building blocks overlap in time and, hopefully, synchronize with each other from time to time. Unfortunately, no other documentation supports this fascinating thesis, and the authenticity of these few letters is doubtful. The story, as inferred from the correspondence, involves Stadler himself (Max), Franz J. Haydn, Wolfgang A. Mozart, and is summarized below.

In obscure circumstances, Max comes into possession of a book (*process Max*). Every page (*state*) contains a few musical episodes (*actions*) . Every episode is labelled with one out of four possible labels : "Adagio", "Largo", "Larghetto", "Presto" :

process Max [Adagio, Largo, Larghetto, Presto].

Also, every episode terminates with the number of a new page to go to (*state transition*). Max has

written similar manuals and seems to know how to use it. He starts at page 1 (*initial state*), which offers the choice between an "Adagio" and a "Presto", randomly chooses one of them, plays it at the harpsichord, goes to the page indicated at the end of the episode, and repeats the procedure over and over again. Finding himself at page 1 for the second time (*recursion*), Max realizes that the music might go on forever. The discovery of a white page, with no episode at all (*process 'stop', offering no action*) tells Max that the music may terminate.

One kind of musical episode puzzles Max. It has no label, and its score is empty, i.e., no note whatsoever appears on the pentagram; yet, at the end, the indication of a new page is regularly given. He then visits his friend Franz for advice. Strangely enough, Franz owns a similar book, also found in mysterious circumstances, which includes "Largo", "Larghetto", "Presto" and "Rondo'" episodes, plus several empty ones. With excitement they go to the harpsichord and play for each other in turn, each performing a few episodes from his own book, then leaving the keyboard to the other. They never play simultaneously (*pure interleaving, or concurrent composition with no synchronization*) :

Max [Adagio, Largo, Larghetto, Presto]
|||
Franz [Largo, Larghetto, Presto, Rondo']

Also Franz is puzzled by the anomalous empty episodes, and suggests to call them "silent episodes" (*silent, or unobservable action*): when one is chosen the player should simply pause for a while and then go to the page indicated at its end. Max objects that silent episodes are pointless; Franz replies that they do have an effect: the set of alternatives (episodes) that can be accessed *before* (or, rather, *instead of*) a silent one, is different from the set accessible *after* its "performance", because of the change of page involved. Max is not convinced, but temporarily agrees to treat those episodes as pauses.

By accident, the two musicians realize that episodes with equal labels can be nicely played simultaneously, four-hands (*interaction*). Thus, they decide to always synchronize on the common episodes, i.e. "Largo", "Larghetto" and "Presto" (*concurrent composition with the maximum of synchronization*):

Max [Adagio, Largo, Larghetto, Presto]
||
Franz [Largo, Larghetto, Presto, Rondo']

This implies that when Max reaches a page offering, among others, a "Largo", and he wishes to play that episode, he will not do so until Franz has found a "Largo" too, and is willing to play four-hands. The same holds for Franz.

Unfortunately Max has problems in keeping synchronized with Franz, who is definitely more talented in playing fast. They soon agree to give up with any attempt to synchronize on "Presto" episodes, that is, to treat "Presto's" in the two books as though they were differently labelled. Synchronization must occur only on "Largo's" and "Larghetto's" (*concurrent composition*):

Max [Adagio, Largo, Larghetto, Presto]
|[Largo, Larghetto]|
Franz [Largo, Larghetto, Presto, Rondo']

Max and Franz are content with their achievements. Still they feel they do not have a good theory about silent episodes, and decide to visit their friend Amadeus, the genius, to play for him and ask for advice. Amadeus listens carefully to their music, then asks for the two books, gives his guests leave, and receives them back few days later, with a third book in his hands.

"This book" Amadeus explains "contains all the music that the two of you played, or could have played to me". Then he sits at the keyboard and plays, using the new book as Max was doing with his book before visiting Franz. Although the interleaving of "Adagio","Presto" and "Rondo'" episodes sounds the same as for the Max-Franz music, Max objects that, obviously, Amadeus alone cannot play four-hands "Largo's" and "Larghetto's". Without interrupting his playing, Amadeus admits that, in compiling the book, he has "hidden" those passages, by *turning them into silent episodes*: they can no longer be distinguished from the other silent episodes deriving from the two original books. "However" he clarifies "four-hands means synchronization, and synchronization means constraints on the ways in which your sequences, Max, can be interleaved with the ones played by Franz. These constraints are perfectly reflected in my book; just the score of your four-hands episodes has been hidden, and no further musician will be able to synchronize with you on those episodes" (*hiding*):

(
Max[Adagio, Largo, Larghetto, Presto]
| [Largo, Larghetto] |
Franz[Largo, Larghetto, Presto, Rondo']
)\ [Largo, Larghetto]

Max is lost. Franz is only half-convinced, and one thing now puzzles him. If some of the silent episodes in the new book hide four hands episodes played by Max with Franz himself, what is hidden behind the silent episodes of their own books?

1.2 LOTOS

A LOTOS **process** is an abstract entity able to perform **actions**, either **observable** or

unobservable, and to **interact** with other processes in its environment. An interaction may occur between two or more processes whenever every one of them is ready to perform the same *observable* action. The interaction can be seen as a unique action identical to the component actions, performed by the process composed by the interacting processes. An observable action performed by a (possibly composite) process p can be considered as a potential interaction with other processes, i.e., as the manifestation of p's ability to interact with any environment offering that same action. In music: Max and Franz may interact when they are both ready to play, say, a "Largo". In playing a "Presto" alone, Franz expresses his desire to interact with some other "presto" player (Max excluded, as they agreed). Finally a four-hands "Largo" is as good as a two-hands one for further "Largo" players to synchronize with it.

Consider this 'process abstraction':

```
Max3 [in1, in2, in3, out] : noexit :=
    (Max2[in1, in2, mid]
    ||
    Max2[mid, in3, out]
    ) \ [mid]
```

It defines a process called Max3 in terms of an expression similar to the ones used in Max's story. Process Max3 is defined as the concurrent composition (||) of two instances of process Max2 (incidentally, 'Max' stands now for Maximum, rather than Maximilian). The first (second) instance of Max2 is capable of performing actions named **in1, in2, mid** (**mid, in3, out**). The two streams of actions may interleave in all ways that are compatible with their synchronizing on action mid, the only action shared by the two processes. Such synchronizations would appear as mid-actions performed by process Max3, if the hiding operator \ **[mid]** were not used. In fact, because of the hiding of mid-actions, these interactions will appear as silent, unobservable, internal actions of Max3, denoted **i**. Hence, only in1-, in2-, in3- and out-actions are observable from the environment of Max3, and are available for further interactions. This is reflected by the parenthesized list of gates in the first line of the process abstraction above. '**:** noexit' means that process Max3 cannot pass control to other processes on its completion.

As interactions originate from the execution of equal observable actions by a set of processes, the name of an observable action can be thought of as the name of an **interaction point**, or gate, shared by the processes, where interaction takes place. Although not essential, the spatial notion of gate has the practical advantage of allowing pictorial representations such as that in Figure 1.2, which refers to the process abstraction above, and reads as follows. The behaviour of process Max3 consists of sequences of actions performed at its external gates **in1, in2, in3, out**. The alternative terminology by which the process *offers* actions at its external gates stresses the interpretation of an action as a potential interaction with the environment. Looking inside the box

representing process Max3 (i.e., reading beyond the ':=' symbol in the process abstraction), the process reveals two component processes, sharing only gate **mid**. However, mid-actions are hidden, thus gate **mid** is internalized and not accessible from the outside of the outer box.

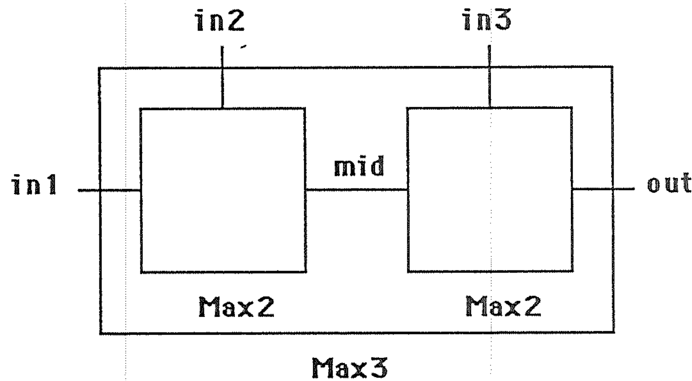


Figure 1.2 - Spatial representation of process Max3

Concurrent composition of processes is a central notion in LOTOS. Of course this is not the only way of combining LOTOS processes together, or defining new processes in terms of existing ones. Some of these new combinators (e.g. the choice operator '[]') appear in the small specification below, and the complete set of operators is given in Section 3.

2. A small specification

A typical LOTOS specification is syntactically structured as follows:

specification <specification identifier> *[*[<gate list>]*][*(<parameter list>)*]* : <functionality>
    :=
    <behaviour expression>
    where
        <type definitions>
        <process definitions>
endspec

We adopt these notational conventions: terminal symbols are boldface (they literally appear in an actual specification), reserved words of the language are underscored, nonterminal symbols are enclosed in '<>' brackets; for clarity, these brackets will denote nonterminals, i.e. syntactical categories of the language, throughout the paper. Finally, '*[...]* ' denotes optionality.

The <type definitions> sections of a specification describe the data sets being used and the operations that can be performed on them. This component of LOTOS essentially coincides with **ACT ONE** [7], the aforementioned language for the definition of abstract data types. The central component of a specification is the <behaviour expression>, which describes the dynamic behaviour of the specified system as a process, by implicitely defining all possible sequences of events which can be offered at the gates in <gate list> and are observable from the environment of the system. This <behaviour expression> is usually defined in terms of other processes which, in turn, may use new <type definitions>: these new types and processes are all defined after the where keyword.

We introduce now a small but complete LOTOS specification, and describe informally how it is read and what the described system is meant to achieve. We take the example also as a means for introducing some terminology. Comments are enclosed in (*...*) brackets.

```
1    Specification Max_of_three [in1, in2, in3, out]:noexit :=
```

(* Defines a 4-gate process that accepts three natural numbers at three input gates, in any temporal order, and then offers the largest of them at an output gate *)

```
2           (Max2[in1, in2, mid] || Max2[mid, in3, out]) \ [mid]
3    where

4           type natural is
5                  sorts  nat
6                  opns   zero:    --> nat
7                         succ:    nat --> nat
8                         largest: nat, nat --> nat
9                  eqns   largest(zero, x) = x
10                        largest(x, y) = largest(y, x)
11                        largest(succ(x), succ(y)) = succ(largest(x, y))
12          endtype (* natural *)

13          process Max2[a, b, c] : noexit :=
14                 a ?x:nat;  b ?y:nat;  c !largest(x,y); stop
15                 []
16                 b ?y:nat;  a ?x:nat;  c !largest(x,y); stop
17          endproc (*Max2*)

18   endspec (*Max_of_three*)
```

Max_of_three is a non-parametric specification, as no <parameter list> appears on line 1. Its defining <behaviour expression> is given in line 2. Lines 4 through 12 give the only <type definition>, while lines 13 through 17 give the only <process definition> of the specification.

Let us first give a quick glance at the definition of type **nat** (see [7, 8, 9] for a complete introduction to abstract data types and ACT ONE). A <type definition> in general is meant to define sets of values and operations on them, and consists of three sections called sorts (line 5), opns (for "operations", lines 6 through 8) and eqns (for "equations", lines 9 through 11). The first two sections together are called **signature** of the type. The signature of our type **natural** defines one <sort identifier> ('**nat**'), and one nullary ('**zero**'), one unary ('**succ**'), one binary ('**largest**') <operation identifier>'s. By combining <operation identifier>'s we construct <value expressions> of sort **nat** such as:

a) **zero**
b) **succ(zero)**
c) **largest(zero, succ(zero))**
d) **largest(succ(zero), zero)**

We will say that two <value expression>'s denote the same value if and only if it is possible to rewrite them, according to the equations in section eqns, until they become syntactically identical. By using the two equations on lines 9 and 10 we can easily prove that <value expression>'s (b), (c) and (d) above denote the same value. Notice that we are checking that two expressions denote the *same value* without asking *what* the value is. The answer to this question would require an introduction to the initial algebra semantics of abstract data types, which we do not provide here. We will simply assume the existence of a function *val* which maps <value expression>'s of some sort t (e.g. 'nat') onto a set of **values** of sort t (e.g. {0, 1, 2...}), which will be called **domain** of sort t, and denoted **Domain(t)**. A concluding remark: basic data type definitions such as the one above would typically be found in a standard library, so that specifications could refer to them by name via a 'library' construct (e.g. " library **natural** endlib").

Let us now consider the process part of the specification. We find convenient, for our future discussions, to give a name to the <behaviour expression> of line 2. Thus we easily rephrase the specification to make it appear as a <process definition>. This is done in Figure 2.1, where some fundamental syntactical categories of LOTOS are identified too. We are specifying a process called **Max3** which interacts with its environment through gates **in1**, **in2**, **in3** and **out**, and cannot enable any subsequent activity (**noexit**). The behaviour of this process is defined, in a process abstraction, by a <behaviour expression> which refers to process **Max2**. We have already met (see Figure 1.2) and commented this process abstraction . Let us now turn to the definition of Max2. The synchronization and communication capabilities of this process are expressed in terms of the *formal* gates **a**, **b**, **c** (*actual* gates are used in the process instantiations **Max2[in1, in2, mid]** and **Max2[mid, in3, out]**). Process **Max2** offers the choice ([]) between two alternative streams of

actions, which only differ by the order in which the actions at gates **a** and **b** occur. Let us simply say, for the moment, that symbols '?' and '!' following a gate name denote, respectively, the input and output of some value at that gate. Thus, both alternatives input two values, associated to variables **x** and **y**, and output the largest of them at gate **c**. Notice that <value expression>'s may contain <value identifier>'s (variables) of some sort. For example, largest(x, y) contains variables **x** and **y** of sort **nat**. In this case *val*(largest(x, y)) is a function of the values *val*(x) and *val*(y) which the variables are bound to by their *binding occurrences* in '?x:nat' and '?y:nat'.

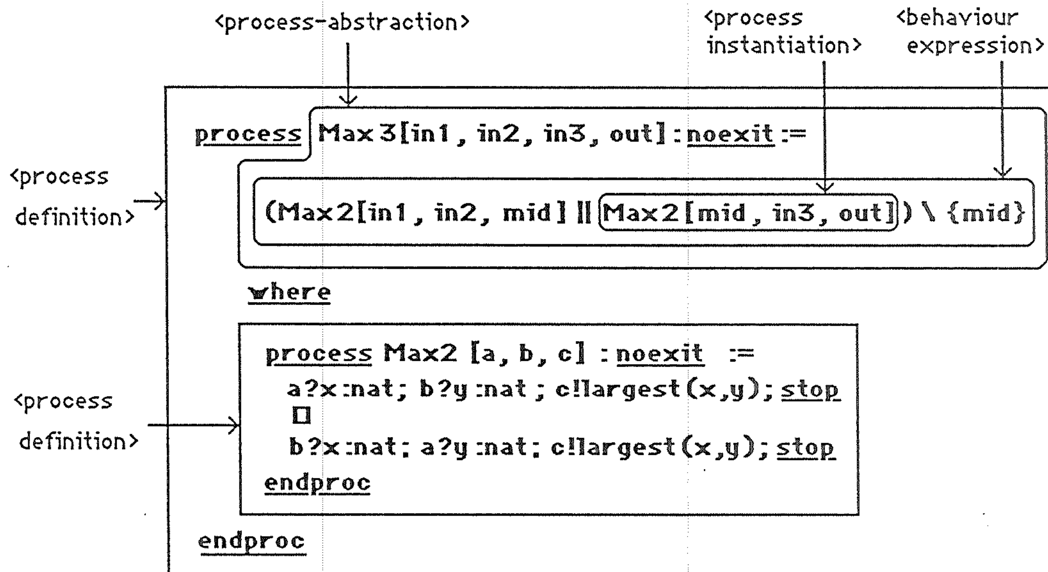


Figure 2.1 - Process Max3

Despite the superficiality of our first reading, the behaviour of the specified system should now be clear: it accepts three natural numbers at gates in1, in2, in3, in any temporal order, and then offers the largest of them at gate out. Note how the structure of the specification reflects the way it was obtained by "top-down" refinement.

We are ready for a more formal and exhaustive attack to the syntax and semantics of LOTOS <behaviour expression>'s.

3. Syntax and semantics

Every LOTOS process is defined in terms of a <behaviour expression>. Therefore the latter is the fundamental syntactical construct that we need to concentrate upon. Syntactically, a <behaviour expression> is obtained by applying a nullary, unary or binary **operator** respectively to zero, one

or two other <behaviour expression>'s. Since a <behaviour expression> defines the behaviour of a process, and the behaviour of a process is made up with transitions (i.e. actions and related process transformations), we would like to have a formal mechanism to derive all possible transitions of a process from the syntactic structure of its defining <behaviour expression>. This is exactly what the structured **operational semantics** approach provides [19]. By applying the **axioms** and **inference rules** of the operational semantics, we can infer the potential transitions of a compound <behaviour expression> from the potential transitions of its components (notice that we conveniently confuse the terms "process" and "behaviour expression").

We need now to define actions and transitions precisely. We will let:

G denote the set of user-definable gates;

GV be the set of observable actions $\{g<v_1, ..., v_n> \mid g \in G$, v_i a value of some sort t_i $(i=1...n), n \geq 1\}$;

DV be the set of observable, successful termination actions $\{d<v_1,...,v_n> \mid d$ is a distinguished gate, not in G, called "successful termination"; v_i is a value of some sort t_i $(i=1...n), n \geq 0\}$;

An **action** is an element of set $GV \cup DV \cup \{i\}$. In particular:

- $g<v1,...,vn> \in GV$ is the **observable action** which offers the n-tuple of values $<v_1, ...,v_n>$ at gate g;
- $d<v1,...,vn> \in DV$ is the **successful termination** action, which also offers the (possibly empty) n-tuple of values $<v_1, ...,v_n>$ at special gate d;
- i is the **unobservable action**.

We will use the following mnemonic variables for actions:

'o'	ranges over GV	('o' for 'observable')
'od'	ranges over $GV \cup DV$	('d' for 'done')
'oi'	ranges over $GV \cup \{i\}$	('i' for 'internal')
'odi'	ranges over $GV \cup DV \cup \{i\}$	
'di'	ranges over $DV \cup \{i\}$	

For example, 'od' could stand for 'g<3, TRUE, 'tree'>', or 'd<5, FALSE>'; 'odi' could stand for

'i', and so on. We will need a function which indicates the gate where an action occurs:

gate(odi)	= undef	if odi = i
	= g	if odi = $g<v_1 \ldots v_n>$
	= d	if odi = $d<v_1 \ldots v_n>$,

and also function ***gates***(B), which, informally, indicates the set of all the gates where B may perform an action.

Please notice that, conceptually, an <action denotation>, such as "**a** !3" and an action, such as "a<3>" are different objects, of pertinence, respectively, to the syntax and to the semantics of the language. The former is a building block of a specification, and denotes the latter, which is a building block of the system behaviour.

Let B denote the set of <behaviour expression>'s. The idea of an action as a state transition, or transformation of a process into another process, is captured by the following definition. A **labelled transition** relation is a binary relation over B, denoted '−odi→', where label 'odi' is an action (we will often omit the attribute "labelled" in the sequel). The fact that <behaviour expression>'s B1 and B2 are related via action 'odi' is usually represented with the infix notation:

$$B1 \xrightarrow{odi} B2,$$

which reads: "B1 is able to perform action 'odi' and transform into B2".

We may say, in conclusion, that the purpose of the operational semantics of LOTOS is the formal definition of transition relations over the set of <behaviour expression>'s. Despite the tutorial nature of this paper, we found appropriate to present the semantics in a formal way, because, in this case, the formalism directly and naturally reflects our intuitive understanding of the meaning of expressions; and the little cost of explaining how to read axioms and inference rules is more than compensated by the advantages in terms of clarity and conciseness.

We give now a complete syntax table of <behaviour expression>'s , for quick reference, and then present them one by one, with their semantics.

3.0 Table of behaviour expressions

The complete list of LOTOS <behaviour-expression>'s is given in Table 1 below, which includes all LOTOS operators. Again, '<...>' brackets denote nonterminals and '*[...]*' brackets denote optionality. Also, {...}* denotes zero or more occurrences of '...'. Symbols 'B', 'B1', 'B2' in the table stand for any <behaviour-expression>. Any <behaviour-expression> must match one of the forms listed in column SYNTAX. We have taken the metalinguistic liberty of representing some lists with dots.

Table 1 - Syntax of <behaviour expression>'s

<table>
<tr><th>NAME</th><th>SYNTAX</th></tr>
<tr><td>inaction</td><td><u>stop</u></td></tr>
<tr><td>action prefix</td><td></td></tr>
<tr><td>- unobservable (internal)</td><td><u>i</u> ; B</td></tr>
<tr><td>- observable</td><td>g {<value-offer>}*; B</td></tr>
<tr><td>choice</td><td>B1 [] B2</td></tr>
<tr><td>boolean guarding</td><td>[<boolean-expression>] → B</td></tr>
<tr><td>parallel (concurrent) composition</td><td></td></tr>
<tr><td>- general case</td><td>B1 |[g_1, ... , g_n]| B2</td></tr>
<tr><td>- pure interleaving</td><td>B1 ||| B2</td></tr>
<tr><td>- with the maximum of synchronization</td><td>B1 || B2</td></tr>
<tr><td>hiding</td><td>B \ [g_1, ... , g_n]</td></tr>
<tr><td>successful termination</td><td><u>exit</u> [(E_1, ... ,E_n)]</td></tr>
<tr><td>enabling (sequential composition)</td><td>B1 >> [<u>def</u> x_1:t_1, ... , x_n:t_n <u>in</u>] B2</td></tr>
<tr><td>disabling</td><td>B1 [> B2</td></tr>
<tr><td>process instantiation</td><td>p [[g_1, ... ,g_n]] [(E_1, ... , E_m)]</td></tr>
<tr><td>local definition</td><td><u>let</u> x_1=E_1, ... ,x_n=E_n <u>in</u> B</td></tr>
<tr><td>summation - on values</td><td><u>for</u> x:t <u>choice</u> B</td></tr>
<tr><td>- on gates</td><td><u>for</u> g <u>in</u> [g_1, ... , g_n] <u>choice</u> B</td></tr>
</table>

Operator priorities are given in the following list, in decreasing order:

hiding
> action prefix
> boolean guarding
> choice
> parallel composition
> enabling
> disabling

> local definition and summation.

3.1 Inaction: stop

This is one of the basic <behaviour expression>'s. No axiom or inference rule is associated with it and it is thus impossible to derive transitions from/for it. Hence we understand stop as denoting a predefined LOTOS process which is unable to perform any action, and is uncapable of interacting with any other process. A typical example of a process that does not interact with the environment is the following vending machine:

```
broken_machine := stop
```

3.2 Action prefix - unobservable: i; B

We may understand the action denotation 'i;' as a prefix operator applied to <behaviour expression> B. The resulting behaviour expression> (process) is only able to perform the unobservable action 'i' (the silent episode that puzzles Max and Franz in Section 1.1), and transform into B. Indeed the unique axiom for this construct is:

=========

i;B –i→ B

=========

The unobservable action models an event internal to the process, which no external process can synchronize with (i.e., be directly aware of). Example:

```
powercut_machine := i; stop
```

Here i models a powercut. Notice that there is no way a user can distinguish between this vending machine and the previous one: this is a first, trivial example of the concept of *observational equivalence* to be formally introduced later.

3.3 Action prefix - observable

For convenience of presentation we distinguish three cases, the last one being the general case.

• *Action prefix with multiple value offer:* **g ?x:t; B**

'?x:t' is a <multiple value offer>, where **x** is a variable declared of sort **t**. This process is able to offer any value of sort **t** at gate **g**. The associated axiom is:

==

g?x:t; B(x) –g<v>→ B(v), for any value v of sort **t**

==

The occurrence of this action binds variable **x** to value v, so that the new process B, which represents the scope of this binding, may access the value, through the variable. Indeed, B(v) denotes <behaviour expression> B(x) after substituting v for all *free* occurrences of x in it. Let us clarify these concepts with an example. Consider the <action prefix expression> at line 14 of specification Max_of_three (Section 2):

a ?x:nat; b ?y:nat; c !largest(x,y); <u>**stop**</u>

where ' **a ?x:nat;**' is the <action prefix>, and B(x) = '**b ?y:nat; c !largest(x,y);** <u>**stop**</u>' is the <behaviour expression> to which the prefix applies, and which includes, in turn, an <action prefix>. The whole expression does not include free occurrences of **x**, nor of **y**, as the occurrences of the variables in '**largest(x,y)**' are *bound*, that is, they fall within the scopes of their binding occurrences in the two prefixes. However, if we consider '**b ?y:nat; c !largest(x,y);** <u>**stop**</u>' in isolation, then the occurrence of **x** in '**largest(x,y)**' is *free*, as no binding occurrence of **x** is there any more to bind a value to it. According to the axiom above, the whole <behaviour expression> is able to perform any one of the actions in the set {a<0>, a<1>, ..., a<n>}, i.e. to offer any natural number at gate **a**. A possible sequence of two transitions is:

a ?x:nat; b ?y:nat; c !largest(x,y); <u>**stop**</u> –a<0>→

b ?y:nat; c !largest(0,y); <u>**stop**</u> –b<3>→

c !largest(0,3); <u>**stop**</u>

Because of the binding of values 0 and 3 to variables **x** and **y**, which allows the new process to refer to these values, we could say that, rather than *offering* values, the process *offers to accept* values, and

think of the '?' symbol as indicating **input transitions.**

The usual rules for nested scopes apply. For instance, consider expression **'a?x:t; b?x:t; c!(x+1)'** : the value output at gate **c** will depend on the value input at gate **b**, not gate **a**.

Another example:

---

trap_machine[money]:= money?x:coin; <u>stop</u>

Unlike the two previous machines, this one is able to accept a coin; but then it stops.

• *Action prefix with single value offer:* **g !E;** B

'!E' is a <single value offer>, where **E** is a <value expression>. The associated axiom is:

==================

g!E; B –g<*val*(E)>→ B

==================

where *val*(E) denotes the value of expression **E**, as discussed in Section 2. As an example, we complete the sequence of two transitions derived above, with a third and terminal transition:

c !largest(0,3); <u>stop</u> –c<3>→ <u>stop</u>

No memory of the value offered is kept by the process offering this value, hence we think of the '!' symbol as indicating **output transitions.** The following machine:

---

simple_machine [money, box]:= money?x:coin; box!cookie; <u>stop</u>

---

accepts *any* coin at gate money, and offers a cookie at gate box.

• *Action prefix with structured value offer (general case)* g {<value offer>}*;B

This is the general form of the observable <action prefix expression>, where the gate name is followed by a list of zero or more <value offer>'s, every one of them being either a <multiple value

offer> or a <single value offer>. The associated axiom is essentially the combination of the two axioms for single and multiple offers, where π<> denotes any permutation of the list it is applied to:

==

$g\ \pi\langle ?x_1:t_1,\ldots,\ ?x_n:t_n,\ !E_1,\ldots,\ !E_m\rangle;\ B(x_1,\ldots,x_n)$

$-g\ \pi\langle v_1,\ldots,v_n,\ val(E_1),\ldots,\ val(E_m)\rangle\rightarrow$

$B(v_1,\ldots,v_n)$ for any value v_j of sort t_j (j=1...n)

==

Example:

a ?x:nat !(2+2) ?y:bool; B(x,y) –a<0,4,TRUE>→ B(0,TRUE)

It is important to realize that values <0, 4, TRUE> are offered/accepted at gate a *simultaneously*.

3.4 Choice: B1 [] B2

B1 [] B2 is a <behaviour expression> which offers a nondeterministic choice between the two alternatives B1 and B2. Its behaviour is captured by the two inference rules:

==

B1 –odi → B1'	*implies*	B1 [] B2 –odi → B1'
B2 –odi → B2'	*implies*	B1 [] B2 –odi → B2'

==

These rules essentially say that the action capability (set of possible actions) of a choice expression is the union of the action capabilities of its components; however, once an action is chosen from one component, the other component becomes unavailable.

Process Max2 (Section 2) was defined in terms of a choice expression. By applying the first rule above, where the premise B1 –odi→ B1' is instantiated to the already derived transition

'a?x:nat;b?y:nat; c!largest(x,y); stop –a<0>→ b?y:nat; c!largest(0,y); stop ', we derive:

```
a?x:nat; b?y:nat; c!largest(x,y); stop [] b?y:nat; a?x:nat; c!largest(x,y); stop

–a<0>→
b ?y:nat; c !largest(0,y); stop
```

In this example process Max2 is ready to accept the first input either at gate a or at gate b, and it will be up to the environment (possibly) to resolve this nondeterminism, by offering an integer *first*, say, at gate a, and *then* at gate b. Conversely, in the example below:

```
unsafe_machine[money, box] :=
        money ?x:coin;
                (box !cookie; stop   (* offer cookie *)
                []
                i; stop               (*power cut *) )
```

the first branch of the choice implies synchronization with the environment (a user), while the second branch can be taken by the process independently. The user can always insert the coin, but he may not be able to get the cookie.

3.5 Boolean guarding: [E] → B;

This expression behaves like B, but only if boolean expression E has value TRUE. The rule is:

B –odi → B' and *val*(E) = TRUE	*implies*	[E]→B –odi → B'

Example:

```
fair_machine[money, box] :=
        money ?x:coin;
                [x=10] → box !cookie; stop
                []
                [x≠10] → box !x; stop
```

The machine may accept different coins, but only when the coin happens to be one of value 10 a cookie is offered at gate box, otherwise the coin is returned. The example shows that the **if-then-else** construct of sequential programming languages has an obvious counterpart in LOTOS.

3.6 Parallel composition: B1 |S| B2

S stands for a set $[g_1,...,g_n]$ of user-definable gates. The |S| operator expresses the fact that B1 and B2 communicate through the gates in S. Three inference rules are given:

==

B1 –oi→B1' and gate(oi) ∉ S *implies* B1|S|B2–oi→ B1'|S|B2

B2 –oi→B2' and gate(oi) ∉ S *implies* B1|S|B2–oi→ B1|S|B2'

B1 –od→B1' and B2 –od→B2' and gate(od) ∈ S ∪ {d}

implies B1|S|B2–od→B1'|S|B2'

==

The rules essentially say that a parallel composition expression is able to perform any action that either component expression is ready to perform at a gate not in S, or any action that both components are ready to perform at a gate in S or at gate d (which represents successful termination and is discussed later). This implies that when process B1 is ready to execute some action at a gate in S, it is forced, in the absence of alternative actions, to wait until its "partner" process B2 offers the same action. (Similarly, Max and Franz were waiting for each other for playing "Largo's" and "Larghetto's".)

This *need* to synchronize imposes constraints to the possible temporal orderings of the events; thus parallel composition is a sort of an **and** operator which allows to add temporal ordering constraints to an already specified behaviour. To illustrate this, we give an example in pure LOTOS, where value communication is simply ignored and interactions are pure synchronizations, i.e. an observable action consists merely of a gate name.

Consider process Max2[a, b, c] introduced in Section 2, and degrade it to the pure LOTOS process:

```
X2[a, b, c] := a; b; c; stop [] b; a; c; stop
```

This process offers actions a and b, in either order, followed by action c. We may equivalently say that the only temporal constraints involved are "a precedes c" **and** "b precedes c", where the 'c' in the two costraints has to be regarded as a unique action. This is expressed by the parallel composition operator as follows:

```
X2[a, b, c] :=  a; c; stop  |c|  b; c; stop
```

Trees

To convince ourselves that the two definitions of X2 mean exactly the same thing we need the notion of **action tree**. So far we have applied inference rules to expressions to derive, at most, a sequence of three consecutive transitions (see Section 3.3). By exhaustively applying the rules to a given expression we get, in general, not just a transition sequence, but a **transition tree**. In a transition tree nodes are labelled by <behaviour expressions> (the starting expression being the label of the root), and arcs are labelled by actions. An **action tree** is a transition tree where node labels have been deleted. By exercising all applicable axioms and inference rules, the reader may easily check that the two alternative definitions of X2 above yield precisely the same action tree, shown in Figure 3.1.

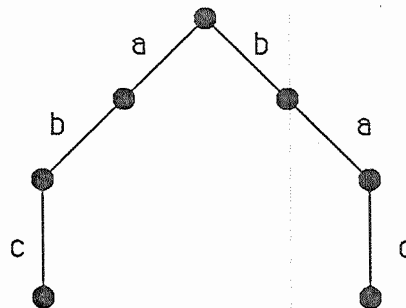


Figure 3:1 - Action tree of process X2

Let us go back to full LOTOS and consider the system:

```
users[ether, money] :=
        ether ?x:song; money !10; stop  (*a user*)
        |[ether]|
```

ether ?y:song; money !10; stop (*another user*)

This example represents a system of two users of some unspecified vending machine. Each of them offers (to listen to) any song, transmitted through **ether**, and then offers a coin of value 10 to the **money** gate of some machine. They are composed in parallel, but only gate **ether** is referred to by the parallel operator. This implies that they perform their actions at this gate simultaneously, while the actions at gate **money** are executed independently from each other: gate **money** is meant only for user-machine interaction. Notice that we are free to interpret each user's action at gate **ether** as either singing or listening (or even both things simultaneously). The essential point is that the two persons *agree* on a song, so that, after synchronization, variables x and y mean the same tune, the choice of which is made nondeterministically. In this respect the synchronization between two or more '?'-actions (input actions) can be seen as a way to extablish random values.

Two convenient shorthands are defined for the parallel composition operator. They correspond to limit cases for the set S of synchronization gates, and are presented below.

• Parallel composition with pure interleaving

B1|||B2 stands for B1|Ø|B2

i.e. S is the empty set of gates, and the third inference rule will never apply, except for termination at gate d. On the other hand, given an expression B1|||B2, if both B1 and B2 are ready for some action (say actions b1 and b2 respectively), the first two inference rules are both applicable, and both action orderings (b1 before b2, b2 before b1) are possible. Notice that b1 and b2 may even occur at the same gate. As B1|||B2 transforms, after an action, into an expression still involving the '|||' operator, we conclude that this case of parallel composition expresses nothing but any interleaving of the actions of B1 with the actions of B2.

• Parallel composition with the maximum of synchronization

B1||B2 stands for B1 |*gates*(B1) ∩ *gates*(B2)| B2

i.e. S is the set of gates shared by B1 and B2. Consider process Max3, as defined in Figure 2.1, degrade it to pure LOTOS, and remove the hiding operator '\mid', which we have not introduced yet, thus obtaining process:

```
X3'[in1, in2, in3, out] :=
            X2[in1, in2, mid] || X2[mid, in3, out]
```

where

```
        process X2[a, b, c] :=
                a; b; c; stop []  b; a; c; stop
        endproc
```

Symbol || stands here for |**mid**|, as this is the only gate shared by the two components of the parallel composition expression. The action tree for **X2[a, b, c]** has been already derived. The tree for X3' , shown in Figure 3.2b, is obtained by composing the two trees in Figure 3.2a, which are instances of the tree for X2, according to the rules of parallel composition, with **mid** as a synchronization gate. The construction rule is simple. A node of the composed tree stands for a pair of nodes of the component trees; the root stands for the roots. The three arcs emanating from the root of the composed tree, for instance, are obtained by considering any non-**mid** arc out of either component's root (there are three), and any **mid-mid** arc pair out of the pair of roots (there is none).

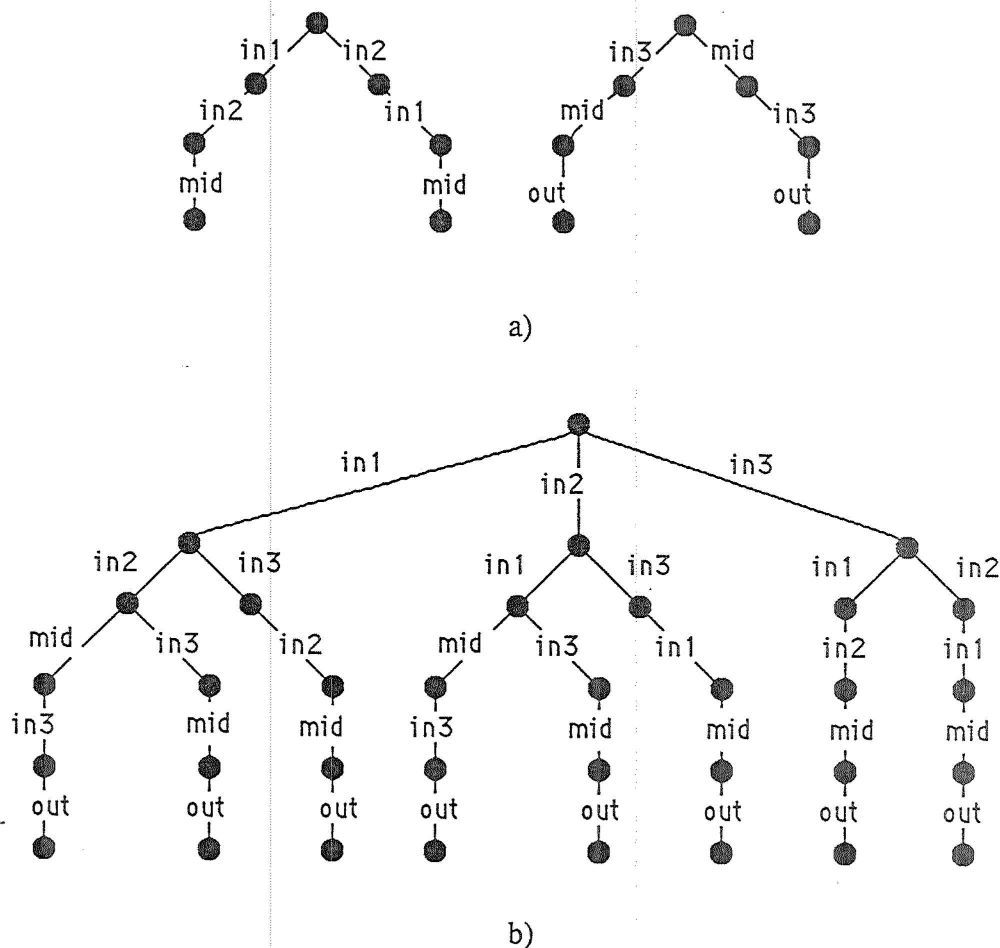


Figure 3.2 - Building the action tree of process X3'

3.7 Hiding $\qquad$ B\S

Hiding allows to internalize the gates of set S = [g1,...,gn]: the operator transforms the actions occurring at these gates into unobservable actions, thus making them unavailable for synchronization with respect to the environment. The inference rules are:

==

B –odi→ B' and *gate*(odi) ∉ S $\qquad$ *implies* $\qquad$ B\S–odi→ B'\S

B –o→B' and *gate*(o) ∈ S $\qquad$ *implies* $\qquad$ B\S–i→ B'\S

==

The following is an example of *internal* synchronization. Two actions synchronize at gate **money** and the resulting action is hidden. Correspondingly, gate **money** does not appear in the list of gates after the process name.

```
user_and_machine[box] :=
        ( money !10; box !cookie; stop          (*user*)
          ||
          money ?x:coin; stop                  (*machine*)
        )\[money]
```

The user inserts his coin into the machine he is composed with, which accepts the coin but is then unable to offer anything. However the user's action of pulling a box is still visible, and further composition of the whole system with, say, a broken vending machine offering cookies for free is a possibility.

In terms of trees, the hiding operation simply replaces the arc labels which include gates of S with label 'i'. Pure LOTOS process **X3'** introduced above can be refined by hiding gate **mid** :

```
X3[in1, in2, in3, out] :=
          (X2[in1, in2, mid] || X2[mid, in3, out])\mid
where
          process X2[a, b, c] :=
                  a; b; c; stop  []  b; a; c; stop
          endproc
```

The corresponding tree is immediately derived from the one in Figure 3.2b, by replacing mid-labels

with i-labels.

3.8 Successful termination

Exit can be seen as a process able to emit a successful termination signal at the special gate d, with or without value offers, and transform into the dead process **stop**. There are two cases:

(a) successful termination without value offers: exit
(b) successful termination with value offers: exit(E1,...,En)

where E_i (i = 1,...,n) is a <value expression>. The two corresponding axioms are:

==

exit −d<>→ stop

exit(E1,...,En) −d<*val*(E1),...,*val*(En)>→ stop
==

Gate d is used exclusively for this purpose, and will never be explicitely referred to in a specification (i.e., any gate accidentally named d in a specification is regarded as a "normal" gate, with no termination significance).

The way the successful termination action is "seen" in the inference rules introduced so far needs no special discussion; simply notice that the rules for parallel composition imply that the successful termination of B1|S|B2 may occur if and when both components are ready to successfully terminate (distributed termination). The ultimate purpose of successful termination is revealed, instead, in the context of sequential composition of processes. However, before tackling this construct we need to introduce the notion of *functionality*, which is meant to characterize processes with respect to the sorts of the values they offer at their terminations, if any.

Functionality.
Processes, or <behaviour expression>'s, can be characterized by an attribute called **functionality**. Its introduction of is motivated by the following consideration: whenever a process is capable of two or more (alternative) successful terminations, we want them to offer the same pattern of sorts to the process subsequently enabled. Some rules are thus needed for determining the functionality of behaviour expressions, together with some constraints on the ways expressions

with different functionalities can be combined. The functionality of a process which may successfully terminate with '<u>exit</u>(E_1,...,E_n)' is indicated by the list t_1,...,t_n, where E_i is of sort t_i, for i=1,...,n. In the case of n=0 (successful termination with no value communication), the functionality degenerates to the empty list, and is denoted **1**. The functionality of the remaining processes is denoted **0**. Notice that a process P with functionality different from **0** is not guaranteed to reach a successful termination; what we know of P is simply that *if* P terminates, then it will do so in the specified way. In fact a parallel <behaviour expression> such as 'a; <u>exit</u>(3) |[a]| b; <u>exit</u>(3)' is of functionality 'nat', but never terminates successfully.

The functionality of a process is indicated in its defining process abstraction, which associates the process name to a behaviour expression Bp of equal functionality, according to the following syntax:

Functionality of p and Bp	Process abstraction
0	$p[g_1,...,g_n](x_1{:}t_1,...,x_m)$ **:noexit** := B_p
1	$p[g_1,...,g_n](x_1{:}t_1,...,x_m)$ **:exit** := B_p
$t_1,...,t_n$	$p[g_1,...,g_n](x_1{:}t_1,...,x_m)$ **:exit**($t_1,...,t_r$) := B_p

For example, in specification Max_of_three (Section 2) we had 'Max2[a, b, c]**:noexit**:=...' because the choice expression which defines Max2 has functionality **0**. Notice that in order not to distract the reader with unnecessary details we have omitted such a functionality indication in several of the previous examples (e.g. in the vending machines)

Functionality can be defined rigorously by induction on the structure of behaviour expressions (see [12]), with rules such as "func(<u>stop</u>) = **0**", or "func(**a?x:t;** B) = func(B)". However rules and constraints on functionality have not yet been complely settled by the ISO experts, and we do not need here to investigate in detail the current proposals.

3.9 Sequential composition (enabling)

This operator expresses the fact that the successful termination of a process enables a subsequent process. In corrispondence with the two cases for the <u>exit</u> construct, there are two cases of sequential composition:

(a) sequential composition *without* value communication: B1>>B2

(b) sequential composition *with* value communication: B1>> def x_1:t_1... x_n:t_n in B2

There are two axioms for each case:

==

B1 −oi→B1'	*implies*	B1>>B2 −oi→B1'>>B2
B1 −d<>→B1'	*implies*	B1>>B2 −i→B2
B1 −oi→B1'	*implies*	B1>> def x_1:t_1... x_n:t_n in B2 −oi→ B1'>> def x_1:t_1... x_n:t_n in B2
B1 −d<v_1...v_n>→B1'	*implies*	B1 >> def x_1:t_1...x_n:t_n in B2(x1,...,xn) −i→ B2(v1,...,vn)

where n ≥1.

==

The first and third rules say that before B1's termination the actions of the compound expression are the actions of its first component B1. The second and fourth rules say that the termination action of B1 transfers control and, possibly, values to B2, and that this event is seen as an internal action of the compound expression. Variables x_1,..., x_n, used in B2, identify the values offered by B1 at its successful termination. For this value communication to be correctly defined it is necessary that the functionality of B1 be t_1,...,t_n. In the case without value communication, the functionality of B1 is required to be 1. Finally, the functionality of the whole construct is defined to be that of B2.

The enabling operator is conveniently used in conjunction with process instantiation, so that subparts of a system can be first defined separately and then instantiated in the desired sequence. An example is found in Section 3.12 (on process instantiation).

3.10 Disabling B1 [> B2

Process B1 may be disabled by process B2 according to the following rules:

==

B1 –oi→B1'	*implies*	B1 [> B2 –oi→ B1' [> B2
$B1\ -d<v_1 \ldots v_n>\ \rightarrow B1'$ where $n \geq 0$	*implies*	$B1\ [>\ B2\ -d<v_1 \ldots v_n>\ \rightarrow B1'$
B2 –odi→B2'	*implies*	B1 [> B2 –odi→B2'

==

Process B1 may (third rule) or may not (first and second rules) be interrupted by the first action of process B2; and the choice is nondeterministic. In the first case control is irreversibly transferred from the interrupted B1 to the interrupting B2. In the second case the interruptable B1 performs an action: if this action is not a successful termination (first rule) B2 survives. If the action is a successful termination (second rule) B2 disappears: the process it was expected to interrupt has completed its job, and the disabling process itself has been disabled.

The restriction is imposed that the functionalities of B1 and B2 be the same, or that the functionality of B1 be ω. The functionality of B1[>B2 is defined to be that of B2.

As an example, consider:

```
reusable_machine[money, box] :=
      (money ?x:coin;
            [x=10] → box!cookie; reusable_machine[money, box]
            []
            [x<>10] → box!x; reusable_machine[money, box]
      ) [> i; stop                (* powercut *)
```

Normally the machine accepts a coin, offers a cookie or the coin back, and endlessly repeats these actions. A powercut, modelled as an internal action, may interrupt such behaviour at any time, thus transforming the machine into a dead one. Fortunately when the machine crashes not only the box

but also the money gate becomes unaccessible.

3.11 Relabelling $B[h_1/g_1,..., h_n/g_n]$

It is important to stress that this is an *auxiliary* operator which does not belong to the LOTOS language but to its metalanguage. It serves to define two other constructs: process instantiation and summation on gates. The purpose of the operator is simply to rename some of the gates of a process: gate g_i becomes gate h_i, for i=1,...,n. Its rules are:

==

B –odi→ B' and gate(odi) not in $[g_1,...,g_n]$ *implies* $B[h_1/g_1,..., h_n/g_n]$ –odi→ $B'[h_1/g_1,..., h_n/g_n]$

B $-g_i$<$v_1,...,v_m$>→ B' and $1 \le i \le n$ *implies*

$B[h_1/g_1,...,h_n/g_n]$ $-h_i$<$v_1,...,v_m$>→ $B'[h_1/g_1,..., h_n/g_n]$

==

It follows from these rules that the action trees of B and B $[h_1/g_1,..., h_n/g_n]$ are the same, except for the renaming of gates which affects some of the arc labels.

3.12 Process instantiation $p[h_1,...,h_n]$ $(E_1,...,E_m)$

The instantiation of a process is, intuitively, a straightforward concept, similar to the invocation of a procedure in programming languages such as Pascal. In fact, we have already applied it, for instance, in the definition of process Max3, and that premature use has probably caused no difficulty of interpretation. Consider process abstraction:

$p[g_1,...,g_n](x_1{:}t_1,..., x_m{:}t_m)$: <functionality> := $Bp(x_1,...,x_m)$

p is the process name, g_i's are formal gates, x_j is a variable of sort t_j (j=1,...,m), <functionality> stands for **noexit**, **exit**, or **exit**$(t_1,...,t_r)$, as discussed in Section 3.8, and Bp is the <behaviour expression> abstractly defining the behaviour of **p**, where its dependency on variables

$x_1,...,x_m$ is evidenced. We say "abstractly" because the behaviour is defined in terms of gates and variables yet to be given actual values. Given process instantiation $p[h_1,...,h_n](E_1,...,E_m)$, the inference rule for process instantiation expresses how its actual parameters (gates $h_1,...,h_n$ and values $val(E_1),...,val(E_m)$) are passed to the corresponding process abstraction:

==

$Bp(val(E_1),...,val(E_m))\ [h_1/g_1,..., h_n/g_n]$ -odi→ B'

implies $p[h_1,...,h_n](E_1,...,E_m)$ −odi→ B'

==

where $[h_1/g_1,..., h_n/g_n]$ denotes the relabelling operation of Section 3.11. Notice that both the <gate list> and the <parameter list> in a process instantiation (and the corresponding lists in the process abstraction) may be missing.

Recursion

A **recursive** process is one which instantiates itself within its defining <behaviour expression> (either directly or through a chain of nested instantiations). Care must be taken in defining recursive processes, to avoid cases such as "**P:exit(nat) := exit(zero); stop [] P**" which could lead to an endless application of the instantiation rule which never yields a transition.

As an example of process instantiation, and of sequential composition, consider the following:

```
machine_scheme[money1, money2, box] :=
        Cash[money1, money2]
>>      def t : bool in Deliver[box](t )
>>      machine_scheme[money1, money2, box]
where
        ......
```

The dots after "where" stand for the definitions of process **Cash**, which accepts coins at gates **money1** and **money2**, and **Deliver**, which may offer a cookie at gate **box**. We do not specify them in detail, but simply assume that the former terminates with an "exit(TRUE)" or "exit(FALSE)", depending on coin values, insertion order and gate, and that the latter may or may not deliver the cookie, depending on the truth value above.

3.13 Local definition $\underline{let}\ x_1=E_1...x_n=E_n\ \underline{in}\ B(x_1,...,x_n)$

When lengthy value expressions E_i appear frequently within behaviour expression B, this construct allows to preliminarily bind their values to variables x_i, so that quicker reference to those values is possible within B. The rule is:

===

B (*val*(E_1),...,*val*(E_m)) –odi→ B' *implies* $\underline{let}\ x_1=E_1...x_n=E_n\ \underline{in}$ B ($x_1,...,x_n$) –odi→ B'

===

3.14 Summation on values $\underline{for}$ x:t $\underline{choice}$ B(x)

Let B(x) be a behaviour expression parametric in variable x. This construct offers a multiple choice among instances of B(x) which differ by the value v bound to variable x, and generalizes the (binary) choice operator B1 [] B2. The rule is:

===

B(v) –odi→B' *implies* $\underline{for}$ x:t $\underline{choice}$ B(x)–odi→B' for any value v of sort t

===

Action prefix involving <multiple value offer> and summation on values are strictly related constructs: 'a?x:t; B(x)' can in fact be seen as a shorthand for '$\underline{for}$ x:t $\underline{choice}$ (a!x; B(x))', as the application of the axiom for the former and of the inference rule for the latter yield the same actions and the same transformed processes.

3.15 Summation on gates $\underline{for}$ g $\underline{in}$ [$g_1...g_n$] $\underline{choice}$ B

This construct offers the choice between any one of the processes B[g_j/g] (with $1 \le j \le n$), where [g_j/g] is a relabelling (Section 3.11). The rule is:

==

$B[g_j/g]$ −odi→B' *implies* <u>for</u> g <u>in</u> $[g_1 ... g_n]$ <u>choice</u> B −odi→B'

where $1 \leq j \leq n$.

==

4. Behavioural equivalences

In the previous sections we have basically described the **expressive power** of LOTOS by presenting all its operators together with a few toy examples; in this section we brefly turn our attention to its **analytical power**, which mainly relies on a notion of behavioural equivalence.

One can describe systems at various levels of abstraction; for example it is possible to describe how they are structured internally in terms of predefined subcomponents or how they behave from the point of view of a user or of an external observer. LOTOS is a specification language which allows to specify systems at different descriptive levels. Indeed it does not differentiate between descriptions at different levels: they are all expressions in the language, and the difference between them is purely subjective. This range of descriptive levels is commonly partitioned into:

specifications, which are rather high level descriptions of the desired behaviour of the system, e.g. as seen by the user (extensional description);

implementations, which are more detailed descriptions of how the system works or of how it is constructed starting from simpler components (intensional description).

Needless to say, the border line between the two classes is quite arbitrary. The relationships between different LOTOS descriptions of a given system and, in particular, between specifications and implementations, can be studied by using a notion of systems equivalence, proposed in [18] and used for a CCS-like calculus in [17]. This equivalence, known as **observational equivalence**, is based on the idea that the behaviour of a system is determined by the way it interacts with external observers. Informally two systems are considered as equivalent whenever

no external observer can tell any difference between their behaviours.

Theories of equivalences turn out to be very useful. In fact, they allow not only to prove that an implementation is correct with respect to a given specification but also to replace complex subsystems with simpler, equivalent ones, within a large system, thus simplifying the analysis of the latter. As an example, suppose we have specified the **alternating bit protocol** [1] by describing how SENDER, RECEIVER and MEDIUM processes have to exchange messages and acknowledgements so that all messages sent will eventually be received, in the same order, even when the medium is unreliable. Then, to prove our **protocol** correct we need only to prove it equivalent to the **service** it is meant to provide, namely:

```
AB_Spec [in, out] := in ?x:message; out !x; AB_Spec [in, out]
```

Indeed this would amount to prove that, whatever the behaviour of the medium and whatever the internal exchange of messages, the external view of our protocol is one of a machine which inputs a sequence of messages and ouputs them preserving their temporal ordering. Once we have proved this equivalence we can substitute the detailed description of the protocol with the simple one above wherever we use the alternating bit protocol.

As an example of different descriptive levels consider the following pure LOTOS processes:

```
Process X3_Spec [in1, in2, in3, out] :=
    in1;  (in2, in3, out, stop
           []
           in3, in2, out, stop )
    []
    in2;  (in1, in3, out, stop
           []
           in3, in1, out, stop )
    []
    in3;  (in1, in2, out, stop
           []
           in2, in1, out, stop )
endproc
```

```
Process X3_Impl [in1, in2, in3, out] :=
            (X2[in1, in2, mid] || X2[mid, in3, out])\mid
where
```

```
        process X2 [a, b, c] :=
                a; b; c; stop
                []
                b; a; c; stop
        endproc
endproc
```

Process **X3_Impl** (identical to process **X3** introduced in Section 3.7) can be seen as an implementation, in terms of process **X2**, of process **X–3_Spec**. The latter gives a direct, unstructured description of a machine which outputs a signal only after receiving three input signals. In fact, our claim is that **X–3_Spec** and **X3_Impl** exhibit the same observable behaviour. The notion of observational equivalence allows to formally prove our claim.

Basically two LOTOS expressions are considered as equivalent whenever they can offer the same sequences of observable actions and transform into equivalent (sub)expressions. Observational equivalence is thus based on the transition relation (–odi→) defined by the operational semantics given in the Section 3. More precisely we need a transition relation on LOTOS <behaviour expressions> which involves *sequences* of actions but allows to abstract from unobservable ones. The definition of such a relation is reported below, and is based on the notational conventions fixed at the beginning of Section 3.

Definition 4.1

If s denotes strings $od_1 \cdot od_2 ... od_n$ in $(GV \cup DV)^*$ and ε denote the empty string then

i) $B\text{–}s\rightarrow B'$ if and only if there exist B_i, $0 \le i \le n$, such that $B=B_0\text{–}odi_1\rightarrow B_1\text{–}...\rightarrow odi_n\rightarrow B_n=B'$

ii) $B{=}\varepsilon{\Rightarrow}B'$ if and only if for some $n \ge 0$ we have $B\text{–}i^n\rightarrow B'$;

iii) $B = od \Rightarrow B'$ if and only if there exist B_1 and B_2 such that $B{=}\varepsilon{\Rightarrow} B_1\text{–}od\rightarrow B_2{=}\varepsilon{\Rightarrow}B'$;

iv) $B = s \Rightarrow B'$ (with $s \neq \varepsilon$) if and only if there exist B_i, $0 \le i \le n$, such that:

$$B = B_o = od_1 \Rightarrow B_1 = od_2 \Rightarrow B_2 \; ... = od_n \Rightarrow B_n = B''.$$

◆

Definition 4.2

If C and D are two elements of *B*, the set of <behaviour expressions>, then we say that C is **observationally equivalent** to D , and write $C \approx D$, if there exists a relation $\Re$ over *B*,

called a *bisimulation*, which contains the pair <C, D>, and such that if $B_1 \mathfrak{R} B_2$ then, for all s $\in$ $(GV \cup DV)^*$,

i. whenever $B_1 =s\Rightarrow B'_1$ then, for some B'_2, $B_2 =s\Rightarrow B'_2$ and $B'_1 \mathfrak{R} B'_2$

ii. whenever $B_2 =s\Rightarrow B'_2$ then, for some B'_1, $B_1 =s\Rightarrow B'_1$ and $B'_1 \mathfrak{R} B'_2$.

♦

Consider the following examples, in pure LOTOS:

B_1 = **a; (b; Stop [] i; c; Stop) [] a; c; Stop**

B_2 = **a; (b; Stop i; c; Stop)**

B_3 = **i; B**

B_4 = **B**

We have that:

i) $B_1 \approx B_2$ and

ii) $B_3 \approx B_4$

Proof In case i) the relevant bisimulation is $\mathfrak{R} = \{<B, B> \mid B \in \mathcal{B}\} \cup \{<B_1, B_2>\}$ since whenever $B_1 =s\Rightarrow B$, with $B \neq B_1$, we have that also $B_2 =s\Rightarrow B$, and whenever $B_2 =s\Rightarrow B$, we have that also $B_1 =s\Rightarrow B$; and $<B, B> \in \mathfrak{R}$. If $B_1 =s\Rightarrow B_1$ then $s = \varepsilon$; hence $B_2 =s\Rightarrow B_2$, and $<B_1, B_2> \in \mathfrak{R}$. In case ii), the relevant bisimulation is $\{<B, B> \mid B \in \mathcal{B}\} \cup \{<B_3, B_4>\}$ and the proof is similar. ♦

It is not difficult to prove that **observational equivalence**, as defined above, is an equivalence relation. However in many cases we are interested in substitutive relations, which guarantee that we can interchange equivalent terms in any context without affecting the overall behaviour.

Definition 4.3 A context **C**[ ] is a LOTOS expression where one or more subexpressions have been replaced by "holes". We write **C**[B] for the result of inserting expression B into each "hole".

♦

It is not difficult to see that $\approx$ is not preserved in every LOTOS context. For example we have seen that

i; Stop $\approx$ **Stop**

but if we consider the context C[ ] = **a; Stop** [] [ ] then we have that C[**i; Stop**] is not

equivalent to C[Stop] since

a; Stop [] i; Stop $=\varepsilon\Rightarrow$ **Stop**

while we only have

a; Stop [] Stop $=\varepsilon\Rightarrow$ **a; Stop [] Stop.**

(i.e. **C[Stop]** is not able to **simulate** the unobservable transition of **C[i; Stop]** to a state where no further action is possible)

If we want to rely upon the substitutive properties of systems, we need to consider equivalences which are also **congruences**. Indeed, we have already seen that $\approx$ is not preserved by [] contexts. We can define a new equivalence $\approx^c$ which is based on $\approx$ but satisfies our needs.

<u>Definition 4.4</u> If B_1 and B_2 are two <behaviour expressions> then we say that B_1 is **observationally congruent** to B_2, and write $B_1 \approx^c B_2$, if and only if $C[B_1] \approx C[B_2]$ for every context C[ ]. ♦

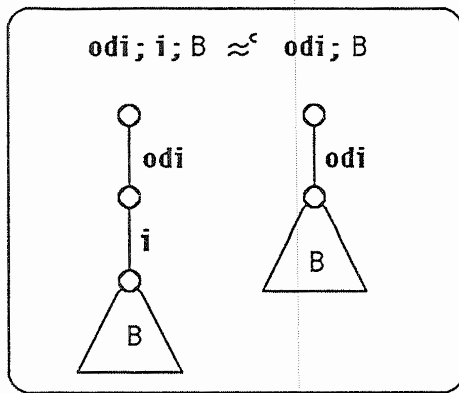
This definition obviously meets our requirements. Unfortunately in many cases it is unpractical to exhibit a bisimulations for every context. For this reason, more direct and practically tractable characterizations of $\approx^c$ have been given, such as the one below.

<u>Proposition 4.5</u> $B_1 \approx^c B_2$ if and only if for all odi $\in GV \cup DV \cup \{i\}$,

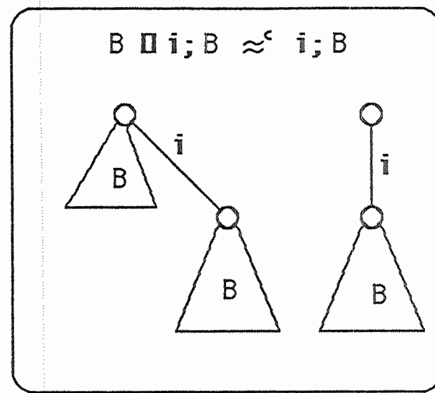
i. whenever $B_1 -odi\rightarrow B_1'$ then, for some B_2', $B_2 =odi\Rightarrow B_2'$ and $B_1' \approx B_2'$

ii. whenever $B_2 -odi\rightarrow B_2'$ then, for some B_1', $B_1 =odi\Rightarrow B_1'$ and $B_1' \approx B_2'$. ♦

On the basis of the operational semantics it is possible to prove several useful **congruence laws**. Two of them are reported in Figure 4.1, together with the induced transformations on the corresponding action trees.



Forgetful law

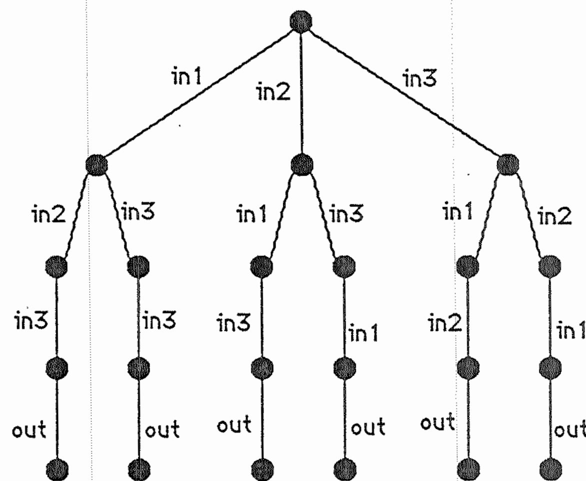


Absorbtion law

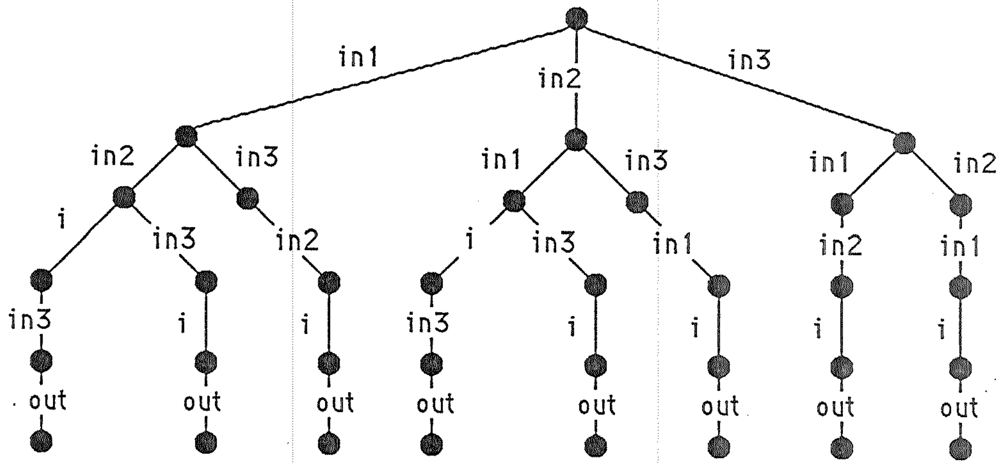
Figure 4.1 - Two congruence laws

Given two behaviour expressions B_1 and B_2 it may be possible to prove them observationally equivalent by repeatedly applying congruence laws as rewriting rules. Subexpressions are rewritten until B_1 and B_2 become syntactically equivalent. Notice that contextual rewriting is rendered possible by the fact that the relation we are dealing with is preserved by all LOTOS operators.

The laws above are sufficient to prove that processes X3-Spec and X3-Imp are observationally equivalent. Consider the two action trees of the processes, shown in Figure 4.2.



a) Action tree for X3-Spec



b) Action tree for X3-Impl

Figure 4.2 - Two observationally equivalent action trees

The proof can be easily conducted by simple graphical manipulations. First the forgetful law is applied to collapse six 'i' actions of the tree corresponding to X3-Impl. Then two subtrees of the resulting tree are reduced according to the absorbtion law, and eventually the first law can be applied twice again to give us a tree identical to the one for X3-Spec.

The proof of equivalence between two *full* LOTOS specifications (i. e. with value offers) is much more complicated than for pure LOTOS, because trees become, in general, infinitely branching, and because we have to handle and compare <value expressions>. Similarly to the case of processes X3_Spec and X3_Impl, we might wish to prove that specification Max_of_three in Section 2 can be seen as an implementation of the unstructured specification Max_of_three_Spec given below:

```
Specification Max_of_three_Spec

type nat is
   sorts  nat
   opns   zero:    --> nat
          succ:    nat --> nat
          largest: nat, nat --> nat
   eqns   largest(zero, x) = x
          largest(zero, x) = largest(x, zero)
          largest(succ(x), succ(y)) = succ(largest(x, y))
endtype (* nat *)

process Max3-Spec[a, b, c, out] :=

   a?x:nat; ( b?y:nat;  c?z:nat; c!largest(largest(x,y), z); stop
            []
```

```
               c?y:nat;  b?z:nat; c!largest(largest(x,y), z); stop)
   []
   b?x:nat; ( a?y:nat;  c?z:nat; c!largest(largest(x,y), z); stop
              []
               c?y:nat;  a?z:nat; c!largest(largest(x,y), z); stop)
   []
   c?x:nat; ( a?y:nat;  b?z:nat; c!largest(largest(x,y), z); stop
              []
               b?y:nat;  a?z:nat; c!largest(largest(x,y), z); stop)

endproc (*Max3-Spec*)

endspec (*Max_of_three-Spec*)
```

As a first step we could restrict ourselves to proving that *for given inputs* the synchronization trees associated with the two descriptions are observationally equivalent. Indeed, if we fix the three input values a priori, we have two trees which are structurally identical to the ones in Figure 4.2, but where labels in1, in2, in3 and out are replaced, respectively, by in1<v1>, in2<v2>, in3<v3> and out<largest(largest(v1, v2), v3)>. Notice that when the implementation performs 'in3<v3>' as a first action, the terminal action becomes out<largest(v3, largest(v1, v2))>, and our equivalence proof must rely also on properties of the specified data, namely on the (easily provable) commutativity of function 'largest'. The need to handle simultaneously processes and types, whose definitions follow two substantially different approaches, is certainly a source of difficulty in carrying out proofs.

The trick of proving equivalence for fixed input values is applicable only to restricted cases. In the general case proofs are based on the direct definitions of observational equivalence and congruence. and are likely to require massive amounts of computation. For this reason, and for the interplay between the analysis of data and process expressions, it would be naive to think to do this by hand, and many people are now trying to define proof techniques which can be assisted by the machine (e.g., in the ESPRIT/SEDOS project funded by the European Community). A promising start is found in [21] where results are proved which allow, under certain "reasonable" assumptions, to check whether a relation is a bisimulation by considering only derivations of the form B–odi→B' rather than B =od⇒B'. These results allow the definition of an algorithm for a step-by-step construction of a bisimulation between B_1 and B_2 (if there exists one), starting with the relations $\{\langle B_1, B_2 \rangle\}$ and adding further ordered pairs as necessary.

Observational equivalence is not the only key to compare LOTOS processes. In particular, with respect to the issue of specification versus implementation, one could argue that the latter need not display the *complete* behaviour allowed by the former, particularly when the specification is

nondeterministic and leaves some freeedom to the implementor. Because of this, we may want to prove that an implementation is an acceptable refinement of a given specification, rather than an alternative description with the same external behaviour. For example we would like to say that behaviour B is a legal implementation of specification '**i**; B [] **i**; C', which says that a satisfactory system behaves either like B or like C. A notion of preorder between CCS terms has been defined in [3] and has been extended in [4] to labelled transition systems (and the operational semantics of LOTOS is one of them). This preorder is based on the idea that a description *implements* another description whenever

1. The implementation **may** perform only actions which are allowed by the specification
2. The implementation **must** perform all the actions which cannot be refused by the specification

Saying it positively, B_imp implements B_spec whenever:

1. If B_imp **may** "do something" then B_spec **may** "do something" and
2. If B_spec **must** "do something" then B_imp **must** "do something".

This approach has the additional advantage of identifying processes which cannot be distinguished by external experiments but which are not observationally equivalent. Consider, as an example, the two behaviours below:

B_1 = **a; (a; a; Stop [] a; Stop)** and

B_2 = **a; a; a; Stop [] a; a; Stop**

Both of them will certainly support the observation of action sequences a and aa, and may or may not support the observation of aaa; any other observation will not be supported. In spite of this, they would be distinguished by observational equivalence, since:

B_1 –a→ **(a; a; Stop [] a; Stop)** = B_3 and

B_2 –a→ **a; a; Stop** = B_4 and B_2 –a→ **a; Stop** = B_5.

Clearly B_3 is not equivalent to B_4 because B_3 may refuse to accept the action sequence ' aa' while B_4 will certainly accept it; and B_3 is not equivalent to B_5 because B_3 may accept 'aa' while B_4 will certainly refuse it.

One of the advantages of LOTOS is that, on the basis of its operational semantics, different relations between specifications can be defined, which suit different analytical needs.

5. Conclusions

We have presented the specification language LOTOS and have focused our attention on the component which deals with the definition of interacting processes in terms temporal ordering of their actions and interactions. The language has a strong algebraic nature and the first impact with the apparently complex symbology of specifications may be discouraging. However, we hope we have proved, with the series of small examples given, that once some familiarity is achieved with the operators, systems can be specified in a natural way which reflects quite directly our primary, intuitive pictures of their structure and behaviour. The specifier, in general, does not feel forced to express unnecessary details with respect to his *abstract* understanding of the processes being specified. Whereas process specifications can be natural and concise, this may not be the case for type definitions. Abstraction (from implementation details) is certainly guaranteed by the abstract data type definitions of LOTOS. However some type definitions included in current specifications of OSI protocols and services (e.g. [5, 22, 23]) have shown to require lengthy lists of equations, to the detriment of readability. Some improvements are expected in this direction.

A major problem to be addressed in sketching a preliminary outline of a realistically complex specification relates to the tradeoff between process and type definitions. It is a fact that many elements of a system can be equally specified as processes or as data types. On one hand we may rule out this problem as a mere matter of taste and style. On the other hand we have mentioned that the interplay between processes and types has an impact also on the analysis of specifications. It is felt that a deeper understanding of the relation between the two components could be beneficial, and that some harmonization between them could be attempted (in the sense, for instance, of devising a common semantical model). This is also an area where interesting developments are possible.

We have illustrated a notion of observational equivalence and stressed its important role in an analytical sense. We have seen that the equivalence presented here is defined, in a sense, on top of the operational semantics of the language. It is important to observe that the same basis can support alternative definitions of equivalences, or preorders, in order to meet different analytical needs.

LOTOS has the merit (and takes the risks) of adopting relatively recent theories, which have been

mainly confined, so far, to academic environments. The wide exposure that the language is already undergoing, in terms of OSI protocols and services specifications, and the ongoing effort in building automated tools for the development, analysis and verification of specifications, are valuable opportunities to test the practical applicability of those theories. And these efforts are crucial for the success of LOTOS

Acknowledgement

The development of LOTOS started within ISO/TC97/SC16/WG1/FDT Subgroup C in 1981. Since then several people from various national Member Bodies of ISO and from the CEC research project COST11/bis FDT/TOS have contributed to the definition of the language. Subgroup C was initially chaired by Chris Vissers and then by Ed Brinksma, who coordinated the efforts which brought the language to its current shape. Both are at Twente University of Technology; they donated time and energy to this activity far beyond their assigned task. It would be impossible to acknowledge all people who partecipated to the development of LOTOS, since the technical contributions input at international meetings by national delegations or experts are often the result of discussions taking place in the various local institutions involved. The first author wishes to acknowledge, among those with whom he interacted more directly: Giuseppe Scollo, Ed Brinksma, Alastair Tocher, Elie Najm, Luigi Logrippo and Jan de Meer. Both authors wish to thank their colleague Diego Latella for fruitful technical discussions.

6. References

[1] Bartlett, K.A., R.A.Scantlebury, P.T.Wilkinson, A Note on Reliable Full-duplex Transmission over Half-duplex Lines, Communications of the ACM, 12, 5, pp.260-261, 1969.

[2] Brookes,S.D., C.A.R.Hoare, A.D.Roscoe, A Theory of Communicating Sequential Processes. Journal of ACM, Vol. 31, No. 3, pp. 560-599 , 1984.

[3] De Nicola, R., and Hennessy,M. Testing Equivalences for Processes. Theoret. Comput. Sci., Vol. 34, pp. 83-133, North Holland, Amsterdam, (1984).

[4] De Nicola, R., Extensional Equivalences for Transition Systems . Internal Report IEI B84-13, Pisa 1984.

[5] Di Stefano, A. (Ed.), Draft specification in LOTOS of the OSI session protocol (Version 1), ESPRIT/SEDOS/C1/WP/6/C, Univ. of Catania, October 1985.

[6] Ehrig, H., W.Fey, H.Hansen, ACT ONE: An Algebraic Specification Language with Two

Levels of Semantics, Technische Universitaet Berlin, Bericht-Nr. 83-03, 1983.

[7] Ehrig, H., and Mahr, B., Fundamentals of Algebraic Specification - 1, Springer-Verlag, Berlin, 1985.

[8] Goguen, J.A., J.W.Thatcher and E.G.Wagner, An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, IBM Research Report RC 6487, 1976. Also: Current Trends in Programming Methodology IV: Data Structuring, R.Yeh (Ed), Prentice Hall, 1978.

[9] Guttag, J., Abstract Data Types and the Development of Data Structures, Communications of the ACM, Vol.20, N.6, June 1977.

[10] Hennessy, M., and R.Milner, Algebraic Laws for Nondeterminism and Concurrency, Journal of ACM, Vol.32, No. 1, pp. 137-161, 1985.

[11] ISO DIS8824 - Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), 1985.

[12] ISO DP8807 (also: ISO/TC 97/SC 21 N 423), Information Processing Systems - Open Systems Interconnection - **LOTOS** - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, 1985.

[13] ISO DP9074 (also: ISO/TC 97/SC 21 N 422), Information Processing Systems - Open Systems Interconnection - **ESTELLE** - A Formal Description Technique Based on an Extended State Transition Model, 1985.

[14] ISO IS7498 - Information Processing Systems - Basic Reference Model for Open Systems Interconnection, 1983.

[15] Milne, G., CIRCAL and the Representation of Communication, Concurrency and Time. ACM Toplas Vol. 7, No. 2, pp. 270-298, 1985.

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

[17] Milner,R. A Complete Inference System for a Class of Regular Behaviours. Journal of Computers and Systems Sciences, Vol. 28, No. 3, pp.439-466, (1984).

[18] Park,D. Concurrency and Automata on Infinite Sequences. Proc. 5th GI Conference, LNCS 104, pp. 167-183, (1981).

[19] Plotkin,G. A Structural Approach to Operational Semantics, Lecture Notes, Aarhus University, (1981).

[20] Prieberg, F. K., Musica ex machina, Verlag Ullstein, Berlin-Frankfurt-Wien, 1960.

[21] Sanderson, M.T., Proof Techniques for CCS, Ph.D. Thesis, University of Edinburgh, CST-19-82, (1982).

[22] Scollo, G. (Ed.), Draft specification in LOTOS of the OSI transport protocol (Version 4), ESPRIT/SEDOS/C1/WP/5/C, Univ. of Catania, October 1985.

[23] Tocher, A.J., OSI Transport Service: A Constraint-Oriented Specification in Extended LOTOS, ESPRIT/SEDOS/C1/WP/11/IK, ICL, Kidsgrove, November 1985.

[24] Proceedings of the IEEE - Special issue on OSI, Vol.71. No.12, Dec. 1983

[25] Voelker,J., Helping Computers Communicate, IEEE Spectrum, Vol.33, No.3, pp.61-70, March 1986.

[26] Brinksma, H., A Tutorial on LOTOS, Proceed. 5th IFIP WG6.1 Workshop on Protocol Specification, Testing, and Verification (North Holland, Amsterdam, 1985).