Introduction to the ISO Specification Language LOTOS

Tommaso Bolognesi
CNUCE-C.N.R.
Via S. Maria 36 - 56100 Pisa, Italy
earn: bolog @ icnucevm

Ed Brinksma
University of Twente
P.O.Box 217 - 9700 AE Enschede, The Netherlands
uucp: mcvax!utinu1!infed
earn: hiddink @ hentht5

Abstract

LOTOS is a specification language that has been specifically developed for the formal description of the OSI (Open systems Interconnection) architecture, although it is applicable to distributed, concurrent systems in general. In LOTOS a system is seen as a set of processes which interact and exchange data with each other and with their environment. LOTOS is expected to become an ISO international standard by 1988.

Keywords: Concurrent Languages, Formal Description Techniques, Open Systems Interconnection, Protocol Specification, Specification Languages,

0. Introduction

LOTOS (Language of Temporal Ordering Specification) is one of the two Formal Description Techniques [26, 27] developed within ISO (International Standards Organization) for the formal specification of open distributed systems, and in particular for those related to the Open Systems Interconnection (OSI) computer network architecture [24, 39]. It was developed by FDT experts from ISO/TC97/SC21/WG1 ad hoc group on FDT/Subgroup C during the years 1981-86. The basic idea that LOTOS developed from was that systems can be specified by defining the temporal relation among the interactions that constitute the externally observable behaviour of a system. Contrary to what the name seems to suggest, this description technique is not related to temporal logic, but is based on process algebraic methods. Such methods were first introduced by Milner's work on CCS [33], soon to be followed by many closely related theories that are often collectively referred to as *process algebras*, e.g. [2, 5, 22, 32, 35]. More specifically, the component of LOTOS that deals with the description of process behaviours and interactions has borrowed many ideas from [22, 33].

LOTOS also includes a second component, which deals with the description of data structures and value expressions. This part of LOTOS is based on the formal theory of abstract data types, and in particular the approach of equational specification of data types, with an initial algebra semantics (see, e.g. [16]). Most concepts in this component were inspired by the abstract data type technique ACT-ONE [16], although there are a number of differences.

LOTOS is an FDT generally applicable to distributed, concurrent, information processing systems. However, it has been developed particularly for OSI. The main objectives for such a technique is that it should allow the production of OSI standards specifications that are:

- *unambiguous*, *precise*, *complete* and *implementation independent* descriptions of the standards;

- *readable reference documents* for OSI users, implementers and conformance testers;

- a *formally well-defined* basis for the verification and validation of the standards, and for the conformance testing of their implementations;

It is clear that these objectives are particularly important of a distributed, standard architecture, such as OSI. Machines must communicate and cooperate with each other, and informal, ambiguous specifications of the related software could easily lead to incompatible implementations. Furthermore, the possibility to carry out rigorous analysis of a protocol at the design level, that is, in an early stage of the development cycle, is crucial to avoid the proliferation of errors in the expectedly large number of its implementations.

The consideration of the requirements above has led to a number of design criteria for the language itself. The general criteria that have determined the present definition of LOTOS are:

- *Expressive power:* an FDT should be capable of expressing the wide range of properties that are relevant for the description of OSI services, protocols and interfaces.

- *Formal definition:* syntax and semantics of an FDT should have a complete and formal definition. In particular, the formal model on which the semantics of the language is based must support the development of an analytical theory for verification, validation and conformance testing.

- *Abstraction:* the language constructs should represent the relevant architectural concepts at a sufficiently high level of abstraction, where implementation oriented details are not expressed. This avoids the specification of undesirable constraints on implementers, and favours of a precise representation of the requirements.

- *Structure:* an FDT should offer means for structuring a specification in a meaningful and intuitively pleasing way. Good structuring implies readability, ease of maintenance, and may simplify the analysis. If desirable, structure may also be used to reflect the logical or even physical organization of an implementation.

LOTOS is expected to become ISO international standard by 1988.

The layout of the paper is as follows. Section 1 is meant to introduce informally the basic elements of the underlying model of LOTOS, namely processes, their interactions and their composition, in order to provide an intuitive support for their formal treatment in the rest of the paper. *Basic LOTOS* is introduced in Section 2. This is the subset of LOTOS where processes interact with each other by pure synchronizations, without exchanging values. In basic LOTOS we can appreciate the expressiveness of all the LOTOS process constructors (operators) without being distracted by interprocess value communication. Section 3 deals with equivalences, which are important for comparing specifications, and for giving a complete, formal semantics to the language. Value communication is not necessary to treat equivalences, and for this reason we introduce them right after Section 2 on basic LOTOS. The reader only interested in the "surface" of the language may safely skip this section.

The way data values are defined and expressed is the subject of Section 4 on data types. In Section 5 value expressions are integrated into the language: processes may exchange these values, or be parametrized by them, and we have *full LOTOS.* A small but complete LOTOS specification is provided in Section 6, as an example of the so called "constraint-oriented" specification style. Section 7 contains some concluding remarks and a number of pointers to the literature on LOTOS applications and tools. Already some tutorials on LOTOS have been published (e.g. [7]). These however still refer to previous versions of the language and/or are less complete in their presentation.

1. Processes

In LOTOS a distributed, concurrent system is seen as a *process*, possibly consisting of several sub-

processes. A sub-process is a process in itself, so that in general a LOTOS specification describes a system via a *hierarchy of process definitions*. A *process* is an entity able to perform *internal, unobservable actions*, and to interact with other processes, which form its *environment*. Complex interactions between processes are built up out of elementary units of synchronization which we call *events*, or *(atomic) interactions*, or simply *actions*.

Events imply process synchronization, because the processes that interact in an event (they may be two or more) participate in its execution at the same moment in time. Such synchronizations may involve the exchange of data. Events are *atomic* in the sense that they occur instantaneously, without consuming time. An event is thought of as occurring at an interaction point, or *gate*, and in the case of synchronization without data exchange, the event name and the gate name coincide.

The environment of a process P, in a system S, is formed by the set of processes of S with which P interacts, plus an unspecified, possibly human, *observer* process, which is assumed to be always ready to observe anything observable the system may do. And, to be consistent with the model, observation is nothing but interaction. Hence, when we say that process P *performs an observable action* we refer to the *interaction* between P and, at least, the observer. (Note that although we use the words *action* and *interaction* as synonyms, we may prefer one or the other depending on the context: we talk about the *action* performed by *one* process and the *interaction* involving *n* processes. The reason for blurring this distinction is simply that *n* processes together can be seen as *one* process.)

The most abstract representation of process P, able to interact with its environment via gates, say, *a* through *g*, is the black-box in Figure 1.1.

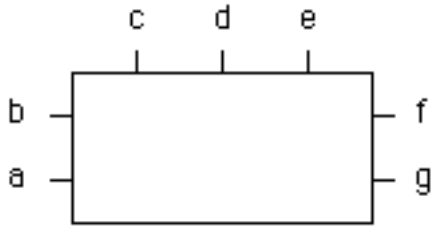


Figure 1.1 - Process P with gates *a* through *g*, as a black box

The *process definition* of P will then specify its behaviour, by defining the sequences of observable actions that may occur (be observed) at the seven gates of the process. We will soon represent such behaviour as a *tree* of actions.

Black boxes are the traditional intuitive representation for processes. Vending machines are also used, sometimes, to give a more concrete model of processes and interactions [22]. As another variant of the black box concept we introduce here a music instrument, to be called *proto-pianola*, where interaction with the environment is achieved via a keyboard. Speaking about these devices turns out to be essentially the same thing as speaking about LOTOS processes.

The proto-pianola fills a gap between the piano and the pianola. A piano is a completely passive instrument, since it plays only when its keys are pressed; conversely, a pianola is active, in that it

includes a predefined score (on punched paper rolls) and is able of automatically performing it. The proto-pianola is active and passive at the same time: it needs external interaction at the keyboard, for playing, and yet it possesses an internal score, and is able, from time to time, to perform autonomous choices. Figure 1.2a represents a 7-key proto-pianola. We can immediately think of it as a LOTOS process, called, say, PP1, and write:

PP1[a, b, c, d, e, f, g]

to indicate that the 7 keys are the gates through which the process interacts with its environment. In this case the environment, or the observer, is a player. The LOTOS view that processes cannot engage in more than one interaction at a time is reflected by the assumption of monophony for our musical instrument: it cannot produce more than one sound at a time.

The pressing of a key is an interaction between the proto-pianola, which is ready to have that key pressed, and the performer/observer, who is ready to press the key. Both parties participate to, or experience that event, and in fact the "genuine" LOTOS point of view insists on this symmetry, without distinguishing between *active* and *passive* roles. Observable actions (thus interactions) are simply identified by the gate where they occur (later they will be given more structure, to allow value communication between processes). We will sometimes express the fact that a process is ready for an interaction at gate *a* by saying that it *offers* observable action *a* to its environment.

A (one-finger) performer sitting at the keyboard of PP1 would not always succeed in pressing a chosen key, since some keys sometimes are locked, and the success in pressing a key depends on the tune played up to that point. The behaviour of the instrument (or, equivalently, its nondeterministic score) is depicted as a tree in Figure 1.2b. In this specific case the one-finger performer will be able to play a four note scale, but his only freedom is exercised in the choice of the initial key, which can be *a*, *d* or *g*. The other four keys are initially blocked. After the choice of *a* or *g*, the performer will succeed only in completing his scale, moving, respectively, upwards or downwards. If the initial choice had been for the central key *d*, both directions would have had a chance to be successfully executed, but this choice is not up to the performer. An i-labelled arc in the tree indicates an internal, unobservable action autonomously performed by the machine, which the performer cannot observe nor hear; the pair of i-labelled branches indicates that the choice of which key becomes unblocked after *d* is pressed is made by the machine.

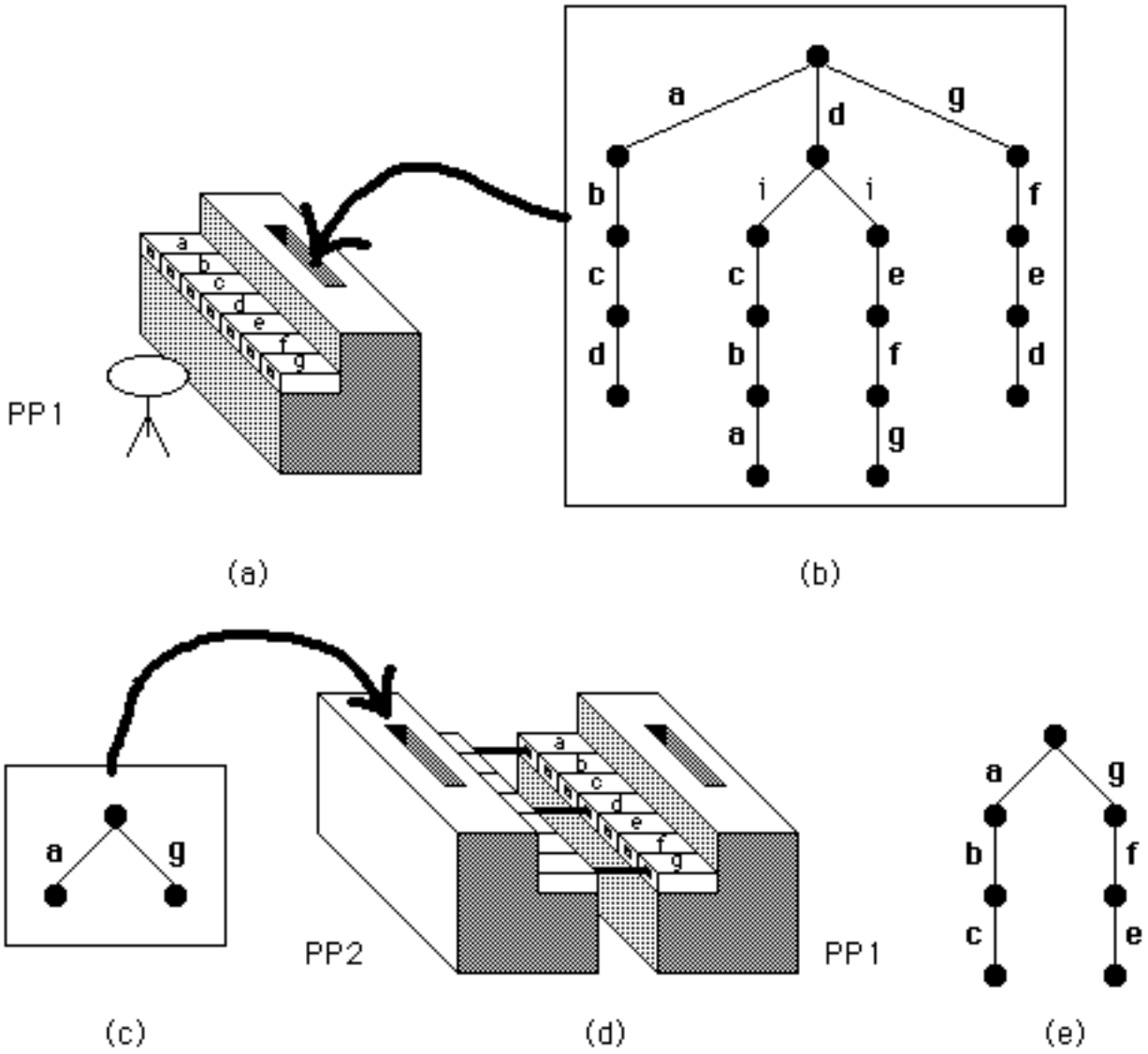


Figure 1.2 - Processes as proto-pianolas

The reason for the little holes observed in the front of the proto-pianola keys is revealed by Figure 1.1d, where two pianolas, PP1 and PP2, are coupled front to front. Their keyboards appear as mirror images. Metal bars have been inserted into the holes to couple some of the keys. Again a performer is supposed to play the trial and error game on the resulting "keyboard". His success in pressing an independent (uncoupled) key of pianola PP1 (PP2) depends only on the "score" of PP1 (PP2), but for a double-key (a pair of coupled keys) to be unlocked *both* scores must agree, at that point, on the executability of that note. This is exactly the idea of parallel composition of processes in LOTOS. The appropriate behaviour expression would be:

PP1[a, b, c, d, e, f, g]
|[a, d, g]|

PP2[a, b, c, d, e, f, g]

where '|[a, d, g]|' is a parallel composition operator: the two processes are coupled via the *synchronization gates* a, d and g, thus they may (in fact, must) synchronize only at these gates. The pair of coupled proto-pianolas is essentially a new instrument, and its behaviour is again representable by a tree. Suppose that the behaviour of PP2 be the one in Figure 1.1c. Then the behaviour of the new, double instrument, with the indicated key couplings, would be as in Figure 1.1e: only two three-note tone rows are allowed, starting from either extreme of the keyboard.

Observing and composing LOTOS processes is basically like playing and coupling proto-pianolas. However, since LOTOS has been mainly designed for specifying communication protocols for computer networks, it also includes features which the inventors of the proto-pianola failed to anticipate. One of these is *hiding*. This feature is better introduced by going back to the non-musical, more abstract world of black-boxes, where a process is represented as in Figure 1.1.

Consider Figure 1.3. The intended interpretation of the depicted system is as follows. Process Max3 is defined by composing in parallel two instances of process Max2. Each one of the component processes may interact with its *own* environment, which consists of the other instance of Max2 *and* the outer environment, via three gates; but the only synchronization gate between the two processes is *mid*. Notice that this gate is included in the outer box which represents process Max3. Since we insist that this box is really black, the gate is not visible from the outer environment: it has been hidden. The two process instances are thus allowed to independently interact with the environment at all gates except *mid*. At this gate they are required to synchronize with each other, without the environment observing (taking part to) these interactions: due to the hiding, these *interactions* have become *internal actions* of the system.

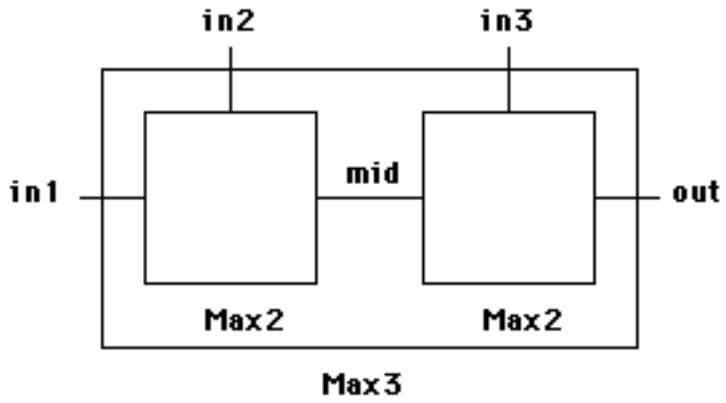


Figure 1.3 - Spatial representation of process Max3

The informal description above is made formal below.

```
process Max3 [in1, in2, in3, out] :=
        hide mid in
                (Max2[in1, in2, mid]
                | [mid] |
                Max2[mid, in3, out]
```

```
                    )
          where ...
endproc (* Max3 *)
```

The fact that he system may interact with its environment via actions (at gates) *in1*, *in2*, *in3*, *out*, is explicitly indicated in the first line of the specification. Since gate *mid* is hidden, by the *hide* operator, it does not appear in that list.

The partial specification above is completed in the next section.

2. Basic LOTOS

Basic LOTOS is a simplified version of the language employing a *finite* alphabet of observable actions. This is so because observable actions in basic LOTOS are identified only by the name of the gate where they are offered, and LOTOS processes can only have a finite number of gates. Three examples of observable actions that we have already met in the previous section are:

g
in2
out

The structure of actions will be enriched in *full* LOTOS by allowing the association of data values to gate names, and thus the expression of a possibly infinite alphabet of observable actions.

Basic LOTOS only describes process synchronization, while full LOTOS also describes interprocess value communication. In spite of this remarkable difference, we will initially concentrate on basic LOTOS for three reasons. First, within this proper subset of the language we can appreciate the expressiveness of all the LOTOS process constructors (operators) without being distracted by interprocess communication; second, for basic LOTOS we can give an elegant and, most importantly, *formal* presentation of the semantics, without boring the reader with cumbersome notation; third, behavioural equivalences are more conveniently introduced at this level. Full LOTOS will be introduced only in Section 5.

2.1. Process definitions and behaviour expressions

The typical structure of a basic LOTOS *process definition* is given in Figure 2.1, which completes the definition of process Max3 started in Section 1. As a convention we will use italics for syntactic categories, that is, nonterminal symbols (e.g. *behaviour expression*), and boldface for reserved LOTOS keywords (e.g. **process**).

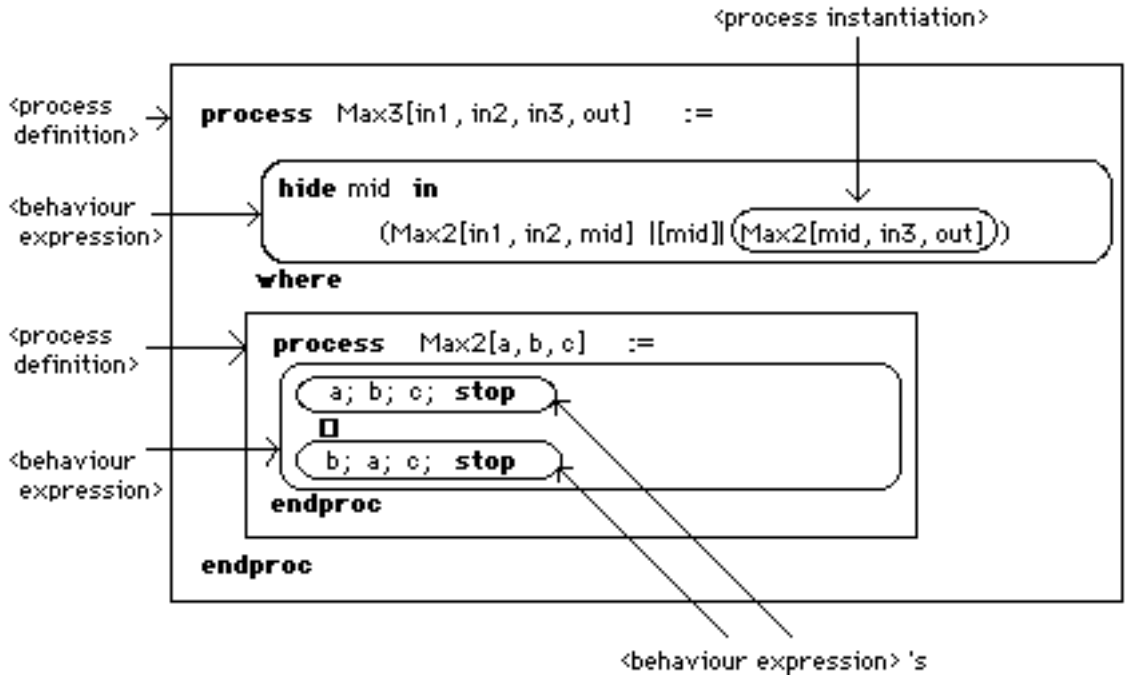


Figure 2.1 - Definition of process Max3

An essential component of a process definition is its *behaviour expression*. A *behaviour expression* is built by applying an operator (e.g., '[]') to other behaviour expressions. A *behaviour expression* may also include *instantiations* of other processes (e.g. Max2), whose definitions are provided in the **where** clause following the expression. Given *behaviour expression* B, we will allow calling B also "a process", for convenience, even when no process name is explicitly associated with the behaviour expressed by B.

The complete list of basic-LOTOS *behaviour expressions* is given in Table 2.1 below, which includes all basic-LOTOS operators. Symbols 'B', 'B1', 'B2' in the table stand for any *behaviour expression*. Any *behaviour expression* must match one of the formats listed in column SYNTAX. We have taken the metalinguistic liberty of representing some lists with dots. By inspecting Table 2.1 we may observe that basic LOTOS includes *nullary* operators (e.g. inaction), *unary* operators (e.g. action prefix) and *binary* operators (e.g. parallel composition), that is, operators applicable to, respectively, none, one and two *behaviour expressions*.

Table 2.1 - Syntax of *behaviour expressions* in basic LOTOS

NAME	**SYNTAX**
inaction	**stop**
action prefix	
- unobservable (internal)	**i ;** B
- observable	g**;** B

choice	B1 **[]** B2
parallel composition	
- general case	B1 **|[**g_1, ... , g_n**]|** B2
- pure interleaving	B1 ||| B2
- full synchronization	B1 || B2
hiding	**hide** g_1, ... , g_n **in** B
process instantiation	p **[**g_1, ... ,g_n**]**
successful termination	**exit**
sequential composition (enabling)	B1 >> B2
disabling	B1 **[>** B2

Operator precedences are as follows:

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

This means that, for example, expression

hide a in a; P [] Q >> R || S [> T

is equivalent to expression

hide a in (((a; P) [] Q) >> ((R || S) [> T)).

2.2. A basic process, two basic operators

Inaction: **stop**

The completely inactive process is represented by **stop.** It cannot offer anything to the environment, nor it can perform internal actions, and it is as basic in LOTOS as number zero in arithmetic. Notice that **stop** can be interpreted as the behaviour expression obtained by applying the *nullary* operator **stop** to zero arguments.

Action prefix: **i ;** B
g**;** B

This is a unary, prefix operator which produces a new *behaviour expression* out of an existing one, by prefixing the latter with an action (gate name) followed by a semicolumn. Examples of action prefix behaviour expressions, taken from process Max3 (Figure 2.1) are:

c; **stop**
b; c; **stop**
a; b; c; **stop**

Choice: B1 **[]** B2

If B1 and B2 are two *behaviour expressions* then B1 **[]** B2 denotes a process that behaves either like B1 or like B2. The choice offered is resolved in the interaction of the process with its environment. If (another process in) the environment offers an initial observable action of B1, then B1 may be selected, and if the environment offers an initial observable action of B2, then B2 may be selected. If an action is offered from the environment that is initial to both B1 and B2, then the outcome is not determined. An example of a choice *behaviour expression*, again taken from Max3, is

a; b; c; **stop []** b; a; c; **stop**

On the basis of the three constructs above, the behaviour of process Max2[a, b, c] defined in Figure 2.1 is now clear. As we did for proto-pianolas, we can immediately build the tree of actions associated with this expression. However, it is now time to describe the construction of *action trees* in a more precise, formal way, which could be systematically applied to any *behaviour expression*.

2.3 Operational semantics: growing trees from expressions

The *operational semantics* [38] of LOTOS provides a means to systematically derive the actions that

a process (*behaviour expression*) may perform from the structure of the expression itself. More precisely, given an expression B, what we derive are *labelled transitions*, that is triples of type:

$$B \text{—} x \rightarrow B'$$

where x is an action and B' is another *behaviour expression*: B may perform action x and transform into B'. In defining the semantics we will let:

G	denote the set of *user-definable* gates;
$g, g_1,\ldots,g_n$	range over G;
i	denote the unobservable action;
Act	denote the set $G \cup \{i\}$ of*user definable* actions;
μ	range over Act.

Furthermore, we will need to handle a special action (gate) ' δ', which is *not* user-definable, and whose occurrence indicates the *successful termination* of a process and the *enabling* of a subsequent process. We will thus let:

δ	be the successful termination action
G^+	be the set $G \cup \{\delta\}$ of observable actions
g^+	range over G^+
Act^+	be the set $Act \cup \{\delta\}$ of actions
μ^+	range over Act^+.

If *BE* is the set of *behaviour expressions*, then we may say, more formally, that the *axioms* and *inference rules* of the operational semantics allow the definition of the *labelled transition relation* '$\rightarrow$', which is a subset of *BE*x Act x*BE* ('x' is the cartesian product of sets). By applying axioms and rules to a given expression we build the *transition tree*, also called *synchronization tree*. (An introduction to these topics, and to the way axioms and inference rules are used, can be found by the interested reader also in [33], which gives the operational semantics of CCS.) 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.

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. In fact, since the beginning of the ISO/FDT activities, the definition of a *formal* semantics has been considered as a major requirement in defining Formal Description Techniques.

Semantics of inaction, action prefix and choice

No axiom or inference rule is associated with *behaviour expression* **stop**, and it is thus impossible to derive any transition from it. Hence we understand **stop** as a predefined LOTOS process which is

unable to perform any action or to interact with any other process.

The semantics of the action prefix *behaviour expression* is captured by a single axiom:

$$\frac{}{\mu;B \text{ —}\mu\rightarrow B}$$

where B is any *behaviour expression* and μ is either the unobservable action *i* or some observable action *g*. This axiom states the true fact, subject to no condition, that process 'μ; B' is capable of performing action μ and transform into process B. Notice that we use μ and not μ^+. This is because the user of the language is not allowed to express the successful termination action 'δ' directly, but only indirectly, by the 'exit' construct (to be discussed later).

B1 **[]** B2 is a choice *behaviour expression* which behaves either like B1 or like B2. Its behaviour is captured by the two inference rules:

$$\frac{B1 \text{ –}\mu^+\rightarrow B1'}{B1 \textbf{ [] } B2 \text{ –}\mu^+\rightarrow B1'} \qquad \frac{B2 \text{ –}\mu^+\rightarrow B2'}{B1 \textbf{ [] } B2 \text{ –}\mu^+\rightarrow B2'}$$

These rules are used to derive the actions of B1 [] B2 from those of B1 or B2. More precisely, 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 disappears from the resulting expression.

If we apply the axiom for action prefix, for example, to expression 'a; b; c; **stop**' we obtain the transition:

$$a; b; c; \textbf{ stop } \text{ –a}\rightarrow \text{ b; c; } \textbf{stop}$$

We may now use this result in applying the inference rule for choice:

$$\frac{a; b; c; \textbf{ stop } \text{ –a}\rightarrow \text{ b; c; } \textbf{stop}}{a; b; c; \textbf{ stop } \textbf{ [] } \text{ b; a; c; } \textbf{stop } \text{ –a}\rightarrow \text{ b; c; } \textbf{stop}} \quad \textit{implies}$$

We have thus derived a transition for a choice expression, based on the operational semantics of the language. By exhaustively applying the axiom and the rules in all possible ways, the reader may easily find the seven-node tree associated to the choice expression above.

As a final example of inaction, action prefix and choice we give the basic LOTOS description of the process illustrated in Figure 2.2. This process describes the externally observable behaviour of a full-

duplex channel between two points, which can be used only once for each direction. The description is abstract, in the sense that it only accounts for the ordering of inputs and outputs, and not for the data actually transmitted.

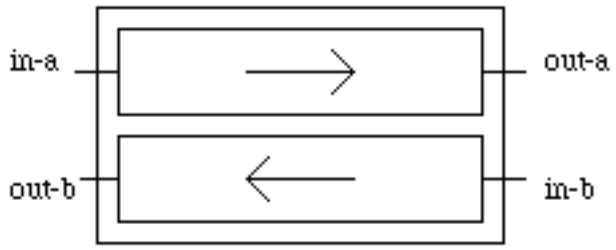


Figure 2.2 - A simple, full-duplex buffer

process duplex-buffer [in-a, in-b, out-a, out-b] **:=**
in-a; (in-b; (out-a; out-b; **stop**
[] out-b; out-a; **stop**)
[] out-a; in-b; out-b; **stop**)
[] in-b; (in-a; (out-a; out-b; **stop**
[] out-b; out-a; **stop**)
[] out-b; in-a; out-a; **stop**)
endproc

As it appears from the example above, describing a behaviour using only inaction, action prefix and choice forces the specifier to explicitly describe all different orderings in which independent actions may take place. This is of course a rather clumsy solution, and we will show below how parallel composition solves this problem in a more concise and structured way.

2.4 Parallelism

General case: B1 $|[g_1, ..., g_n]|$ B2

Let S = $[g_1, ..., g_n]$ be a set of user-defined gates, called *synchronization gates*. Given a parallel *behaviour expression* 'B1 |**S**| B2', the transitions it can perform depend on the transition capabilities of B1 and B2, and on S , as expressed by the following inference rules:

==========

$B1 \xrightarrow{\mu} B1'$ and $\mu \notin S$	*implies*	$B1\|S\|B2 \xrightarrow{\mu} B1\|S\|B2$
$B2 \xrightarrow{\mu} B2'$ and $\mu \notin S$	*implies*	$B1\|S\|B2 \xrightarrow{\mu} B1\|S\|B2'$

$B1 \xrightarrow{-g^+} B1'$ and $B2 \xrightarrow{-g^+} B2'$

and $g^+ \in S \cup \{\delta\}$ *implies* $B1|S|B2 \xrightarrow{-g^+} B1'|S|B2'$

==========

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 (excluding successful termination 'δ'), or any action that both components are ready to perform at a gate in S, or at gate δ. This implies that when process B1 is ready to execute some action at one of the synchronization gates, it is forced, in the absence of alternative actions, to wait until its "partner" process B2 offers the same action.

As an example, consider the parallel *behaviour expression*

'Max2[in1, in2, mid] |[mid]| Max2[mid, in3, out]'

used to define the behaviour of process Max3 in Figure 2.1. The action trees associated with the two instances of process Max2 are given in Figure 2.3. They are easily obtained by first building the transition tree of process Max2, also defined in Figure 2.1, and then by properly replacing its *a*, *b*, *c* labels with the actual gate names used in the two process instantiations (we are giving an informal preview of the semantics of *process instantiation*). By repeatedly applying the inference rules for parallel composition, the reader may check that the action tree for the parallel expression above is as depicted in Figure 2.4.

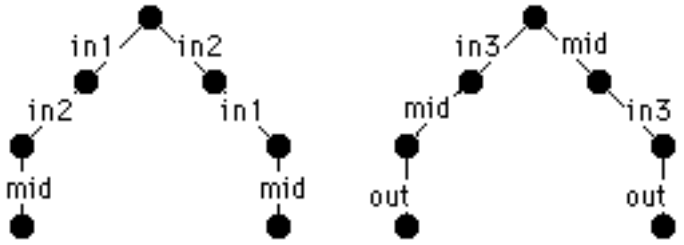


Figure 2.3 - Two action trees

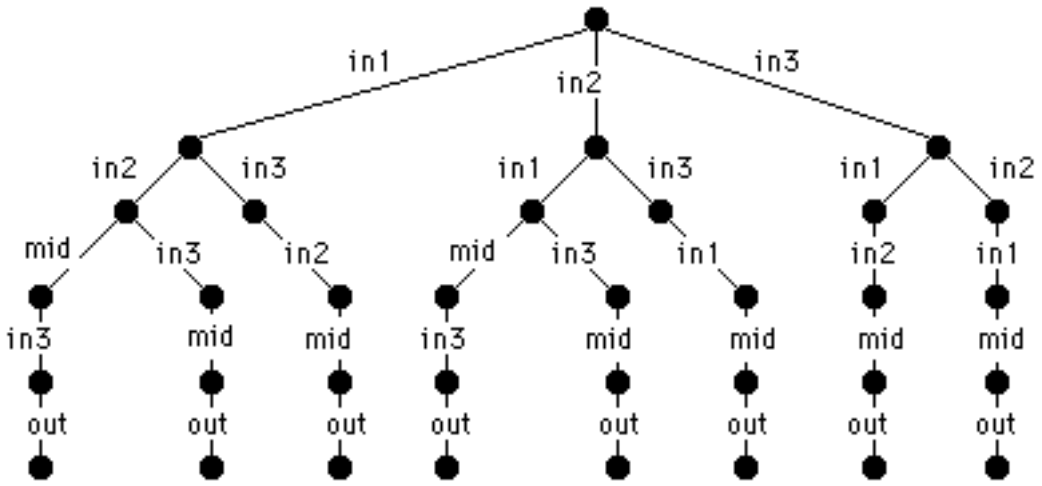


Figure 2.4 - A parallel composition of the two action trees of Figure 2.3

Notice that action *mid* is not hidden, as it was in the definition of process Max3. Thus, it is available for further synchronizations with the environment, exactly as is the case for actions *in1*, *in2*, *in3* and *out*. This feature of *multi-process* or *multi-way synchronization* is important for both technical and methodological reasons. The technical reasons have to do with specific applications. In some applications the structure of interprocess communication is reflected best by specifying a multi-way synchronization between processes. This is the case with, for example, message broadcasting.

The methodological reasons for introducing multi-way synchronization are related to the fact that where many processes synchronize on a single action, each of these processes may add constraints with respect to the occurrence of that action. In other words, complex temporal ordering relations among actions may be decomposed as the *conjunction* of several simpler constraints, each of which may be captured by a simple process definition. The complex constraint is then expressed by the parallel composition of all these simpler processes. This method is referred to as *constraint-oriented specification*. We illustrate this with a small example.

Consider (again !) the *behaviour expression* defining process Max2[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 constraints has to be regarded as a unique action. The conjunction of these two constraints is precisely expressed by the parallel composition operator as follows:

a; c; **stop** |[c]| b; c; **stop**

In fact, the action trees for the two expressions above turn out to be identical.

This approach of 'logical modularity' allows for an incremental combination of constraints. A further constraint such as "x precedes c" can be added later, with no need to affect the expression built so far:

(a; c; **stop** |[c]| b; c; **stop**) |[c]| x; c; **stop**

There exist two special cases of the parallel operator, for which convenient shorthands are defined. They are called *pure interleaving* and *full synchronization*.

Pure interleaving B1 ||| B2

When the set of synchronization gates, S, is empty, the parallel operator '|**S**|' is written '|||'. By inspecting the inference rules for parallel composition, it is clear that in this case the third rule can never be applied, except in the case of successful termination.

The two rules left account for the actions performed by the two component processes independently of each other. Given expression B1|||B2, if both B1 and B2 are ready for some action (say actions b1 and b2 respectively), then both action orderings (b1 before b2, b2 before b1) are possible. Notice that b1 and b2 may even be the same. Since 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.

We have now a means for expressing the simple-duplex-buffer specified at the end of Section 2.3. As suggested by Figure 2.2, such a process is best represented by a parallel composition of two *independent* processes (buffers). A more concise and better structured specification is:

```
process duplex-buffer [in-a, in-b, out-a, out-b] :=
        simplex-buffer [in-a, out-a]
    |||  simplex-buffer [in-b, out-b]

where
        process simplex-buffer [in, out] :=
                in; out; stop
        endproc
endproc
```

Full synchronization B1 || B2

When the set of synchronization gates, S, is the set G of *all* gates , then the parallel operator '|**S**|' is written '||'. Only the third inference rule for the parallel operator is applicable, and the two composed processes are forced to proceed in complete synchrony.

A typical example of use of this parallel operator is when the capabilities of a process are determined by two or more of its subprocesses.

```
process produce [a, b, c, d] :=
        item-available [a, b, c, d]
    ||  item-acceptable [a, b, c, d]

where
        process item-available [a, b, c, d] :=
                a;  (b; item-available [a, b, c, d]
                    [] c; item-available [a, b, c, d]
                    )
        endproc

        process item-acceptable [a, b, c, d] :=
                a;  (b; item-acceptable [a, b, c, d]
                    [] d; item-acceptable [a, b, c, d]
                    )
        endproc
endproc
```

In this simple example we can check that 'produce[a, b, c, d]' may only perform the sequence of actions 'a, b, a, b, ...'.

2.5 Hiding

Hiding allows one to transform some observable actions of a process into unobservable ones. These action are thus made unavailable for synchronization with other processes. The inference rules for the hiding operator are:

==========

$B \xrightarrow{\mu^+} B'$
and $\mu^+ \notin \{g_1, \ldots, g_n\}$ *imply* **hide** g1, ... , gn **in** $B \xrightarrow{\mu^+} B'$

$B \xrightarrow{g} B'$
and $g \in \{g_1, \ldots, g_n\}$ *imply* **hide** g1, ... , gn **in** $B \xrightarrow{i} B'$

==========

Any action occurring at a gate in the set of *hidden* gates is transformed into an i-action (second rule). Any other action, including 'i' and successful termination, is unaffected by the operator (first rule). We may say that hiding introduces unobservable actions in a specification implicitly, while by action

prefix they can be introduced explicitly.

As an example, consider the hiding *behaviour expression*

hide mid **in** Max2[in1, in2, mid] |[mid]| Max2[mid, in3, out]

used to define the behaviour of process Max3 in Figure 2.1. The hiding operator makes the synchronization between the two Max2 processes invisible, and excludes interference from their environment. We do this since we know that no other process will be added later to impose further temporal constraints to the occurrence of the *mid* action, which is to be considered as a "private" interaction between the two instances of Max2.

The action tree for the expression above is directly obtained from the tree of expression 'Max2[in1, in2, mid] |[mid]| Max2[mid, in3, out]', given in Figure 2.4, by replacing the *mid* labels with *i* labels. We will use this tree later (it can be found in Figure 3.5b).

2.6 Process instantiation and recursion P[g1, ...,gn]

A *process instantiation* 'P[g1, ...,gn]' is formed by a *process identifier* 'P' with an associated list $[g_1, ..., g_n]$ of *actual gates*. Such a process instantiation occurs in the *behaviour expression* defining some other process, or process P itself. The instantiation of a LOTOS process resembles the invocation of a procedure in a programming language such as Pascal. Of course a *process instantiation* refers to a *process definition* which must exist somewhere in the specification, and whose behaviour is defined in terms of a list $[g'_1, ..., g'_n]$ of *formal gates.* Example: in Figure 2.1, 'Max2[mid, in3, out]' is a process instantiation, where '[mid, in3, out]' is a list of actual gates, while '[a, b, c]' is the matching list of formal gates.

Although the interpretation of *process instantiation* is simple, in order to formally define how formal gates are replaced by actual gates we need to introduce an auxiliary operator, called *relabelling*, which is only used for talking *about* LOTOS, and not for specifying processes *in* LOTOS. Relabelling is a unary, postfix operator, which consists of a list of gate-pairs $[g_1/g'_1, ..., g_n/g'_n]$, and is interpreted as gate renaming: gate g'_i becomes gate g_i, i = 1, ..., n. It is required that $g_1 ... g_n$ be all different. Formally:

==========

$B -g'\rightarrow B'$,
$\phi = [g_1/g'_1, ..., g_n/g'_n]$, and $g/g' \in \phi$ *implies* $B\,\phi -g\rightarrow B'\phi$

$B -\mu^+\rightarrow B'$ and $\mu^+ \notin \{g'_1, ..., g'_n\}$ *implies* $B\,\phi -\mu^+\rightarrow B'\phi$

==========

Notice that internal action and successful termination are not affected by relabelling. It follows from these rules that the action trees of B and B ϕ are the same, except for the renaming of gates which affects some of the arc labels.

The rules for *process instantiation* are:

==========

If '**process** P[g'1, ..., g'n] **:=** B_P **endproc**' is a *process definition* then:

$B_P\,[g_1/g'_1, ..., g_n/g'_n] \xrightarrow{\mu^+} B'$ *implies* $P[g_1,...,g_n] \xrightarrow{\mu^+} B'$

==========

where $[g_1/g'_1, ..., g_n/g'_n]$ is the relabelling operator. The behaviour of instantiation '$P[g_1,...,g_n]$' is thus defined as the behaviour of the body B_P of the associated *process definition*, with the appropriate gate relabelling.

Recursion

Recursion is achieved, in LOTOS, by *process instantiation*, and is used to express infinite behaviours, namely those which involve action sequences of infinite length. Let us say that "process P invokes process Q" if either an instantiation of Q, or the instantiation of another process that invokes Q, occurs in the behaviour expression defining P. We say that process P is *recursive* if it invokes itself. As a simple example of recursion (and *process instantiation*) we refine the definition of the simplex-buffer given at the end of Section 2.4, by making it reusable:

```
process reusable-simplex-buffer [in, out] :=
                    in; out; reusable-simplex-buffer [in, out]
endproc
```

An infinite sequence **in; out; in; out;** ... of actions is now possible. Incidentally, an identical behaviour is obtained by the following definition:

```
process same-simplex-buffer [in, out] :=
                    in; same-simplex-buffer [out, in]
endproc
```

where every new process instantiation inverts the order of gates.

2.7 Successful termination and sequential composition

So far two ways to express sequentiality in specifications are available. We can do it directly, by prefixing an *action* to a *process*, or indirectly, by composing in parallel *two processes* in such a way that the last action of the first process synchronize with the first action of the second one. It seems desirable to have a *direct* way to express sequential composition of processes too, that is, to have a

separate operator for it. This may help in reflecting more clearly the structure of a system into the structure of its specification.

The idea behind the sequential composition operator is that the second process is enabled only if and when the first one terminates successfully.

Successful termination **exit**

Exit is a process (a nullary operator, a *behaviour expression*) whose purpose is solely that of performing the successful termination action δ, after which it transforms into the dead process **stop**. Its associated axiom is:

==========
exit – δ→ **stop**
==========

Action δ plays an key role in the sequential composition of processes, as shown below. It cannot be used directly in a specification, but only via the **exit** construct. Thus any gate accidentally named δ in a specification is regarded as a "normal" gate, with no termination significance.

Sequential composition B1 >> B2

The informal interpretation of this construct is that if B1 terminates successfully, and not because of a premature deadlock, then the execution of B2 is enabled.

==========

B1 –μ→B1' *implies* B1>>B2 –μ→B1'>>B2

B1 – δ→B1' *implies* B1>>B2 –i→B2

==========

The first rule accounts for the behaviour of B1 before its successful termination. The second rule shows that it is action δ, offered by B1, which enables B2, and that this passing of control is seen as an internal action *i.* Sequential composition and hiding are the only operators which introduce unobservable actions implicitly in a specification.

It is important to realize that the successful termination of the parallel composition of two processes is possible if and when both components are ready to successfully terminate, as expressed by the inference rules for the parallel operator. As a negative example consider this expression:

(a; b; **exit** ||| a; c; **stop**) >> second-process[...]

The expression is equivalent to ' (a; b; **exit** ||| a; c; **stop**)', since **stop** cannot contribute to the successful termination of the parallel subexpression, and no enabling of the second-process takes place.

The enabling operator is conveniently used in conjunction with process instantiation, so that subparts of a system can be first defined as separate processes and then instantiated in the desired sequence. An example is given below.

```
process Sender [ConReq, ConCnf, DatReq, DisReq] :=
        Connection-Phase [ConReq, ConCnf]
   >>   Data-Phase [DatReq, DisReq]
where
        process Connection-Phase [ConReq, ConCnf] :=
                ConReq; ConCnf; exit
        endproc
        process Data-Phase [DatReq, DisReq] :=
                (DatReq; Data-Phase [DatReq, DisReq]
                [] DisReq; stop
                )
        endproc
endproc
```

2.8 Disabling B1 [> B2

In almost any OSI connection oriented protocol or service it is the case that the 'normal' course of action can be disrupted at any point in time by events signalling disconnection or abortion of a connection. This has led to the definition in LOTOS of an 'application generated' operator, namely the *disabling* operator. Process B1 may be disabled by process B2 according to the following rules:

$B1 \xrightarrow{-\mu} B1'$	*implies*	$B1 \,[\!>\, B2 \xrightarrow{-\mu} B1' \,[\!>\, B2$
$B1 \xrightarrow{-\delta} B1'$	*implies*	$B1 \,[\!>\, B2 \xrightarrow{-\delta} B1'$
$B2 \xrightarrow{-\mu^+} B2'$	*implies*	$B1 \,[\!>\, B2 \xrightarrow{-\mu^+} B2'$

Process B1 may (third rule) or may not (first and second rules) be interrupted by the first action of process B2. 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 which B2 was expected to interrupt has terminated, and the disabling process itself is disabled.

As an example, let us first define the two processes:

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

process Disrupt [discon reason]  :=
        discon; reason; stop
endproc
```

Then the expression:

Activity [a, b, c] **[>** Disrupt [discon, reason]

is equivalent with:

```
(    discon; reason; stop
[]   a;  (    discon; reason; stop
         []   b;  (    discon; reason; stop
                  []   c;   ( Activity [a, b, c]
                            [> Disrupt [discon, reason]
)        )        )        )
```

With disabling, we have completed our presentation of the basic-LOTOS operators.

2.9 Nondeterminism and internal actions

Before giving a final example of a specification in basic LOTOS, we briefly discuss how nondetermism can be expressed in it. A simple example of nondeterminism is represented by the following expression:

a; b; **stop** [] a; c; **stop**

where the result of observing *a* is not determined. The unobservable action is also a source of nondeterminism, as shown by the expression

i; b; **stop** [] i; c; **stop**

(proto-pianola PP1 in Figures 1.2 (a) and (b) provides a similar example). In fact, from the point of view of an observer who is interested in observing action *b* (or *c*), the two expressions above offer the same uncertainty: in both cases the observation may succeed or fail (but in the first case a preliminary and always successful observation of *a* is also needed). We discuss now, with an example, the

special case of nondeterminism where the alternative is between an observable and an unobservable action.

We want to model a vending machine (although very little remains to be written about these devices after the publication of [22]). After accepting a *coin*, it will offer some *candy*. The user can obtain the latter by pulling a drawer.

```
process Vending_machine [coin, candy1, candy2] :=
    coin;
        ( candy1; Vending_machine [coin, candy1, candy2]
        [] candy2; Vending_machine [coin, candy1, candy2]
        )
endproc
```

Now suppose the system also contains a little devil that can try at any time to pull a drawer (before the user) and consume the candy.

```
process Devil [candy] :=
    candy; Devil [candy]
endproc
```

The total system, as observed by the client, is defined by

```
process System [coin, candy]:=
    hide candy' in
        Vending_machine [coin, candy, candy']
        |[candy']|
        Devil [candy']
endproc
```

By applying the axiom and inference rules introduced so far we could start the construction of the action tree for this System. We would then soon realize that the behaviour of the system is equally well described by this expression:

```
coin;   (candy; System [coin, candy]
        []  i; System [coin, candy]
        )
```

The first alternative in the choice subexpression represents the "normal" behaviour expected by the client. The second alternative is the Devil's one, where *i* models the hidden interaction between the Devil and the machine on action *candy'*. Although this interaction is invisible, we cannot drop it from our expression without affecting the behaviour of the system. If we write:

```
coin;   (candy; System [coin, candy]
        [] System [coin, candy]
        )
```

we are describing a system where the client can choose between getting his candy and inserting a new coin. In the original description, on the contrary, the occurrence of *i* is not at the client's discretion; it may simply happen, unnoticeable, and the client is confronted afterwards with only one possible course of action, viz. System.

The case of "asymmetric" nondeterminism with a choice between an observable and an unobservable action as was just discussed, is often found in an OSI context. Typically we have:

```
        normal-course-of-action
[]  i; disconnect indication; ...
```

where a process may be forced to accept a disconnect indication although, in principle, other alternatives exist.

2.10 An example in basic LOTOS

In all OSI protocol specifications one can identify parts that are responsible for the management of the connections in the underlying service, i.e. the setting up, using and disconnection of the logical communication channels that exist between the service users. Here we present a small and simplified portion of the manager of a Transport service, which would typically be a part of a Session protocol. We do not discuss the Transport service here; the uninitiated reader is referred to [44] for more information.

```
process Handler[ConReq,ConInd,ConRes,ConCnf,DatReq,DatInd,DisReq, DisInd]:=
        Connection-phase[ConReq,ConInd,ConRes,ConCnf,DisReq, DisInd]
  >>    (       Data-phase[DatReq,DatInd]
        [>      Termination-phase[DisReq, DisInd]
        )
  >>    Handler[ConReq,ConInd,ConRes,ConCnf,DatReq,DatInd,DisReq, DisInd]
where

        process Connection-phase[CRq,CI,CR,CC,DR,DI] :=
                (i; Calling[CRq,CI,CR,CC,DR,DI]
                []  Called[CRq,CI,CR,CC,DR,DI]
                )
        where

                process Calling[CRq,CI,CR,CC,DR,DI] :=
                        CRq;  (CC; exit
                                [] DI; Connection-phase[CRq,CI,CR,CC,DR,DI]
                                )
                endproc

                process Called[CRq,CI,CR,CC,DR,DI] :=
                        CI;     (i; CR; exit
                                [] i; DR; Connection-phase[CRq,CI,CR,CC,DR,DI]
                                )
                endproc

        endproc (* Connection-phase *)

        process Data-phase[DtR,DtI] :=
                i; DtR; Data-phase[DtR,DtI]
          []    DtI; Data-phase[DtR,DtI]
        endproc

        process Termination-phase[DR,DI] :=
                i; DR; exit
          []    DI; exit
        endproc

endproc (* Handler *)
```

3. Behavioural equivalences

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. In moving within this range of descriptive levels, it is common to distinguish between:

> *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).

LOTOS is a specification language which allows the specification of systems at different descriptive levels. In LOTOS the words 'specification' and 'implementation' have a relative meaning, not an absolute one. Given two (syntactically homogeneous) LOTOS specifications S1 and S2, we will say that S2 is an *implementation* of the *specification* S1 when, informally, S2 gives a more structured and detailed description of the system specified in S1. *Structure* in a LOTOS specification is another concept which cannot be given an absolute measure. We might say that a specification is made structured by a "generous" use of the parallel, the enable, and, perhaps, the disable operators, and of process definitions. For example, process 'duplex-buffer' in Section 2.4 shows more structure than its version in Section 2.3.

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 equivalence, proposed in [37] and used for a CCS-like calculus in [34]. 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. Theories of equivalences turn out to be very useful. In fact, they allow one 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.

A typical example of two different descriptive levels found in the OSI architecture is provided by the concepts of *protocol* and *service* [24, 39]. The *specification* of the N-service is *implemented* by the composition of the N-protocol entities with the (N-1)-service, and it seems natural to require that the two descriptions be equivalent. Unfortunately the complexity of OSI services and protocols is such that a proof of equivalence will certainly require the assistance of automated tools; and when the full language is used (this is of course the case for applications to OSI) the development of verification algorithms is a challenging task in itself. We are not concerned about analytical tools here. For our illustrative purposes it will be enough to give an example of a specification/implementation pair for which the equivalence proof can be carried out by hand. But before doing this, we want to stress the importance of equivalences from a slightly different perspective, namely for having a satisfactory definition of the formal semantics of the language. This will also give us the opportunity to shift our discussion of equivalences into the domain of trees, with the obvious pictorial advantages.

What is the *meaning* of the LOTOS expression below ?

a; (b; **stop** [] i; c; **stop**) [] a; c; **stop**

We might apply the operational semantics of Section 2 for deriving the action tree of Figure 3.1(a) from the expression, and then be tempted to say that the tree is the semantics of the expression.

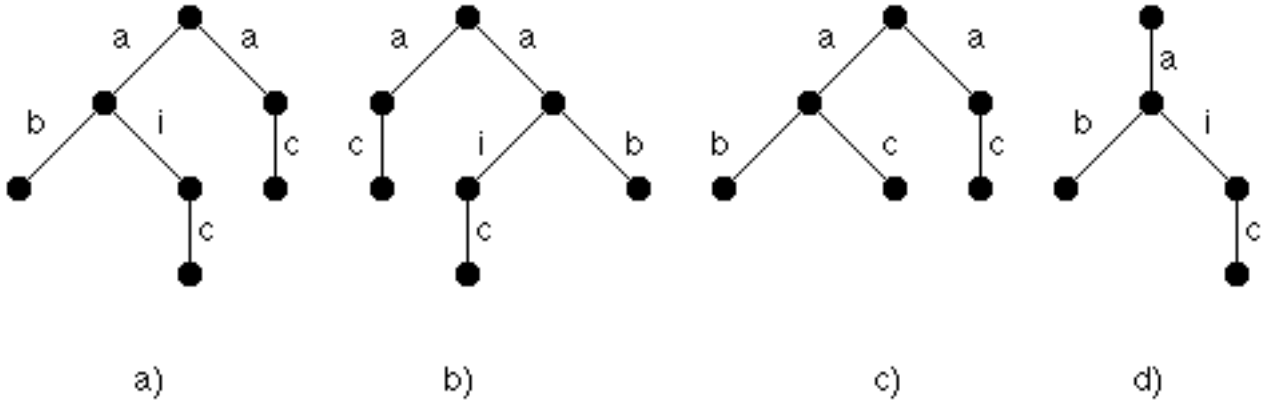


Figure 3.1 - Comparing action trees

Since we regard the tree as a description of a behaviour in terms of observable actions, we would consider the colour of the tree arcs and nodes as immaterial. Similarly, we would not object on the choice of a different ordering for the outgoing arcs of a node. For instance, we could accept the tree in Figure 3.1(b) as well. On the other hand, expression 'a; c; **stop** [] a; (b; **stop** [] i; c; **stop**)' also admits tree (a), or tree (b), as an action tree, and we may conclude that the two expressions should also be considered as equivalent. Rather than viewing the semantics of an expression as a tree, we will talk then about equivalence classes of trees and, consequently, of expressions. Once a proper notion of equivalence between trees is chosen, we will say that two expressions are equivalent (or that they have the same meaning) if their trees are in the same equivalence class. Thus the meaning of an expression can be identified with its equivalence class. We concentrate now on trees.

We have easily accepted the equivalence between trees (a) and (b) in Figure 3.1. Following the discussion on nondeterminism and internal actions in Section 2.9, we would not put tree (c), where an *i* action has been dropped, in the same class as (a) and (b). Consider now trees (a) and (d): do they represent the same observable behaviour ? In order to give a convincing answer we need a formal definition of *observational equivalence*.

The idea of observational equivalence is that two systems are considered as equivalent whenever we cannot tell them apart by *external* observations. As external observers we do not directly see trees (a) and (d) as in Figure 3.1, but we may only experiment with the keys of the two proto-pianolas in Figure 3.2, which incorporate these two trees as their hidden scores. As discussed in Section 1, observations consist in simply pressing keys, one at a time, and noticing whether they are free or blocked. Our experiments are formalized by the *observable sequence* relation ' $\Rightarrow$'. We refer to the notational conventions fixed in Section 2.3. However we may now imagine that B, B' and B_i denote

simply tree nodes, or states, rather than behaviour expressions (recall that behaviour expressions are node labels in derivation trees). An element of the *observable sequence* relation is a triple (B, s, B'), where s is a string of observable actions, and is written $B =s\Rightarrow B'$. The purpose of this relation is to abstract from the invisible actions that are on the path between two tree nodes.

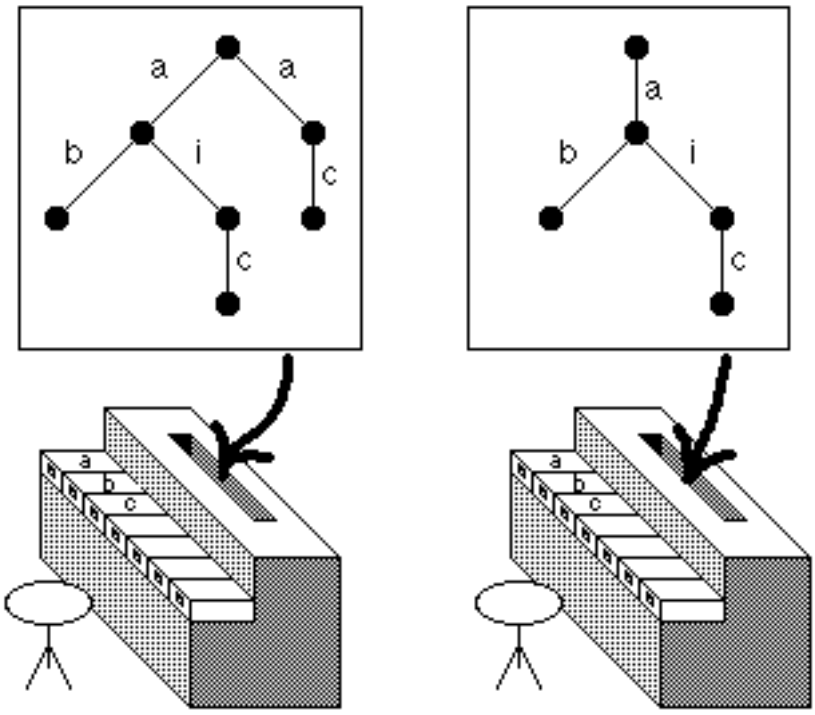


Figure 3.2 - Are these two proto-pianolas observationally equivalent ?

Definition 3.1

i) Let s denote a string $\mu^+_1 \mu^+_2 ... \mu^+_n$ of actions. We define relation $-s\rightarrow$ as the obvious extension of the transition relation (tree arcs) to action sequences:

$B-s\rightarrow B'$ if and only if there exist B_i, $0\leq i\leq n$, such that $B=B_0- \mu^+_1\rightarrow B_1..B_{n-1}- \mu^+_n\rightarrow B_n=B'$. In particular, for n = 0 we have $B- \varepsilon\rightarrow B$ for any B, where ε is the empty string.

ii) Let s denote now a string $g^+_1 g^+_2 ... g^+_n$ of *observable* actions, and let i^k denote a sequence of k (k≥0) i-actions. Then we have $B =s\Rightarrow B'$ whenever there exists a sequence $(i^{k0} g^+_1 i^{k1} g^+_2 ... g^+_n i^{kn})$ of actions such that:

$$B- (i^{k0} g^+_1 i^{k1} g^+_2 ... g^+_n i^{kn})\rightarrow B'.$$

This implies that $B = \varepsilon\Rightarrow B'$ whenever $B- i^k\rightarrow B'$, and that $B = \varepsilon\Rightarrow B$ for any B. •

Examples: given a tree path $B_0- i\rightarrow B_1-a\rightarrow B_2-i\rightarrow B_3-b\rightarrow B_4$, we may write:

$B_0 - iaib \rightarrow B_4$
$B_0 = ab \Rightarrow B_4$
$B_1 = a \Rightarrow B_2$
$B_0 = \varepsilon \Rightarrow B_1$
$B_0 = \varepsilon \Rightarrow B_0$.

Based on the observable sequence relation, we define a notion of *bisimulation*.

Definition 3.2
A relation $\Re$ between tree nodes is a *bisimulation* if for any pair (B_1, B_2) in $\Re$ and for any string s of observable actions:

i. whenever $B_1 = s \Rightarrow B'_1$ then, for some B'_2: $B_2 = s \Rightarrow B'_2$ and $B'_1 \Re B'_2$
ii. whenever $B_2 = s \Rightarrow B'_2$ then, for some B'_1: $B_1 = s \Rightarrow B'_1$ and $B'_1 \Re B'_2$. •

The idea of bisimulation is that two bisimilar nodes must be able to "simulate" each other, in terms of observable sequences, and then reach still bisimilar nodes. Finally:

Definition 3.3
Two tree nodes B_1 and B_2 are *observationally equivalent* , written $B_1 \approx B_2$, if there exists a bisimulation $\Re$ which contains the pair (B_1, B_2). •

When we talk about the observational equivalence of two trees we refer, of course, to the equivalence of their roots. For proving the observational equivalence of trees (a) and (d) in Figure 3.1, we must then provide a bisimulation between their nodes, which include also the pair of roots. The reader may check that the relation defined by the dashed lines in Figure 3.3 is in fact a bisimulation: any pair of nodes connected by a dashed line satisfies conditions (i) and (ii) in Definition 3.2.

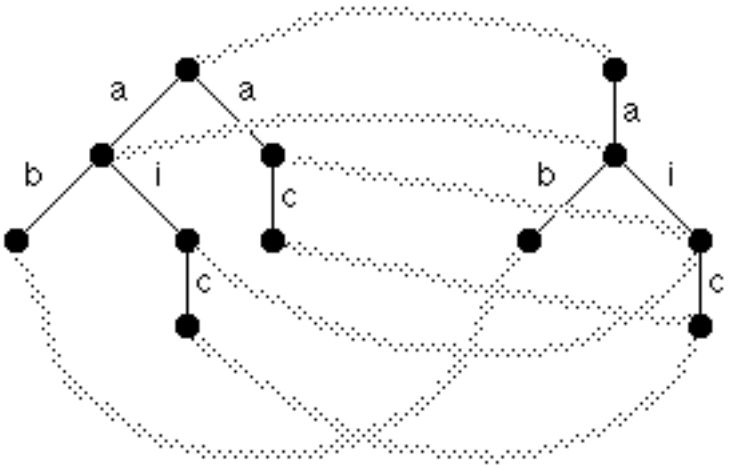


Figure 3.3 - A bisimulation

Hence the two trees are observationally equivalent, and we can write:

a; (b; **stop** [] i; c; **stop**) [] a; c; **stop** ≈ a; (b; **stop** [] i; c; **stop**)

Now that we have switched from trees back to expressions, we may try to solve our equivalence problems directly, by algebraic manipulations of the given expressions. In particular we might want to substitute some subexpression F of a given expression E with an expression F' equivalent to F, without affecting the overall behaviour, that is, without leaving the equivalence class of E. Unfortunately, observational equivalence is not a substitutive relation. We need to consider a different relation, called *observational congruence*, written ' $\approx^c$'.

We will not formally define *observational congruence* here (see [33]). It will only suffice to say that it is defined in terms of observational equivalence, it is stronger than it (that is: B1 $\approx^c$ B2 implies B1 ≈ B2), it is substitutive, and it satisfies a number of useful laws [21]. Three *observational congruence laws* are given in Figure 3.4, in tree form. Recall that, by our conventions, μ^+ denotes any action.

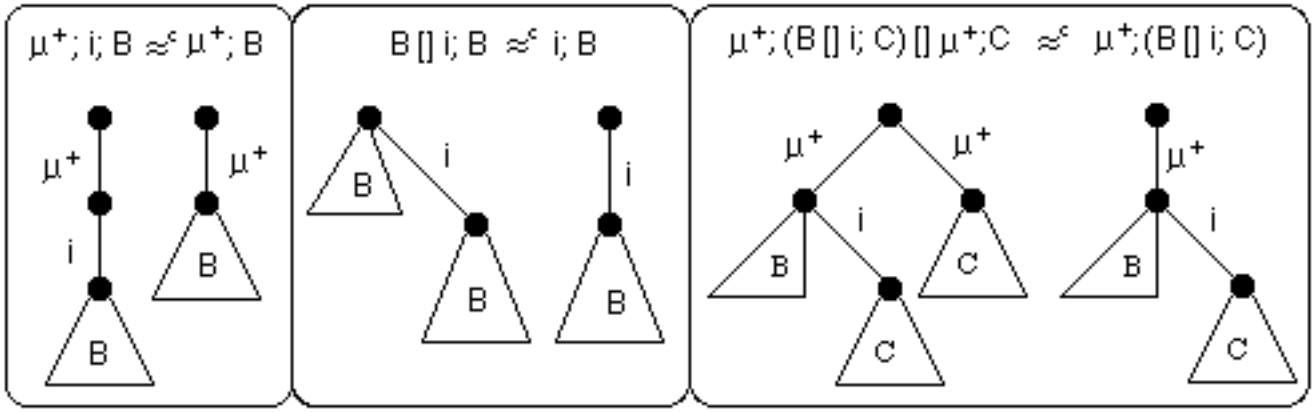


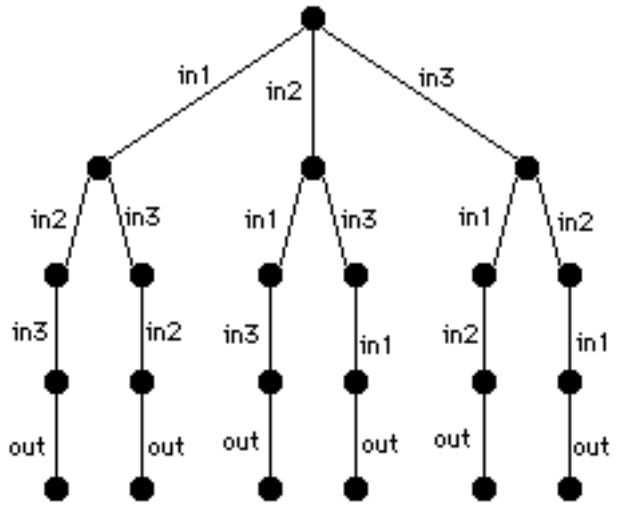
Figure 3.4 - Three observational congruence laws

Notice that the third law matches trees (a) and (d) of Figure 3.1, thus providing another proof of their observational equivalence. We will now use the first two laws to provide the proof of observational equivalence between two systems. Consider the following basic LOTOS processes:

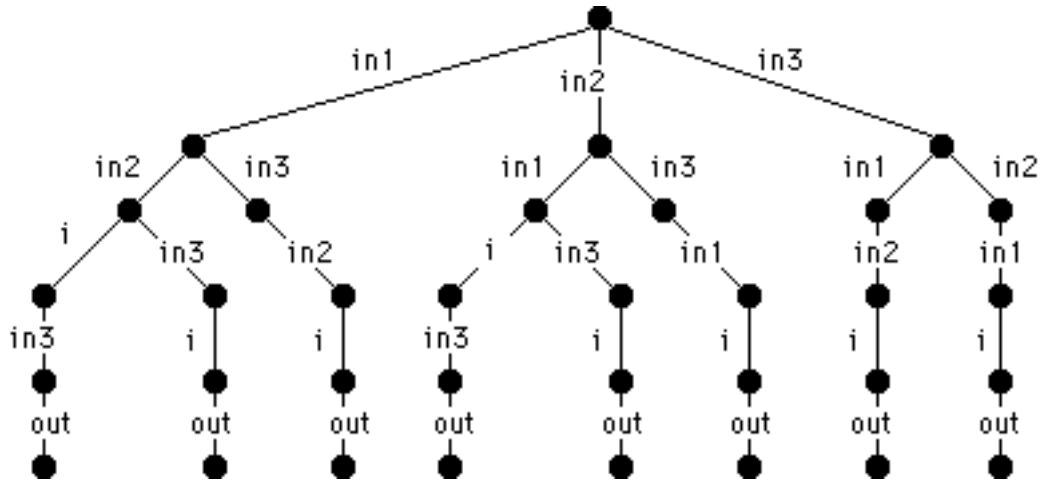
```
Process  Max3-Spec [in1, in2, in3, out] :=
     in1;   (   in2, in3, out, stop
            []  in3, in2, out, stop )
[]   in2;   (   in1, in3, out, stop
            []  in3, in1, out, stop )
[]   in3;   (   in1, in2, out, stop
            []  in2, in1, out, stop )
endproc


Process Max3 [in1, in2, in3, out] :=

          hide mid in
```

```
                (Max2[in1, in2, mid] |[mid]| Max2[mid, in3, out])
where
        process Max2 [a, b, c] :=
                a; b; c; stop
                []
                b; a; c; stop
        endproc
endproc
```

Process Max3 (which was already introduced in Section 2.1, Figure 2.1) can be seen as an implementation of process Max3_Spec, in terms of process Max2. The latter describes a black-box which outputs a signal only after receiving two input signals, in any order. Max3_Spec, instead, describes a black-box which outputs a signal only after receiving three input signals, in any order. Our claim is that Max3_Spec and Max3 are observationally equivalent. Consider the two action trees of the processes, shown in Figure 3.5.



a) Action tree for Max3-spec



b) Action tree for Max3

Figure 3.5 - Two observationally equivalent action trees

The proof can be easily conducted by simple graphical manipulations. The first congruence law is applied to collapse six 'i' actions of the tree of Max3. Then two subtrees of the resulting tree are reduced according to the second law, and eventually the first law can be applied twice again to give us a tree identical to the one of Max3-spec. The substitution of subtrees is allowed because we work with a congruence relation. In doing this we obtain a slightly stronger result: the two trees are not only observationally equivalent, but also congruent.

A survey on observational equivalence verification algorithms can be found in [4] (see also [30]).

Apart from observational equivalence, there exist a number of other ways to compare LOTOS processes. When specifying complicated behaviours it is a generally adopted strategy to specify all the behaviour that would be acceptable in implementations of the specification. This usually leads to a specification that includes a number of options of behaviour, all of which need not necessarily be part of any single implementation. In this case, one may want to establish that the behaviour of an implementation is an acceptable *reduction* of the behaviour of the specification, rather than verifying their equivalence.

To deal with this question, a number of asymmetric relations between behaviours have been suggested, which all are based on the same main idea. For CSP this 'implementation relation' was introduced in [11], and for CCS in [15], which was generalized to the context of labelled transition systems in [14]. The elaboration of such a relation for LOTOS can be found in [10].

The main idea is that 'B *red* S' (behaviour B reduces specification S, where B and S are processes) iff

i) B can only execute actions that S can execute; and
ii) B can only refuse actions that can be refused by S.

We still consider action *i* as invisible, and when we say "B can execute action x" we mean that x is observable, and that $B=x \Rightarrow B'$, for some process B'. The key to the understanding of this relation is that a specification S may be nondeterministic: after having interacted in a sequence of events s, S may both offer and refuse a particular set A of actions. Two instances of this relation are:

a) B *red* i; B [] i; C
b) B *red* i; B [] C

To fix ideas, let us consider case (b). It is clear that B can only perform actions that are also in i; B [] C, so that condition (i) is fulfilled. Also, every non-initial state of B has an equivalent state in i; B [] C, so that condition (ii) needs only verification for the initial state of B. It follows easily from $(i; B [] C) = \varepsilon\Rightarrow B$ that all actions that can be refused by B after a sequence of invisible actions can also be refused by i; B [] C after a sequence of invisible actions.

Note that C *red* i; B [] C does not hold: C may refuse initial actions of B that cannot be refused by

i; B [] C.

The implementation relations also induce equivalences between behaviours: B equivalent C iff B *red* C and C *red* B. This equivalence is referred to as *failure equivalence* in [11], and *testing equivalence* [15, 14, 10]. An advantage of these equivalences is that they do not distinguish between processes that cannot be distinguished by experiments, while this may happen with observational equivalence. As an example, consider the following processes:

B_1 = a; (a; a; **stop** [] a; **stop**)
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:

$B_1 -a\rightarrow$(a; a; **stop** [] a; **stop**) = B_3 and
$B_2 -a\rightarrow$a; a; **stop** = B_4 and $B_2 -a\rightarrow$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.

The name *testing equivalence* was chosen because in some sense this relation identifies exactly those processes that cannot be distinguished by testing. In [9] and [8] it is indicated how this relation may be further modified to support the practical testing of processes for conformance to their specification.

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 needs.

4. Data types

The representations of values, value expressions and data structures in LOTOS are derived from the specification language for abstract data types (ADT) ACT ONE [16]. The choice of *abstract* data types for LOTOS, as opposed to *concrete* data types, is consistent with the requirement of abstraction from implementation details which has been a guiding principle also in the design of the other component of the language (process definitions). A *concrete* data type implies a description of *how* data values are represented in memory, and *how* some associated procedures operate on them. In other words the data type is defined by explicitly giving its implementation. For example a Pascal queue can be defined as a list of records and a pair of procedures which manipulate it to realize the 'Add' and 'Remove' operations. An *abstract* data type can be seen as the *formal specification* of a class of concrete data types. It does not indicate *how* data values are actually represented and manipulated in memory, but only defines the essential properties of data and operations that any

correct implementation (concrete data type) is required to satisfy. Ultimately, an ADT definition identifies a mathematical object, namely an *algebra*, formed by sets of data values, called *data carriers*, and a set of associated *operations*. The reader interested in the specification of ADT's in general may refer to [19] and [20].

ACT ONE is an algebraic specification method to write unparametrized as well as parametrized ADT specifications. ACT ONE, and thus LOTOS, includes the following features for the production of structured specifications:

1. use of a library of predefined data types;
2. extensions and combinations of already existing specifications;
3. parametrization of specifications, and actualization of parametrized specifications;
4. renaming of specifications.

The most basic form of data type specification in LOTOS consists of a *signature* and, possibly, a list of *equations*.

4.1 Signature

The first step in specifying a data type consists of defining names of data carriers and operations. The names of the data carriers are referred to as *sorts*. The declaration of every operation will include its *domain* , which consists of a list of zero or more sorts, and *range* , which consists of exactly one sort. The sorts and operations of a data type are referred to as the *signature* of that data type.

Below we list a type definition of the natural numbers, which only consists of a signature. The definition is named 'Nat_numbers', so that it may be referred to by other definitions, and combined with them. The signature of Nat_numbers consists of the single sort 'nat', and the operations '0' and 'succ'. Operation 'succ' can be applied to a single element of sort 'nat', and yields also an element of 'nat' as a result, as indicated by the notation 'nat $\rightarrow$ nat'. Operation '0' is an operation that has no arguments, yet it yields an element of 'nat', as indicated by the notation ' $\rightarrow$nat'.

type Nat_numbers **is**

sorts nat
opns 0 : $\rightarrow$nat
succ: nat $\rightarrow$nat

endtype

We express the fact that an operation has *n* arguments by saying that it is an *n-ary* operation. Thus 'succ' is a unary operation, while '0' is a nullary operation. Nullary operations are called *constants*.

An additional example of a complete data type definition that consists only of a signature is the definition of a set of characters $\{a_1, ..., a_n\}$, where each character is defined as a constant:

type Character **is**

sorts char
opns $a_1, ..., a_n$, e: $\rightarrow$char

endtype

Note that there is a special symbol 'e' which is used in the next chapter to represent an error that is of sort 'char'.

The signature of a type gives all the information required to build syntactically correct *terms*, or *value expressions* , which represent data values of (some sort of) that type. A *term* is the result of applying an n-ary operation to n terms. In particular, a constant is clearly a term. More precisely, if a signature contains the constant declaration:

$$c : \rightarrow s$$

where s is some sort, then we say that c is a constant (or a term) *of sort s*. Similarly, if an operation is declared as:

$$op : s_1, ..., s_n \rightarrow s$$

then $op(t_1, ..., t_n)$ is a term *of sort s*, or an *s-term*, for short, where t_i is an s_i-term, for i = 1, ..., n.

For example, given the signature of type Nat_numbers above, we may construct the following terms, all of sort nat:

$$0, succ(0), succ(succ(0)), ...$$

which are meant to denote, respectively, the elements 0, 1, 2 ... of the algebra of natural numbers.

4.2 Equations

Suppose now that we want to define a 'plus' operation, which combines two nat-terms into a new nat-term:

$$\_+\_ : nat, nat \rightarrow nat.$$

The two underscore symbols '_' mark the position of the operands with respect of the operator, which is thus defined as an infix operator. We have now the possibility to write new nat-terms, such as '0 + succ(0)'. To interpret these new nat-terms correctly, we need a new construct to express properties of operations. This construct is the *equation.* The purpose of an equation is to state that two syntactically different terms denote the same value. For instance, we want to express the fact that terms 'succ(0)' and 'succ(0) + 0' denote the same value, or, more generally, that *for any nat-term* x, terms 'x' and 'x + 0' denote the same value. A correct definition of the properties of the '+' operator is:

```
eqns
    forall  x, y : nat:
    ofsort  nat
        x + 0          = x;
        x + succ(y)  = succ(x + y);
```

where the equations identify nat-terms (**ofsort** nat), and are valid whenever variables x and y are replaced by any pair of nat-terms (**forall** x, y : nat). The first equation expresses the behaviour of the plus operator when it is combined with the constant '0'. The addition with a non-zero number is covered by the second equation (note that term 'succ(x)' always denotes a non-zero number). By induction on the structure of terms, and by using these equations, it can be easily proved that any term containing one or more plus operations is equal to a term containing only '0' and 'succ'. This means that by introducing the plus operator we have not introduced terms that denote 'new' values which could not be expressed before. In this case, we say that the equations of '+' are *complete* w.r.t. the definition of 'Nat_numbers'.

The specification of the natural numbers extended with the plus operation is:

```
type Extended_nat_numbers is

    sorts   nat
    opns    0     :   →nat
            succ :   nat →nat
            _+_  :   nat, nat →nat.
    eqns
        forall    x, y : nat:
        ofsort    nat
            x + 0     = x;
            x + succ(y)  = succ(x + y);

endtype
```

4.3 Extensions and combinations of type specifications

To specify data types with a large number of operations we need language constructs to combine already existing specifications, and/or to extend them by adding further sorts, operations and equations. This way bulky specification can be given in a stepwise fashion, and a same, simple data type can be used as a basis for several, more complex definitions.

As an example of enrichment of a type, we re-define the type Extended_nat_numbers on the basis of type NaturalNumbers (both definitions are given in the previous section):

```
type Extended_nat_numbers is Nat_numbers with
    opns    _+_ :    nat, nat →nat.
    eqns
        forall   x, y : nat:
        ofsort   nat
            x + 0          = x;
            x + succ(y)  = succ(x + y);

endtype
```

In 'Extended_nat_numbers' we have imported the whole definition 'Nat_numbers' by referencing it in the heading of the former, and we have enriched it with one operation and two equations, given after the **with** keyword. In general we may combine several type definitions, and then add specific new elements:

```
type T is T1, ..., Tn with
    sorts ...
    opns ...
    eqns ...
endtype
```

4.4 Parameterized types

Parameterized data type specifications can be considered as partial specifications where only some general features of the type are described, and 'holes' are left to be filled later with further details. A *queue*, for instance, can be described as a parametrized type, which can later be actualized to become a *queue of integers* or a *queue of characters*.

In the absence of the parameterization feature, we could define the queue of natural numbers and the queue of characters as respective enrichments of the types Nat_numbers and Characters:

```
type Nat_number_queue is Nat_numbers with
    sorts   queue
    opns    create:  →queue
            add:     nat, queue →queue
            first:   queue →nat
    eqns    forall x, y: nat, z: queue
            ofsort nat
                first(create)  =  0;
                first(add(x, create))  =  x;
                first(add(x, add(y, z)))  =  first(add(y, z));
endtype
```

```
type Character_queue is Characters with
    sorts   queue
    opns    create:  →queue
            add:     char, queue →queue
            first:   queue →char
    eqns    forall x, y: char, z: queue
            ofsort char
                first(create)  =  e;
                first(add(x, create))  =  x;
                first(add(x, add(y, z)))  =  first(add(y, z));
endtype
```

In these new types the enrichment consists of a new sort 'queue' and of two new operations 'first' and 'add'. 'First' produces the first element at one end of the queue, and 'add' appends an element at the other end of it. The constants '0' and 'e' were already introduced respectively in the type definitions 'Nat_numbers' and 'Characters'. They are used to indicate an error when the 'first' operation is applied to an empty queue. It is clear that the two definitions above are almost the same. To avoid such duplication, we can make the sub-type of the queue that is variable a *formal part* of a parametrized type specification. Thus we specify a queue of a generic element, and the type of this element is made *formal*:

```
type Queue is
    formalsorts     element
    formalopns      e0 :  →element
    sorts   queue
    opns    create:  →queue
            add:     element, queue →queue
            first:   queue →element
    eqns    forall x, y: element, z: queue
            ofsort element
                first(create)  =  e0;
```

```
                first(add(x, create))  =  x;
                first(add(x, add(y, z)))  =  first(add(y, z));
endtype
```

The queue is now equipped with formal components 'element' (a sort) and 'e0' (a constant), which can be actualized by the 'NaturalNumbers' or 'Characters' as follows:

```
type Nat_number_queue is
    queue actualizedby Nat_numbers using
        sortnames nat for element
        opnames 0 for e0
endtype

type Character_queue is
    queue actualizedby Characters using
        sortnames char for element
        opnames e for e0
endtype
```

The formal part of a type definition can even contain formal equations, which are interpreted as requirements that must be fulfilled by an actual type that is substituted for it. For example we could have defined:

```
type Extra_queue is
    formalsorts     element
    formalopns      e0  :  →element
                    _*_  : element →element
    formaleqns      forall x, y: element
                    ofsort element
                        x * y = y * x
    sorts   queue
    opns    create:  →queue
            ...
endtype
```

This time we could actualize 'Extra_queue' with 'Enriched_nat', using the '+' operator for '*', but not with Characters.

4.5 Type renaming

Renaming of data type specifications is useful during the development of a specification in the case where an already defined data type is needed in a specific environment, but without any changes in the intended semantics. Therefore, renaming may be done explicitly by rewriting the data type definition with new sorts and operations. Changes in the signature imply changes in the declaration of variables and in equations. Expecially for long definitions this can be a cumbersome task.

The renaming operation avoids this drawback. Let us assume that the data type definition 'Queue' of the previous section is to be used in the OSI transport service environment, which deals with *channels* and *objects* to be transferred. Then the definition 'Queue' can be conveniently renamed as follows:

```
type Connection is
    Queue renamedby
        sortnames    channel  for  queue
                     object   for  element
        opnames      send     for  add
                     receive  for  first
endtype
```

5. Full LOTOS

In Section 2 we have presented the main features of LOTOS by illustrating a subset of the language based on a finite alphabet of events. Here we increase the expressive power of basic LOTOS by giving a finer structure to observable actions, thus to process interactions, using the facilities for the description of data structures and values presented in Section 4. As a major advantage, in full LOTOS we will be able to enrich synchronizations with value passing, thus providing interprocess communication.

While in basic LOTOS an observable action coincides with a gate name, in full LOTOS (or, simply, LOTOS) it is formed by a gate name followed by a list of zero or more values offered at that gate: g $<v_1 \ldots v_n>$. For example:

g <TRUE, "tree", 3>

is the observable action offering the boolean value TRUE, character string "tree", and natural number 3 at gate *g*. Since the offered values may range over infinite sets (e.g. the natural numbers), an infinite number of observable actions is expressible in full LOTOS.

We have given the operational semantics of basic LOTOS in Section 2 with the purpose to formally define the transition relation '$\rightarrow$'. The axioms and inference rules for full LOTOS are meant to achieve the same goal, except that a transition may now have the form:

$$B_1 \text{—}g\langle v_1 \ldots v_n\rangle \rightarrow B_2$$

that is, it may involve structured, observable actions. Here we will not insist in using a formal style of presentation however, in order to avoid the introduction of further notational complexity (the complete set of axioms and inference rules for LOTOS is found in [27]).

The integration of *type definitions* and *process definitions* in a full LOTOS specification is illustrated in Figure 5.1, which shows the syntax of a typical *specification* and a typical *process definition.*

specification:

specification typical_spec **[** *gate list* **] (** *parameter list* **) :** *functionality*
    type definitions
behaviour
    behaviour expression
where
    type definitions
    process definitions
endspec

process definition:

process typical_proc **[** *gate list* **] (** *parameter list* **) :** *functionality* **:=**
    behaviour expression
where
    type definitions
    process definitions
endspec

Figure 5.1 - Typical structures of *specification* and *process definition*

Process and *type definitions* may appear in the **where** clause of a *specification* or *process definition*, in either order or even interleaved. It clearly appears that a *specification* and a *process definition* have a similar structure. A minor difference is that the *behaviour expression* is preceded by the keyword **behaviour** in the first case, and by the definition symbol '**:=**' in the second case. A more significative difference is that some *type definitions* may appear before the *behaviour expression* of a *specification*, whereas this is not allowed in a *process definition.* Such *type definitions* are meant to be *global* definitions, which can be referenced in the *parameter list* of the *specification* and, potentially, by the environment where the *specification* is set to operate.

The inclusion of *type definitions* in specifications, and thus the possibility to express data values, are used to enrich the language in five different aspects. Values can be:

1) offered at gates, and exchanged among processes (enrichment of the *action prefix* operator);

2) used to express conditions to be satisfied for a given behaviour to take place (introduction of the new construct of *guarding* , and enrichment of *action prefix* with *selection predicates*);

3) used to generalize the *choice* operator.

4) used to instantiate parametric process definitions or actualize parametric behaviour expressions (parametric process definition and instantiation, '*let*' construct);

5) passed by a successfully terminating process to a subsequent, enabled process (enrichment of *successful termination* and *enabling* operators);

For everyone of the five features above we have indicated the corresponding constructs of the language which are affected, or newly introduced. We will address them one-by-one in the sequel.

5.1 Value offers and interprocess communication

In formally describing formal languages it is very important to clearly distinguish between the linguistic and the meta-linguistic levels. Going back to basic LOTOS for a moment, consider the following transition of an action prefix expression:

$$a; B \text{ —}a\rightarrow B$$

It is clear that the first occurrence of *a* is an actual syntactic element of the language, while the second one pertains to the meta-linguistic level used to formally describe the semantics of LOTOS expressions. Similarly, the notation '$g<v_1 \ldots g_n>$' used for structured actions has only meta-linguistic value, and does not belong to the actual LOTOS syntax. We show now how the syntax of the (observable) *action prefix* operator is enriched in order to express such structured, observable actions. Since the structure of observable *action prefix expression* is:

action denotation **;** *behaviour expression*

(where the semicolon is a terminal symbol), we will concentrate on (observable) *action denotation* . The general structure for this construct is:

$$g \ \alpha_1 \ \alpha_2 \ldots \ \alpha_n$$

where *g* is a gate name and the α's represent a finite list of *attributes*. Two types of attribute are possible: a *value declaration* and a *variable declaration.*

Value declarations

A *value declaration* has the form '!E', where E is a *value expression*, i.e. a LOTOS expression describing a data value. Examples of *value declarations* are:

!(3+5), **!**(x+1), **!**TRUE, **!**'example', **!**not(x), **!**min(x,y).

If we combine a *value declaration* attribute with a gate name *g*, and its *value expression* describes the value v, then the *action denotation* describes action g<v>. For example, **tsap !(3+5)** describes action tsap<8>. If the value expression contains variables, then for each set of actual values for those variables an action is described. For example, if x=3 and y=5, then **g !min(x,y)** describes event g<3>. The binding of variables to values is determined by the context, as explained below. In conclusion, if E is a *value expression*, and B is a *behaviour expression*, then the *action prefix expression* 'g !E; B' may offer the value of E at gate *g* and transform into B:

g **!**E**;** B —g<value(E)>→ B

Example

c **!**largest(0, 3)**; stop** —c<3>→ **stop**

Variable declarations

A variable declaration has the form '?x:t', where *x* is a name of a *variable* and *t* is its *sort identifier*. As was explained in Section 4, the *sort identifier* indicates the domain of values over which *x* ranges. Examples of *variable declarations* are

? x:integer, **?**text:string, **?**x:nat, **?**active:boolean.

If a gate name is attributed with a *variable declaration* '?x:t', then action denotation 'g ?x:t' describes a set of actions, viz. the set of all actions g<v> for all values v in the value domain of sort t. Thus ,for example, 'a **?**x:nat' describes the set of actions {a<n> | n ∈ N} . Everyone of these actions is the label of a possible transition of the transition tree. The effect of a transition in this case is slightly more complicated than for the case of *value declaration*. Consider the *action prefix behaviour expression* 'g **?**x:t; B(x)', where B(x) denotes a *behaviour expression* parameterized by some variable x occurring in some *value expression*. Then the associated transitions are:

g ?x:t; B(x) —g<v>→B(v)

where 'v' is any value in the domain of sort t, and B(v) indicates that after the transition has occurred, the value 'v' has been substituted for 'x' in B(x). B(x) represents the *scope* where the binding associated with the *value declaration* '**?**x:t' applies. Let us clarify these concepts with an example. Consider the *action prefix expression* in Figure 5.2:

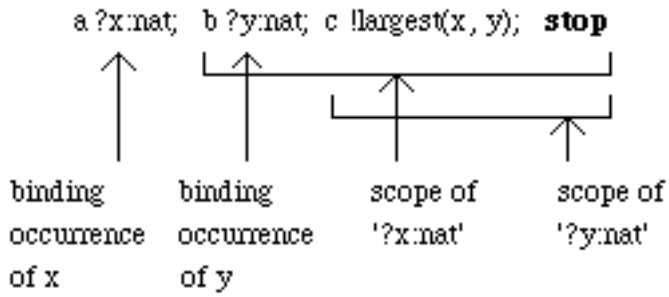


Figure 5.2 - Binding occurrences of variables and associated scopes

The whole expression does not include *free* variables, since the occurrences of 'x' and 'y' in the *value expression* 'largest(x,y)' are *bound*, that is, they fall within the *scopes* associated to their *binding occurrences* in the two *variable declarations*. However, if we consider expression 'b ?y:nat; c !largest(x,y); **stop**' 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. A possible sequence of two transitions for the whole expression is:

a ?x:nat; b ?y:nat; c !largest(x,y); stop —a<0>→
b **?**y:nat; c **!**largest(0,y); **stop** —b<3>→
c **!**largest(0,3); **stop**

Note how variables are replaced by values, in two steps. 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.* In contrast, the values offered via a value declaration in an action prefix expression ('g **!**E; B') may be thought of as *outputs*.

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

The combination of *value declarations* and *variable declarations* in the same *action denotation* 'g α_1 α_2 ... α_n' has the obvious interpretation. For example:

g1 !sap1 ?x:cep-sort !'test'; B(x) — g1<sap1, cep-3, 'test'>→ B(cep-3)

if *cep-3* is a value of sort *cep-sort.*

Interprocess communication

Interprocess communication may occur when two processes composed in parallel are offering the same structured action (same gate, same values), and the gate is one of the interaction gates identified by the parallel operator itself. The semantics of parallel composition is unchanged with respect to basic LOTOS, but now, in light of the discussed input (resp. output) interpretation of *variable* (resp. *value*) *declaration*, value passing is achieved.

Consider this example:

```
        g1 !sap1    ?x:cep-sort   !'test';        g2 !x;    stop
  | |   g1 !sap1    !cep-3        ?y:string;      g3 !y;    stop
```

The two composed expressions (processes) may synchronize, since they are both able to offer, say, action g1<sap1, cep-3, 'test'>. Once the interaction has taken place, the obtained expression is:

```
        g2 !cep-3;  stop
  | |   g3 !'test';  stop
```

where the proper substitutions have been carried out.

Notice that *value declarations* ('output') or *variable declarations* ('input') can match with other *value declarations* or *variable declarations* without constraints, except for the existence of a common value offer. On this basis we can define three types of interaction between two processes, as listed in table 5.1, which is self-explanatory.

Table 5.1 - Types of interaction

Process A	**Process B**	**synchron. condition**	**type of interaction**	**effect**
g **!**E_1	g **!**E_2	value(E_1) = value(E_2)	value matching	synchronization
g **!**E	g **?**x:t	value(E) is of sort t	value passing	after synchronization x = value(E)
g **?**x:t	g **?**y:u	t = u	value generation	after synchronization x = y = v, where v is some value of sort t

As an application of the constructs for value offers and interprocess communication we refine pure LOTOS process Max3 (Figure 2.1), by adding to it the capability of manipulating data values, finally giving a justification for the names chosen for these processes.

```
Specification Max3 [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 *)

type natural is
    sorts   nat
    opns    zero:     → nat
            succ:     nat → nat
            largest: nat, nat → nat
    eqns    ofsort nat
            forall x:nat
                    largest(zero, x) = x
                    largest(x, y) = largest(y, x)
                    largest(succ(x), succ(y)) = succ(largest(x, y))
endtype (* natural *)

behaviour

    hide mid in
            (Max2[in1, in2, mid] |[mid]|  Max2[mid, in3, out])

where

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

endspec (*Max3*)
```

Notice that we have now given a *specification*, not a *process definition.* However, it is perfectly acceptable, and convenient, to keep talking about *process* Max3 as the one defined by such specification. As far as pure synchronization is concerned, this process has exactly the same behaviour as its basic LOTOS version. However, subprocess Max2 is now able to accept any pair of natural numbers at (formal) gates *a* and *b* , and offer the largest between them at gate *c*. Consequently, process Max3 will accept three natural numbers at gates *in1*, *in2*, *in3*, in any order, and offer the largest of them at gate *out*. The keyword **noexit** has to do with successful termination, which is discussed in Section 5.5.

5.2 Conditional constructs

Having added the facilities for defining and describing values in LOTOS we may now express behaviours that depend on conditions on values. Such conditions are expressed as *equations* that relate two *value expressions*: the condition is met if the two expressions evaluate to the same value, in the data type environment of that condition. Also *value expressions* of sort *Bool* of the standard data type *Boolean* are allowed as conditions: an expression *E* of sort *Bool* is used as a shorthand for the equation *E* = true. Below we will refer to the conditions of both kinds as *predicates*. By convention, *predicates* appear enclosed in square brackets.

Selection predicates

An additional feature of *action denotations* is that of the *selection predicates*. An *action denotation* may now terminate with a predicate, containing some of the variables that occur in the *variable declarations* of the *action denotation*. Such predicate is meant to impose restrictions on the values that may be bound to these variables in synchronization events. We illustrate this by two examples.

The only possible transitions of expression 'sap **?**x:nat [x<3]; sap2 **!**x; **stop**' are:

sap **?**x:nat [x<3]; sap2 **!**x; **stop** —sap1<0>→ sap2 **!**0; **stop**
sap **?**x:nat [x<3]; sap2 **!**x; **stop** —sap1<1>→ sap2 **!**1; **stop**
sap **?**x:nat [x<3]; sap2 **!**x; **stop** —sap1<2>→ sap2 **!**2; **stop**

In OSI applications there exist examples where two processes negotiate the value of a parameter in an interaction, each one imposing its own condition. For example, two Transport entities may negotiate the 'quality of service' of the underlying Network service [25]. A simplified example of negotiation is given below.

hide sap **in**
    sap **?**x:nat[x<max]; B1(x)
    |[sap]|
    sap **?**y:nat[y>min]; B2(y)

This process can make internal transitions to any of the processes

hide sap **in**
    B1(n) |[sap]| B2(n)

with 'n' in the open interval (min, max).

Guarded expressions

Any *behaviour expression* may be preceded by a *predicate* and an arrow (that is, by a 'guard'). The interpretation is that if the *predicate* holds, then the behaviour described by the *behaviour expression* is possible, otherwise the whole expression is equivalent with **stop**. A typical scenario is one of a

choice between several guarded expressions.

Examples:

```
      [x>0]→sap !x; P[...](x, ...)
[]    [x≤0]→sap !-x; P[...](x, ...)
```

If x = 1 the above process is equivalent with ' sap **!**1; P[...](1, ...)'. If x = -3, it is equivalent with 'sap **!**3; P[...](-3, ...)'. Case analysis can be specified easily, viz.

```
      [cond1] →process1
[]    [cond2] →process2
...

[]    [condn] →processn
```

The conditions in the guards need not be exclusive, e.g.

```
      [x>0] →process1
[]    [x=5] →process2
[]    [x<9] →process3
```

5.3 Generalized choice

Using the choice-operator '[]' we can only express a finite number of alternatives. In general, we may want to do more. Let B(x) be a *behaviour expression* that may depend on a variable x, say, of sort 'nat'. We can now specify the choice among the processes B(v) for all nat-values v by writing:

choice x:nat **[]** B(x)

Notice that the generalized choice construct allows an alternative representation for the *action prefix* construct, when this includes a *variable declaration* :

a **?** x:t; B(x) is equivalent to **choice** x:t **[]** a **!**x; B(x)

There are more useful applications, however:

choice x:t **[] i**; B(x)

offers a nondeterministic choice between the different instances of B(x), and so does:

choice x:t **[]** a; B(x)

where *a* may be any *action denotation*. More than one variable may be used as an index, so that we may write:

choice x_1:t_1, ..., x_n:t_n **[]** B(x_1, ..., x_n)

Also, sets of *gate identifiers* may be used for indexing, e.g.:

choice g **in** [a_1, ..., a_n] **[]** Process-X[g](...)

In this case a choice is expressed among *n* instances of Process-X: for each one of them formal gate *g* is actualized with a different element of the *gatelist* [a_1, ..., a_n].

5.4 Parametric processes

Full LOTOS offers the possibility to parameterize *process definitions* not only in terms of *formal gates* (as is the case with basic LOTOS) but also in terms of a *parameter list*, which is a list of *variable declarations*: x_1:t_1, ..., x_n:t_n. The syntax for *process definition* in full LOTOS (as anticipated in Figure 5.1) is thus:

process typical_proc [*gate list*] (x_1:t_1, ..., x_n:t_n) : *functionality* := ... **endproc**

Also *specifications* can be parametric, and the syntax is extended analogously. Typically, the variables x_1, ..., x_n occur as free variables in the *behaviour expression* which defines the behaviour of the process or specification. In instantiations, these variables are replaced by *value expressions* (which may include variables): an instantiation of the typical_proc above has the form:

typical_proc [*actual gate list*] (E_1, ..., E_n)

Of course it is required that expressions E_1, ..., E_n match, one-by-one, the sorts of the variables x_1, ..., x_n .This is similar to passing parameters to procedures or functions in traditional programming languages.

Example:

```
process compare[in, out] (min, max: int) : noexit :=
        in ?x:int;
        (   [min < x < max] → out!x; compare [in, out] (min, max)
```

```
        [] [x ≤ min] → out!min; compare [in, out] (x, max)
        [] [x ≥ max] → out!max; compare [in, out] (min, x)
        )
endproc
```

The meaning of the instantiation of a process is the *behaviour expression* that is obtained by substituting the actual parameters for the formal ones, avoiding naming clashes by suitable renaming of binding and bound identifiers, e.g.

compare[one, two](x, 2*x)

is equivalent with

```
one ?y:int;
    (  [x < y < 2*x] → two !y; compare [one, two] (x, 2*x)
    [] [y ≤ x] → two !x; compare [one, two] (y, 2*x)
    [] [y ≥ 2*x] → two !2*x; compare [one, two] (x, y)
    )
```

A more direct way to associate *value expressions* E_1, ..., E_n to the free variables x_1, ..., x_n of a *behaviour expressions* $B(x_1, ..., x_n)$ is offered by the 'let' construct:

let $x_1:t_1=E_1$, ..., $x_n:t_n=E_n$ **in** $B(x_1, ..., x_n)$

5.5 Sequential composition with value passing

Having the possibility to express values it is useful and, sometimes, highly desirable to be able to pass information from the first process in a sequential composition to the second process. In the previously used example:

Connection-Phase[...] >> Data-Phase[...]

we would like to express that the behaviour of the Data-Phase depends on parameters that are established in the Connection-Phase. The Data-Phase is defined as a parametric process, with such parameters as the *expedited-data-option* that indicates whether expedited data can be transmitted or not, and the *quality-of-service* that determines the quality of the connection during the Data-Phase. Therefore, we need a mechanism for passing these parameters from the Connection-Phase to the Data-Phase, at the moment when the former enables the latter. To be able to do such things we must generalize the notion of successful termination, and with that extend the language features with respect to sequential and parallel composition, and add some static constraints to the language.

5.5.1 Successful termination with value offers

In basic LOTOS the **exit** process is used to specify the *successful termination* of a process. We allow now the **exit** process to have a finite list of value expressions added to it. The values expressed are those that are passed on to the subsequent process. Examples:

a **?**x:nat; b **?**y:nat; **exit(**largest(x, y))

tsap **!**cei **?**quality-of-service : quality-parameter-sort **?**expedited-data-option : bool;
exit(quality-of-service, expedited-data-option**)**

The list of the sorts of the values offered at successful termination is called the *functionality* of that termination. The examples above have respective functionalities <*nat*> and < *quality-parameter-sort, bool*>.

In a sequential composition the number and sorts of the values that are passed at the successful termination of the first process must be known. This implies that all the (alternative) successful terminations of the first process must have the same functionality; this functionality is defined as *the functionality* of the first process. Some rules are needed for determining the functionality of *behaviour expressions*, together with some constraints on the ways expressions with different functionalities can be combined. They are listed below. (We write 'func(B)' to denote the functionality of expression B.)

stop

The functionality of processes that do not terminate successfully at all, like **stop**, is indicated with **noexit**.

exit

Simple successful termination without value passing has a functionality that is indicated by the same name: func(**exit**) = **exit**.

Action prefix

The functionality of an expression is clearly unaffected by the prefixing of an *action denotation*: func(*action denotation*; B) = func(B).

Choice

If B_1 and B_2 are processes that both can terminate successfully, then the functionality of the choice expression ' B_1 [] B_2' can only be defined if the restriction is imposed that B_1 and B_2 have the same functionality, in which case this is the functionality of the expression. On the other hand, if func(B_1)=**noexit**, or func(B_2)=**noexit**, then func(B_1 [] B_2) is defined, respectively, as func(B_2) and func(B_1).

For generalized choice the rule is simple: func(**choice** ... [] B') = func(B').

Disabling

This case is analogous to that of (binary) choice:

func(B_1) = func(B_2) = func(B_1[>B_2) =, or
func(B_1) = **noexit**, and func(B_1[>B_2) = func(B_2), or
func(B_2) = **noexit**, and func(B_1[>B_2) = func(B_1).

Parallel composition

In the case of parallel composition, the functionality restrictions/definitions are:

func(B_1) = func(B_2) = func(B_1op B_2), or
func(B_1) = **noexit**, and func(B_1op B_2) = **noexit**, or
func(B_2) = **noexit**, and func(B_1op B_2) = **noexit**.

where 'op' is any parallel operator. Again, if B_1 and B_2 are processes that terminate successfully, then we can compose them in parallel only if they have the same functionality, in which case this becomes the functionality of the parallel expression. In fact, the parallel composition of two processes only terminates successfully if both terminate with the same list of values, in which case the composition terminates also with that list. In this respect, it may be convenient to use the **any**-construct, as a parameter of the **exit** process. It has the format '**any** *sort-identifier*', and can match any value of sort *sort-identifier*'. For instance, **exit**(**any** nat) is a process that can terminate successfully with the offer of any nat-value at the special gate δ. It is clear that 'B_1 op B_2' cannot terminate successfully whenever one of the component processes cannot do so.

Examples:

a **?**x:int; **exit** ||| b **!**'anystring'; **exit**	has functionality **exit**.
a **?**x:int; **exit** ||| b **!**'anystring'; **stop**	has functionality **noexit**.
exit(3**)** ||| **exit(**5**)**	has functionality 'nat', but does not terminate successfully.
exit(3, **any** bool**)** ||| **exit(any** nat, true**)**	has functionality 'nat, bool', and terminates successfully by offering value pair (3, true).
exit(3**)** ||| (a **!**3; **exit** [] a **?**x:nat; **exit**(x))	is not a well-formed LOTOS expression.

The reason why there may exist a process B that cannot terminate successfully, while func(B) is different from **noexit**, is that functionality and actual termination are two different things. The former is a sort of static typing mechanism, which is only meant to guarantee the predictability of the list of sorts offered at successful termination, *in case that such termination* occur. The actual occurrence of a successful termination, in general, cannot be decided statically, nor dynamically, since this problem is equivalent to the well-known 'Halting Problem' for Turing machines [23]. The functionality typing scheme helps in avoiding constructions however, of which the absence of successful terminations can be decided statically.

Process definitions and instantiations

Both a *specification* and a *process definition* include in their headers a parameter indicating the functionality of that *specification* or *process definition*, (see Figure 5.1), which is defined as the functionality of the *behaviour expression* of that *specification* or *process definition*. In this functionality parameter a functionality ' t_1, ..., t_n' is combined with the keyword **exit**, so that the three possible formats of this parameter are:

noexit
exit
exit(t_1, ..., t_n)

where t_1, ..., t_n is a list of sorts. On the other hand, in *process instantiations* the functionality is not given explicitly; it is defined however, as that of the associated *process definition.*

Examples:

```
process P[a]: exit(nat, bool) :=
    a ?x:nat ?y:nat;
        (i; exit(x, true)  []  i; exit(y, false))
endproc

process Q[a, b]: exit :=
    a ?x:nat;
        (b !x; exit  []  i; Q[a, b])
endproc
```

```
process R[a, b]: noexit :=
    a ?x:nat ?y:nat;
        (b !x; stop  []  b !y; stop)
endproc
```

5.5.2 Accepting values from the enabling process

Once a process B_1 with the desired functionality, say **exit(**t_1,...,t_n**)**, has been defined, its sequential composition with another process B_2 has the following form:

$$B_1 \text{ >> } \textbf{accept } x_1{:}t_1, \ ..., \ x_n{:}t_n \textbf{ in } B_2$$

Here x_1, ..., x_n are the variables used in B_2 for the *n* values passed at the successful termination of B_1. The obvious requirement is that the functionality of B_1be matched by the list of sorts t_1,...,t_n after the **accept** keyword. It is also clear that the functionality of the whole construct is defined as the functionality of B_2.

The example quoted at the beginning of this section can now be correctly specified as follows:

```
        Connection-Phase[...](...)
>>      accept    quality-of-service : quality-parameter-sort
                  expedited-data-option : bool
        in
        Data-Phase[...]( quality-of-service, expedited-data-option)
```

As a concluding remark we would like to observe that the value passing in *sequential* composition can be considered as a special case of the value passing in *parallel* composition. We may indeed imagine that the enabling process synchronizes its last action (successful termination) with an "accepting" action implicitly prefixed to the enabled process, and that data is passed by this interaction. We should also regard this communication as private to the enabling and the enabled processes, that is, hidden to other processes. In fact, the operational semantics of the enabling operator in full LOTOS [27] exactly reflects this point of view.

6. An example of constraint-oriented specification

Structured programming, in the context of traditional programming languages, allows the programmer to take a "divide-and-conquer" approach and partition his/her task into smaller sub-tasks

to be handled separately. Similarly, the constraint-oriented specification style is a "divide-and-conquer" approach by which the LOTOS user conceives his/her specification as a collection of clearly separated, small pieces (processes), each one expressing few constraints on the temporal ordering of the system events. All these pieces are then composed via the parallel operator (with synchronization), which acts as a logical conjunction (AND) of all the constraints. As a consequence, any action occurring at some synchronization gate is simultaneously subject to all the constraints expressed by the processes sharing that gate. We gave a trivial example of a composition of constraints in Section 2.4, in discussing the general parallel operator. We provide here a more complex example of the constraint-oriented specification style, written by Chan and Turner [13].

In the 'Daemon Game' a player may start a new game, probe the system for randomly incrementing or decrementing his score, ask for the score, and quit the game. The system may support an unlimited number of players, and every player is required to specify his own 'id' every time he/she interacts with the system. In the specification all users interact with the system via a unique gate (usr), but they are distinguished by their respective id's. All the observable actions have the unique form:

usr <id, sig>

where id, of id_sort, is the identifier of some player and sig, of sig_sort, is a signal in the set {newgame, endgame, probe, win, lose, result} $\cup$ Scores (a set isomorphic to the set of integers). The specification has been conceived as the composition of two main concerns, embodied by processes Login_Check and Sessions. The first process does not impose any temporal constraint on actions, and is only sensitive to the actions where the signal is either 'newgame' or 'endgame', since its only concern is to properly maintain the set of user id's (Used_Id_Set). Any other action is simply absorbed. The second process (Sessions) is the interleaved composition of an infinite number of sessions, where an individual session is described by the *behaviour expression* at the left of the '|||' operator. A session is opened and closed when a player gives, respectively, the 'newgame' and the 'endgame' signals; the actual game is described by process 'Game'. Any individual instance of process Game is only concerned with actions characterized by a fixed value of parameter 'id', in order to properly maintain and display the score of a specific player. Processes Sessions and Game do impose some temporal constraints to the actions: for instance, winning or losing must always be preceded by probing.

```
(*_____________________________________________________________

                This is a slightly modified version of
          the Daemon Game specification by W. F. Chan and K. J. Turner [13]
_____________________________________________________________*)

specification Daemon_Game [usr] : noexit

library
```

Boolean, Set, NaturalNumber
endlib

behaviour
Login_Check [usr] (empty) (* no users initially *)
||
Sessions [usr]

where

type Integer **is**

sorts int

opns 0 : → int
inc, dec : int → int

eqns **forall** n : int
ofsort int
inc(dec(n)) = n;
dec(inc(n)) = n
endtype

type Signal **is**

sorts sig_sort

opns newgame, endgame, probe, win, lose, result : → sig_sort
score : int → sig_sort

endtype

type Identifier **is** NaturalNumber
renamedby
sortnames id-sort **for** nat
endtype

type Identifier_set **is** Set
actualizedby Identifier **using**

sortnames

id_sort	**for**	elem
id_set_sort	**for**	set

endtype

(*_____

The following process ensures that users are given different identifiers on logging in. A set of identifiers in use is maintained

_____*)

process Login_Check [usr] (used_id_set : id_set_sort) : **noexit :=**

usr **?**id : id_sort **!**newgame [id NotIn used_id_set];
Login_Check [usr] (insert (id, used_id_set))

[] usr **?**id : id_sort **!**endgame [id IsIn used_id_set];
Login_Check [usr] (remove (id, used_id_set))

[] usr **?**id : id_sort **!**probe [id IsIn used_id_set];
Login_Check [usr] (used_id_set)

[] usr **?**id : id_sort **!**win [id IsIn used_id_set];
Login_Check [usr] (used_id_set)

[] usr **?**id : id_sort **!**lose [id IsIn used_id_set];
Login_Check [usr] (used_id_set)

[] usr **?**id : id_sort **!**result [id IsIn used_id_set];
Login_Check [usr] (used_id_set)

[] **choice** x:int [] usr **?**id : id_sort **?**score(x) [id IsIn used_id_set];
Login_Check [usr] (used_id_set)

endproc

(*_____

The following process specifies the permitted sequences of interactions between the users and the game as an infinite set of processes in parallel, one for the independent behaviour of each user session

_____*)

```
    process Sessions [usr] : noexit :=

        (usr ?id : id_sort !newgame;
            (    Game [usr] (id, 0)          (* score initially zero *)
            [>   usr !id !endgame;  exit
            )
        )
        |||
        Sessions [usr]

    where


(*_____________________________________________________________________

    The following process specifies the behaviour of a logged-in user.
(*_____________________________________________________________________

        process Game [usr] (id : id_sort, total : int) : noexit :=

            usr !id !probe;
            (    i;
                 usr !id !win;
                 Game [usr] (id, inc(total))
            []   i;
                 usr !id !lose;
                 Game [usr] (id, dec(total))
            )
        []
            usr !id !result;
            usr !id !score (total);
            Game [usr] (id, total)

        endproc

    endproc (* Sessions *)

endspec
```

7. Conclusions

We have presented the specification language LOTOS. 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 examples given, that once the user has achieved some familiarity with the operators of the language, he/she can specify systems in a natural way which reflects quite directly the way the system's structure and behaviour are conceived at the intuitive level. The specifier, in general, does not feel forced to express unnecessary details with respect to his/her *abstract* view of the processes being specified.

LOTOS has the merit (and takes the risks) of being based on relatively new and powerful theories, which so far have mainly been confined to academic environments. The wide exposure that the language is currently undergoing by its application to the specification of OSI protocols and services [42] is a valuable test for the practical applicability of those theories. The first results are encouraging: the LOTOS specifications that have been produced so far (e.g. [1, 3, 12, 17, 40, 41, 43, 44, 45, 47] and many others), indicate that such quite complex systems can be specified with an intuitively appealing structure, and be relatively concise (when compared with other formal description techniques). The conciseness and readability could be increased even further if good notational facilities are developed for the specification of data types, which now in many cases are a substantial part of a specification. Work in this direction is under development [29].

An important problem to be addressed in producing 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 specified both as processes and as data types. On one hand we may rule out this problem as a mere matter of taste and style. On the other hand 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 an area where interesting developments are possible.

An important element in the eventual success of LOTOS will be the adequate training of those that are to apply it in practice [28]. The current trend of the growing importance of formal methods in computer science and telecommunications is not yet reflected in the education of many of its practitioners. This requires a coordinated effort in the development of courses and teaching material, to which this tutorial is a contribution. However, as time passes this problem will disappear. In the longer run the prospects for techniques like LOTOS are bright. Its link to a sound formal theory, and the ongoing efforts to build tools for its application to the design, analysis and testing of open distributed systems [6, 18, 31, 46] offer hopes of a future in which such systems can be developed faster and with more reliability than today.

Acknowledgements

The authors gratefully acknowledge the direct and indirect contributions to this tutorial from their fellow LOTOS-eaters, with whom they have worked together during several years developing the language, applying it, and building tools for it. This work has taken place in several environments, of which the most important are COST11 bis/TOS, ISO/TC97/SC21/WG1/FDT Subgroup C, the ESPRIT/SEDOS Project, and the IPS Group at the University of Twente (The Netherlands). We would like to mention specifically Luigi Logrippo, Jan de Meer, Elie Najm, Giuseppe Scollo, Alastair Tocher and Chris Vissers. We would like to thank Ken Turner and W. F. Chan for their example of the Daemon Game in LOTOS, which we have included in the paper. The first author also wishes to thank Rocco De Nicola for useful discussions during the writing of parts of this paper. This work was partially supported by the CEC as part of the ESPRIT/SEDOS project.

References

[1] I. Ajubi, "Draft Formal Specification of the OSI Connection-Oriented Session Protocol in LOTOS", ISO/TC 97/SC 21 N. 1486, February 1986.

[2] J. Bergstra, J. W. Klop, "Process Algebra for Synchronous Communication", Information and Control 60, pp. 109-137, 1984.

[3] F. Biemans, P. Blonk, "On the Formal Specification and Verification of CIM Architectures Using LOTOS", Computers in Industry 7, pp. 491-504, 1986.

[4] T. Bolognesi, S. A. Smolka, "Fundamental Results for the Verification of Observational Equivalence: a Survey", proceedings of the IFIP Seventh International Symposium on Protocol Specification, Testing, and Verification, H. Rudin and C. West (eds.), North-Holland, 1987.

[5] G. Boudol, "Notes on algebraic calculi of processes, Rapport de Recherche No. 395, INRIA, Sophia Antipolis, April 1985.

[6] J. P. Briand, M. C. Fehri, L. Logrippo, A. Obaid, "Executing LOTOS Specifications", in: B. Sarikaya, G. V. Bochmann (eds.), Proceedings of IFIP Workshop 'Protocol Specification, Testing, and Verification VI', pp. 73-84, North-Holland, Amsterdam, 1987.

[7] E. Brinksma, "A Tutorial on LOTOS", in: M. Diaz (ed.), Proceedings of IFIP Workshop 'Protocol Specification, Testing, and Verification V', pp. 171-194, North-Holland, Amsterdam, 1986.

[8] E. Brinksma, "On the Existence of Canonical Testers", Memorandum INF-87-5, University of Twente, January 1987.

[9] E. Brinksma, G. Scollo, "Formal Notions of Implementation and Conformance in LOTOS", Memorandum INF-87-13, University of Twente, November 1986.

[10] E. Brinksma, G. Scollo, C. Steenbergen, "LOTOS specifications, their implementations, and their tests", in: B. Sarikaya, G. V. Bochmann (eds.), Proceedings of IFIP Workshop 'Protocol Specification, Testing, and Verification VI', pp. 349-360, North-Holland, Amsterdam, 1987.

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

[12] V. Carchiolo, A. Faro, O. Mirabella, G. Pappalardo, G. Scollo, "A LOTOS Specification of the PROWAY Highway Service", IEEE Trans. on Computers, Vol. C-35, No. 11, pp. 949, 968, November 1986.

[13] W. F. Chan, K. Turner, "The Daemon Game in LOTOS", in: ESTELLE, LOTOS, SDL Draft Examples, Joint Meeting ISO/CCITT (ISO/TC97/SC21/WG1/FDT - CCITT X/3), Turin,

December 15-19, 1986.

[14] R. De Nicola, "Extensional Equivalences for Transition Systems", Acta Informatica, Vol. 24, pp. 211-237, 1987.

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

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

[17] ESPRIT/PANGLOSS, Parallel Architectures Networking Gateways Linking OSI Systems. ESPRIT Project 890.

[18] ESPRIT/SEDOS, Software Environment for the Design of Open Distributed Systems. ESPRIT Project ST410.

[19] J. A. Goguen, J. W. Thatcher, 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.

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

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

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

[23] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley 1979.

[24] ISO - Information Processing Systems - "Basic Reference Model for Open Systems Interconnection", IS 7498, 1983.

[25] ISO - Information Processing Systems - Open Systems Interconnection - "Connection Oriented Transport Protocol Specification", IS 8073, 1986.

[26] ISO - Information Processing Systems - Open Systems Interconnection - "ESTELLE - A Formal Description Technique Based on an Extended State Transition Model", DIS 9074, 1987.

[27] ISO - Information Processing Systems - Open Systems Interconnection - "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", DIS

8807, 1987.

[28] ISO/TC 97/SC 21 N. 1534, "Guidelines for the Application of FDT to OSI Protocols and Services", 1986.

[29] ISO/TC 97/SC 21 N. 1540, "Potential Enhancements to LOTOS", 1986.

[30] K. G. Larsen, "Context Dependent Bisimulations Between Processes", Ph.D. Thesis, University of Edinburgh, Dept. of Computer Science, May 1986.

[31] A. Marshall, "LOTOS Tools Development", C3 Progress Report, ESPRIT/SEDOS/C3/WP/20/IK, STC Tech. Ltd., Newcastle-under-Lyme, England, January 1987.

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

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

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

[35] R. Milner, "Calculi for Synchrony and Asynchrony, Theor. Comp. Science 25, pp.267-310, 1983.

[36] E. Najm, "A verification oriented specification in Lotos of the Transport Protocol", proceedings of the IFIP Seventh International Symposium on Protocol Specification, Testing, and Verification, H. Rudin and C. West (eds.), North-Holland, 1987.

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

[38] G. Plotkin, "A Structural Approach to Operational Semantics", Lecture Notes, Aarhus University, 1981.

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

[40] J. Quemada, Data Link Service LOTOS Specification, SEDOS/C1/6&7/M, December 1986.

[41] G. Scollo, "Formal Description in LOTOS of the OSI Transport Protocol (Version 9), ESPRIT/SEDOS/C1/WP/41/T, March 1987.

[42] G. Scollo, F. Minissale, "On the Specification in LOTOS of OSI Protocols", in: G. Bucci, G. Valle (eds.), Computing '85, Proc. 8th ACM European Conf. ICS '85, Florence, Italy, March 1985, pp. 197-206, North-Holland, 1985.

[43] G. Scollo, G. Pappalardo, L. Logrippo, E. Brinksma, "The OSI Transport Service and its Formal Description in LOTOS", in: L. Csaba, K. Tarnay, T. Szentivanyi (eds.), Computer Network Usage: Recent Experiences, pp. 465-488, North-Holland, Amsterdam, 1986.

[44] A. J. Tocher, "OSI Transport Service: A Constraint-Oriented Specification in LOTOS", ESPRIT/SEDOS/C1/WP/25/IK, ICL, Kidsgrove, August 1986.

[45] K. J. Turner, "OSI connection-oriented network service: a constraint-oriented specification in extended LOTOS (draft 4), SEDOS/C1/WP/15/IK, ICL Kidsgrove, England, May 1986.

[46] P. Van Eijk, "Tools for the Specification Language LOTOS", University of Twente, November 1986 (submitted for publication).

[47] M. Van Sinderen, "Draft formal specification of the OSI connection-oriented session service in LOTOS (version 5)", SEDOS/C1/WP/35/T, November 1986.