An Introduction to LOTOS: Learning by Examples[1]

L. Logrippo, M. Faci, M. Haj-Hussein

University of Ottawa
Protocols Research Group
Department of Computer Science
Ottawa, Ontario, Canada K1N 6N5
E-mail: lmlsl@uottawa.bitnet

Abstract. An informal, design-oriented introduction to the specification language for distributed systems LOTOS is presented. Examples based on variations of the well-known producer-consumer problem are used to illustrate the different aspects of the language.

Keywords: Concurrent languages, Formal Description Techniques, Open Systems Interconnection, specification languages, LOTOS.

1. Introduction

1.1 Motivation

This paper should be seen as a companion to the small number of LOTOS tutorials existing today. Bolognesi and Brinksma's paper [BB 87], the only one in print in English we are aware of at the time of writing this paper, concentrates on the theoretical foundations of the language. We intentionally avoid discussing certain important aspects of the language that are covered in [BB 87], such as formal definitions of LOTOS concepts, inference rules and behavioral equivalences. While we consider this tutorial self-contained, the novice reader is encouraged to first read this tutorial and then consult [BB 87] to obtain a broader view of LOTOS. Also, the descripion of ACT ONE, the data part of LOTOS, is left to another paper appearing in this issue [MRV 91]. We introduce LOTOS by using examples, with the aim of showing readers that the language, from a

1. Corrected version of the paper appeared in Computer Networks and ISDN Systems 23 (1992) 325-342

design point of view, offers the power of expressing abstract ideas in precise terms.

1.2 The Origin of LOTOS

LOTOS is a by-product of the effort of standardization of the Open Systems Interconnection (OSI) within the International Organization for Standardization (ISO). As this work started in the late seventies, it was realized that for OSI standards to be effective, they had to be precisely described. The term *Formal Description Techniques* was then coined (FDTs), to refer to techniques for the exact specification of protocols and services.

Some of the main desirable characteristics of the FDTs were recognized to be: abstractness, implementation-independence, formal semantics and support of verification methods. It was soon realized that no suitable FDT existed yet, so a committee was set up in 1979 to work towards the definition of such an FDT. The committee became known as the FDT Ad-hoc Group of ISO/TC 97/SC 16/WG 1 (SC 16, to become later SC21, being the Subcommittee in charge of standardizing the higher layers of the OSI, and WG 1 being the Working Group in charge of developing the basic architectural concepts of the OSI).

The first Rapporteur of the FDT group was John Day, who soon resigned in favor of Chris Vissers. After some initial meetings, where the existing methods were evaluated and directions charted, the FDT group found it impossible to agree on a common conceptual model. Two main subgroups were established: Subgroup B, which was to work on an Extended Finite State Machine Model, and Subgroup C, which was to work on a Temporal Ordering Technique. Subgroup B eventually produced the Estelle standard [ISO3 89], while Subgroup C produced LOTOS [ISO1 89]. The first Chairman of Subgroup C was Chris Vissers.

The year 1983 saw several important developments. Ed Brinksma, who had joined the group the previous year, became Chairman of Subgroup C. He maintained the technical direction of the Subgroup until the end, and he was the editor of the final International Standard as well as of the several drafts. The direction of work became firmly established in the course that led to LOTOS: the language was to be based mostly on Milner's Calculus of Communicating Systems (CCS) [Mil 89], although Hoare's Communicating Sequential Processes (CSP) [Hoa 85] was also quite influential. In the same year, the language acquired its name, which is an acronym for Language Of Temporal Ordering Specifications.

In 1984, it was decided that Ehrig and Mahr's ACT ONE [EM 85] was to be used as the basis for the definition of data types. The first version of the language came to light in 1985, and was published as ISO Draft Proposal 8807. The semantics of LOTOS were defined on the basis of a simpler language, called CCS*, which extended CCS.

The following months brought about considerable discussion on possible improvements of the language. A. Tocher, then a student of C.A.R. Hoare, proposed the introduction of CSP-like multi-way synchronization in the language. To illustrate what improvements in expressiveness this would bring about, he demonstrated the concept of *constraint-oriented style* [VSVB 91] [Tur 88a][FLS 90] with application in the specification of a simplified Transport Service (this constituted the first draft of what has become the current Draft Technical Report on the LOTOS specification of that service [ISO2 89]). Bolognesi, DeNicola, and Latella showed how CCS* could be disposed of, and a more natural, one-level definition of the language could be devised.

These discussions led to the language as it is known today, of which a first version appeared in the Second Draft Proposal of 1986. Readers of early LOTOS papers should beware that those papers relate to a language (the one of the 1985 Draft Proposal) that has some subtle, but important differences with respect to current LOTOS. In 1986, the first interpreter that supported the full language of the 1985 Draft Proposal, was produced at the University of Ottawa [LOBF 88]. What followed was mostly refinement and completion. The static semantics, which must relate concepts of the control and the data parts, required a particularly elaborate effort.

The language was substantially complete in 1987, but it had to wait until 1989 to become an International Standard [ISO1 89]. In 1988, with the end of the SEDOS project, a number of tools (including an interpreter) and technical reports on the language, as well as a book [VVD 89], became available. Current work concentrates on the application of the language, and on the development of the theory and tools. In order to allow this work to proceed on a firm basis, the ISO has decided to temporarily disallow changes and enhancements, however at the time of writing (early 1991) modifications to improve the user-friendliness of ACT ONE are being envisaged. Work on the development of a graphic presentation for LOTOS started as a joint project of ISO and CCITT in 1988, and is now almost complete (responsible for this project are Elie Najm and Paul Tilanus).

As is common in the case of standards, the membership of the LOTOS group varied from meeting to meeting, and several people who did not attend meetings regularly also gave valuable contributions.

Another brief account of the development of LOTOS is given in [VCA 89].

1.3 LOTOS Principles

Some of the principles that have inspired the design of LOTOS are:

1. *Complementary formalisms for 'data' and 'control'*. LOTOS designers felt that no single existing formalism was general enough to express conveniently both the control component

and the data component of a specification. Accordingly, the language was conceived as the union of two formalisms: ACT ONE for the data part, and the CCS/CSP-based language discussed in this paper for the control part.

2. *Formal definition:* Formally defined syntax, static semantics, and dynamic semantics. In particular, the static semantics are defined by an attributed grammar [ISO1 89], and the dynamic semantics are described operationally in terms of inference rules [BB 87].
3. *Process algebra:* Following Milner's ideas, the operational semantics are defined in such a way that it is possible to prove a rich set of algebraic equivalence properties, based on several types of equivalence relations. These properties can be used in order to prove equivalence or correctness of specifications, as well as to transform the structure of a specification. Several examples of these properties are given in this paper.
4. *Interleaving concurrency:* Events are considered to be atomic, and thus the parallel execution of two events *a* and *b* is defined as a situation of choice, where *a* can occur before *b,* or vice versa. Therefore, any LOTOS behavior expression can be rewritten as an expression consisting of a choice between behavior expressions, each prefixed by a single action (i.e., expansion theorem [Mil 80], [Mil 89]). This principle is used in our paper, where we explain the operators by examples for which we show the expansion.
5. *Executability:* Because LOTOS semantics are defined operationally, it is possible to implement these semantics in an interpreter, which for a behavior expression can enumerate the set of possible next actions [LOBF 88][V 88], and the behavior expressions resulting by the execution of each one of them (this is another application of the expansion theorem). Although this set can be infinite, in many cases it can still be described in finite terms. This means that LOTOS specifications can be written, without difficulty, in such a way as to be interpretable, or even, with some user-supplied information, to be translatable into a program [MM 89]. Such specifications can be taken to be *fast prototypes* of the entity specified.
6. *Modularity and module reusability:* LOTOS favors stepwise decomposition of processes. By using parameterization, these processes become reusable.

Much of the power of LOTOS is the result of the power of the parallel composition operator, and of the concept of process *rendezvous*, which is called process *synchronization*. Their semantics are quite different from those found in most common programming languages. The following are some of the salient characteristics of these concepts:

1. *Multiway synchronization:* While much of LOTOS semantics are based on CCS, the multiway synchronization concept was borrowed from Hoare's CSP. In order for synchronization to occur, a number of processes may have to participate, as will be described below.
2. *Symmetric synchronization:* As mentioned above, all processes that participate in a synchronization cooperate in it equally, in particular there is no concept of a process

initiating the synchronization and others responding. For example, while it is often thought that in an output operation a producer process transfers information to a passive external environment, in LOTOS one says that both the producer process and the environment participate in establishing the value being output. There is no directionality in this concept, although some processes may have more information on the value to be established than others.

3. *Anonymous synchronization:* A process that is ready for a synchronization proposes the synchronization to its environment, without being able to direct its proposal to a specific process. It is the structure of the system where the process is embedded that decides which processes will have to participate for a synchronization to occur. Process identification can be specified as an exchange of appropriate values (Section 3.2).
4. *Nondeterministic synchronization.* Often more than one synchronization is possible. One only will be executed according to a nondeterministic choice (Sections 2.4 and 2.6).

A *behavior expression* represents a state of a process. A predefined set of operators is used to combine actions and behavior expressions to form other behavior expressions. There are two predefined behavior expressions, ***stop***, which denotes *deadlock* (or *inaction*) and ***exit***, which denotes *successful termination*. LOTOS *process definitions* are named behavior expressions, similar to procedures in a programming language. Process instantiation is similar to procedure call.

The behavior expression of a process determines which actions (or events) are possible as next actions of the process. There are actions that a process can execute independently, these are represented by the internal action ***i***. And there are actions that need synchronization with the *environment* in order to be executed. These are *offered* at synchronization points called *gates*. The environment of a process consists of other processes, or some external (i.e. non-LOTOS) world that can be a human observer. When an action is executed, the behavior expression of the process is transformed into another behavior expression. It is the inference rules of the dynamic semantics that determine which actions can be offered and executed by a process and how behavior expressions are transformed by effect of the execution of actions. For instance, we shall see that the behavior expression *a; B* evolves into behavior expression *B* after executing action *a*. Similarly, *a; A [] b; B* evolves into behavior expression *A* after executing action *a*.

1.4 Notation

Throughout this paper, we use upper case letters (*A, B, C,* ....) to denote behavior expressions and lower case letters (*a, b, c,* ...) to denote actions. Process names are written in ***Bold_italics*** with capital initials, LOTOS keywords are written in **bold**.

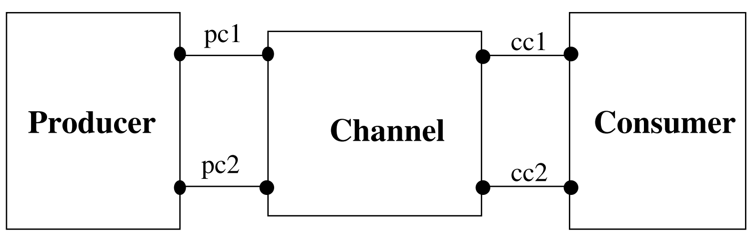
2. Basic LOTOS

Basic LOTOS, also called *pure* LOTOS, is a subset of the language where process synchronization is achieved, but with no data exchange (in other words, no data types are used). In basic LOTOS, an action denotation is simply the identification of a gate.

2.1 The Example

The example that we have chosen for this tutorial is the classic problem of a producer and a consumer communicating by means of a channel. In order to explore all LOTOS operators, we will use the same problem with different formulations to suit our illustrative needs.

Formulation 1: A LOTOS specification for the producer-consumer problem with a two-place channel is to be provided. The channel, which handles two types of messages, is FIFO (First In, First Out); it may not lose or reorder messages. The producer must produce exactly two elements and then terminate. Similarly, the consumer must consume both elements and then terminate. The channel synchronizes with the consumer only after it has finished interacting with the producer.



2. 2 Structure of the Specification

One way to structure the producer-consumer specification is to decompose the problem into three processes, specify each process separately and then compose them to obtain the final solution. At this point, the reader should not be concerned with the question of how the three processes fit together. Rather, it is more important to understand how each process behaves independently with respect to its own environment. Section 2.7 deals with process composition.

2.3 The Action Prefix Operator

The *action prefix* operator, written as a semi-colon **;**, expresses sequential composition of actions. This operator is used to sequentially order actions. For example, *a; B* denotes a behavior where action *a* must be executed before the behavior expression *B*. The producer, as a separate entity, can be seen as a black box, which synchronizes with its environment through two synchronization points, *pc1* and *pc2*. It performs three actions: it produces two elements and then ***exit***s. The process

Producer can then be specified as follows:

> **process** ***Producer*** [pc1, pc2] : **exit** :=
> pc1; pc2; **exit**
> **endproc**

Producer synchronizes with its *environment* through two gates *pc1* and *pc2*, which are formal gates. Actual gates must be specified at instantiation time (Section 2.10). The keyword ***exit***, in the process definition, indicates that the process is able to execute an ***exit*** at the end, i.e. to successfully terminate. In LOTOS terminology, we say that this process has *functionality* ***exit***. The functionality of a process can be either ***exit*** if the process is able to successfully terminate or ***noexit*** otherwise [BB 87]. Successful termination means that execution can continue to other processes, as will be seen in Section 2.5.

Similarly, the behavior of ***Consumer*** can be written as:

> **process** ***Consumer*** [cc1, cc2] : **exit** :=
> cc1; cc2; **exit**
> **endproc**

Finally, the behavior of the ***Channel***, which synchronizes on its left with ***Producer*** and then synchronizes on its right with ***Consumer***, is given by the following process:

> **process** ***Channel*** [pc1, pc2, cc1, cc2] : **exit** :=
> pc1; pc2; cc1; cc2; **exit**
> **endproc**

At this point we are ready to compose all three processes to obtain the complete system. However, this requires the knowledge of some other operators, called *parallel composition operators,* which are suited for exactly that purpose. The composition will be given in Section 2.7.

2.4 Choice Operator

To illustrate the use of the *choice* operator, let us reformulate the problem.

Formulation 2: Modify the specification so that the channel may deliver the first element, produced by the producer, to the consumer before the second element is put in the channel.

To satisfy this new requirement, we need the choice operator [], which denotes the choice between two or more alternative behaviors. In this case, the ***Channel*** process must first synchronize with the ***Producer*** on gate *pc1* and then either synchronize with the ***Producer*** a second time, on gate *pc2*, or synchronize with the ***Consumer***, on gate *cc1*. Once the choice between *pc2* and *cc1* is made, the other alternative is ignored. In other words, if *pc2* is chosen, then the behavior of ***Channel*** becomes *cc1; cc2;* ***exit*** otherwise, the behavior of the channel becomes *pc2; cc2;* ***exit***. The choice operator is commutative and associative. So, *A [] B* is equivalent to *B [] A* and *A [] B [] C* is equivalent to both *(A [] B) [] C* and *A [] (B [] C)*.

In this paper, the term behavior *A* is *equivalent to* behavior *B* means that the sequences of actions which behavior *A* is capable of offering, according to an outside observer, are the same as those which behavior *B* is capable of offering. For example, *a;* ***i****; b;* ***stop*** is equivalent to *a; b;* ***stop***. For a formal discussion of *observational equivalence*, see [BB 87][Mil 89].

```
process Channel [ pc1, pc2, cc1, cc2] : exit :=
    pc1;
    (
        pc2;   cc1;   cc2;   exit
    []
        cc1;   pc2;   cc2;   exit
    )
endproc
```

Note that the choice can be nondeterministic. For example, compare the behaviors of three vending machines:

insert_quarter; get_coffee; **stop** [] insert_dime; get_milk; **stop** (* 1 *)

insert_quarter; get_coffee; **stop** [] insert_quarter; get_milk; **stop** (* 2 *)

insert_quarter; (get_coffee; **stop** [] get_milk; **stop**) (* 3 *)

Machine 1 offers the client (the environment) a choice between inserting a quarter and inserting a dime. If the environment offers a quarter, the behavior of the machine evolves to *get_coffee; stop*. If it offers a dime it evolves to *get_milk; stop*. Machine 2 accepts quarters only, and after synchronization with the environment (i.e., once the client inserts a quarter), the behavior of the machine can evolve to either *get_coffee; stop* or *get_milk; stop* depending on a *nondeterministic choice* (in other words, once the client inserts a quarter the choice to synchronize on either *get_coffee* or *get_milk* would no longer exist). Machine 3 accepts quarters only as well, but it is more democratic. Once the client inserts a quarter, the behavior of the machine evolves to *(get_coffee;* ***stop*** *[] get_milk;* ***stop****)*, meaning that the client can still choose between coffee and milk. Another way to express nondeterminism in LOTOS is by using the internal action, as shown in Section 2.6.

Nondeterminism is a powerful abstraction mechanism, since it allows to withdraw specification of details that are not relevant at a given level of abstraction. An implementation of (2) will have to decide under what conditions each of the two choices is taken, however for specification purposes such a decision may be unessential.

2.5 Enable Operator

The LOTOS *enable* operator **>>** has a similar function as the action prefix operator, which expresses the sequential composition of an action with a behavior expression. The **>>** is used to express the sequential composition of two behavior expressions. For example, if *P1* and *P2* are two processes, *P1 >> P2* is read *P1 enables* P2. Process *P1* must terminate successfully in order for *P2* to be enabled. This is the only condition under which process *P2* is enabled. Execution of an ***exit*** in *P1* results in an action on a special gate δ. The enable causes δ to become an ***i***, and execution to continue with *P2*. For example,

a ; b ; **exit** >> c ; **stop**

is equivalent to *a; b;* ***i****; c;* ***stop***, i.e., *a; b; c;* ***stop***, whereas,

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

is equivalent to *a; b;* ***stop***. The expression on the right hand side of the enable operator cannot be executed, because the expression on the left-hand side cannot terminate successfully.

The process ***Channel*** was written using the action prefix and choice operators only. Using the **>>** operator, we can replace (for the sake of illustration) the process ***Channel*** with a process which exhibits an equivalent behavior. The following informal solution results:

```
process Channel [ . . . ] : exit :=
    ( Get first element;
        (     Get second element;  Put first element; exit
              []
              Put first element ; Get second element; exit
        )
    )
    >>     Put second element
endproc
```

As explained previously, the first synchronization must occur at gate *pc1*. After synchronizing on *pc1*, the ***Channel*** offers to synchronize on *pc2* and *cc1* in any order. Finally, the ***channel*** must synchronize on *cc2*. Translated into LOTOS, we have:

```
process Channel [ pc1, pc2, cc1, cc2] : exit :=
    pc1;
    (
        pc2;   cc1;   exit
        []
        cc1;   pc2;   exit
    )
    >>    cc2;   exit
endproc
```

Note that both branches of the choice have an ***exit*** as the last action. Therefore, if either alternative reaches the ***exit***, the behavior *cc2; **exit*** becomes enabled.

2.6 Internal Action

Let us add new constraints to the requirements of the producer-consumer problem.

Formulation 3: Modify the specification so that the channel may lose either or both elements. Losing an element can be modelled by an internal action of the channel. Once an element is put in the channel, it may either be consumed by the consumer or lost as a result of an unexplained internal action of the channel. Note that the internal action ***i*** is not controlled by the environment and therefore, in conjunction with the *[]* operator, it represents a nondeterministic choice. Informally, the process ***Channel*** becomes:

```
process Channel [ . . . ] : exit :=
    ( Get the first element;
        (   Get the second element ;    Put the first element;     exit   (* 1 *)
            []
            Get the second element;     Lose the first element;    exit   (* 2 *)
            []
            Put the first element;      Get the second element;    exit   (* 3 *)
            []
            Lose the first element;     Get the second element;    exit   (* 4  *)
        )
    )
    >>    ( Put the second element; exit  [] Lose the second element; exit )
endproc
```

Note that from an observational point of view, there is no difference between 2 and 4. This is an application of an equivalence law by which *a; **i**; **exit** [] **i**; a; **exit*** is equivalent to ***i****; a;* ***exit***.

Therefore, these two alternatives are merged into a single behavior expression. In LOTOS, we have:

```
process Channel [pc1, pc2, cc1, cc2] : exit :=
    pc1; ( pc2 ; cc1; exit [] cc1; pc2; exit    []    i;   pc2; exit   )
         >> ( cc2; exit  []  i;   exit)
endproc
```

However, after changing the ***Channel*** specification, the ***Consumer*** is no longer correct, because the ***Channel*** is now ready to synchronize first on *cc1*, if the first element is not lost, or on *cc2*, if the first element is lost. Therefore, the consumer must be specified to synchronize on the following sequences of actions: *cc1, cc2* then ***exit***, *cc1* then ***exit***, *cc2* then ***exit***, or simply ***exit*** when both elements are lost. In LOTOS:

```
process Consumer [ cc1, cc2] : exit :=
    cc1; (cc2; exit [] exit)
    []
    cc2;    exit
    []
    exit
endproc
```

Being able to execute independently of the environment, internal actions provide an additional way of specifying nondeterminism in LOTOS.

coffee; **exit** [] milk; **exit** (*1*)

is a process that is ready to synchronize on both *coffee* and *milk*.

i; coffee; **exit** [] milk; **exit** (*2*)

is a process that is ready to synchronize on *coffee*, but may not be able to synchronize on *milk*. If the environment proposes *milk,* synchronization may be impossible if the process has already decided to execute ***i***; however, if the environment proposes *coffee*, it can be assumed that the internal action will be executed eventually, and synchronization will then occur. Similarly,

i; coffee; **exit** [] **i**; milk; **exit** (*3*)

is a process that may be unable to synchronize on either *coffee* or *milk*, depending on an internal decision.

An interesting application of this concept is the specification of priorities. (*2*) can be interpreted that *coffee* has priority over *milk*. If, in addition, one wants to specify that *milk* is still possible after *coffee*, this could be written

i; coffee; milk; **exit** [] milk; **exit**

By using more complicated cascades of alternatives with internal actions, one can specify priorities with respect to any predetermined and finite number of events.

There are three parallel composition operators in LOTOS: a basic one, the *selective composition* operator, and two derived ones, the *interleaving* and the *full synchronization* operators.

The *interleaving* operator (|||) is used to express the concept of parallelism between behaviors when no synchronization is required.

(out1; out2; **exit**) ||| (in1; in2; **exit**)

is equivalent to (recall the discussion in Section 1.3 regarding the *interleaving concurrency* model of LOTOS):

```
out1;  (out2;   in1;   in2;   exit
        []
        in1;    (out2;   in2 ;  exit  []  in2;  out2;  exit)  )
[]
in1;   (in2;    out1;  out2;  exit
        []
        out1;   (in2;    out2;  exit  []  out2;  in2;   exit   )
```

The interleaving operator mimics a divorced couple, where lifestyles are completely independent. Some married couples maintain the engagement to do together certain things, for example breakfasts and movies every Thursday night. When processes must synchronize on common actions, the *selective parallel* operator, denoted by |*[L]*|, is used, where *L* is the list of actions on which synchronization must occur. For example:

```
  a; b; c; exit        (* subprocess 1 *)
|[a]|
  d;  a; c; exit       (* subprocess 2 *)
```

is equivalent to

d; a; (b; c; **exit** ||| c; **exit**)

or to

d; a; (b; (c; c; **exit** [] c; c; **exit**) [] c; b; c; **exit**)

or of course to

d; a; (b; c; c; **exit** [] c; b; c; **exit**)

This is so because both subprocesses execute independently until one of them reaches a common action, at which point it must wait to synchronize with the other subprocess. Once the second subprocess reaches the same point, synchronization is possible (depending on the context in which the process occurs, participation of other processes may be necessary also) and if it occurs both subprocesses proceed to offer their next actions. In this example, the first action of subprocess 1 is *a*. Since action *a* is common to both subprocesses, it must wait for subprocess 2 to reach action *a*, before offering action *b*. On the other hand, action *d* of subprocess 2 is not in the synchronization set, so no synchronization is required. The same is true for action *c* which is offered independently by both subprocesses. Therefore, after synchronization on *a*, both subprocesses continue independently, with all possible interleavings of the remaning actions. Clearly, when *L* is the empty list the selective parallel composition operator becomes the interleaving operator.

A special case is provided by the action δ, which is produced by ***exits***. The action δ is always considered to be a common gate, for any parallel composition operator. Therefore, all behaviors composed in parallel must synchronize on their ***exits***. In the case of enable, the action δ is transformed into an internal action, after the synchronization with other ***exits*** has occurred. For example,

a; **exit** ||| b; c; **exit**

is equivalent to

a; b; c; **exit** [] b; (a; c; **exit** [] c; a; **exit**)

Both of these behaviors are ready to synchronize with the environment on any of the following sequences:

a b c δ   *b a c* δ   *b c a* δ

Let's take another example:

(a; **exit** ||| b; **exit**) >> c; **stop**

is equivalent to

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

after synchronization on δ which has been transformed into ***i*** by the enable operator. The behavior *c;* ***stop*** is enabled only after the two ***exits*** synchronize.

There are couples where the two partners are so attached to each other that they always do everything together. If at a given moment a possible behavior of one partner is impossible for the other partner, then the first partner will not exercise that behavior. If no possible common behavior can be found, an impasse (a deadlock) occurs. The *full synchronization* operator, denoted ||, is used when the processes involved in synchronization must synchronize on every observable action. Clearly, when *L* is the list of all gates, the selective parallel composition operator becomes identical to the full synchronization operator. For example, the behavior: *a; b; c;* ***exit*** will synchronize with the behavior: *a; b; c;* ***exit***. Therefore, *a; b; c;* ***exit*** || *a; b; c;* ***exit*** is equivalent to *a; b; c;* ***exit***. As a second example, the behavior *a; b;* ***exit*** || *d; a; c;* ***exit*** is equivalent to ***stop*** (deadlock!), because the left hand side offers *a* while the right hand side offers *d*. However,

a; b; **exit** || (a; c; **exit** [] a; b*;* **exit**)

is equivalent to

a; **stop** [] a; b; **exit**

in other words it can lead to either deadlock or success, depending on a nondeterministic choice: if synchronization occurs on the first and second *a*, further synchronization is impossible.

In the presence of a choice, all the alternatives that lead to a deadlock are not considered. This can be expressed by the law: *B [] **stop*** is equivalent to *B*. For example,

a; b; **exit** || (a; c; **exit** [] c; b; **exit**)

is equivalent to *a; **stop***. The first alternative was selected because the second would have led to immediate deadlock, however the deadlock occurred after the first action.

As a further example, note that

(a; b; **stop** [] c; d; **stop**) |[a,b]| (a; b; **stop** [] d; f; **stop**)

is equivalent to

a; b; **stop** [] (c; d; **stop** ||| d; f; **stop**)

It is also interesting to consider the role of the internal action in this respect. For example,

a; **exit** || (a; **exit** [] **i**; b; **exit**)

is equivalent to

a; **exit** [] **i**; **stop**

i.e., it may deadlock if the system chooses to execute the internal action before agreeing on *a*. On the other hand,

(a; **exit** [] b; **exit**) || (a; **exit** [] **i**; b; **exit**)

is equivalent to

a; **exit** [] **i**; b; **exit**

i.e. it will not deadlock (if the environment cooperates), while

a; **exit** || (**i**; a; **exit** [] **i**; b; **exit**)

is equivalent to

i; a; **exit** [] **i**; **stop**

i.e. it may deadlock depending on what internal action is executed.

These examples show that, as already mentioned, there is only interleaving on internal actions.

The parallel composition operators *///* and *//* are commutative and associative. |*[L]*| is commutative, and may be associative depending on the gates in *L*.

2.7 Putting The Modules Together

Now that we have gained some experience with LOTOS operators and specified the behavior of each process separately, we can compose them using the parallel composition operators. But first, let us look at the complete specification and then elaborate on it.

```
1     specification Producer_Consumer  [ pc1, pc2, cc1 cc2  ] :  exit
2
3     behavior
4         (
5         Producer [ pc1, pc2 ]
6           |||
7         Consumer [ cc1, cc2 ]
8         )
9       ||
10          Channel [pc1, pc2, cc1, cc2]
11
12      where
13        process Producer [ out1, out2 ] : exit  :=   . . .        (*As defined previously*)
14        process Consumer [ in1, in2 ] : exit  :=  . . .           (*As defined previously *)
15        process Channel [ le1, le2, , re1, re2 ] : exit  := . . .  (*As defined previously *)
16      endspec
```

Before introducing the rest of LOTOS operators, we must explain some of the notions that we have just introduced. Line 1 introduces the **specification** ***Producer_Consumer***, which synchronizes with the environment through four gates *pc1, pc2, cc1, cc2.* Line 3, **behavior,** indicates the beginning of ***Producer_Consumer***'s behavior expression. Any global data abstractions would have to be declared on line 2. The behavior expression of this specification is the composition of three instances of three processes. Line 12, the **where** clause, defines the processes that are used in the behavior expression of the specification. Lines 13, 14 and 15 introduce the definitions of the three processes ***Producer, Consumer*** and ***Channel.*** Note that each process definition has a list of formal gates, which must be relabelled with actual gates.

It is important to know that relabelling is done dynamically as each action is executed, rather than statically as the process is instantiated. For example, let:

```
process    P [a, b, c] : noexit :=
               a; b; stop |[a]|  a; c; stop
endproc
```

The static relabelling of instantiating ***P*** with ***P****[c, c, a]* would be equivalent to *c; c;* ***stop*** |*[c]*| *c; a;* ***stop***, i.e. to *c; a;* ***stop***. In LOTOS' dynamic relabelling instead, the substitution is done before the actions are offered to the environment of the relabelled behavior. So, the execution of *a; b;* ***stop*** |*[a]*| *a; c;* ***stop*** results in *a; (b;* ***stop*** ||| *c;* ***stop****)*, which is equivalent to *a; (b; c;* ***stop*** *[] c; b;* ***stop****)*. This , of course, becomes *c; (c; a;* ***stop*** *[] a; c;* ***stop****)* after the relabelling.

The behavior given on lines 3 to 12 can be replaced with the following equivalent behavior.

```
behavior
    (

        Producer [ pc1, pc2 ]
      |[pc1, pc2]|
        Channel [pc1, pc2, cc1, cc2]
      |[cc1, cc2]|
        Consumer [ cc1, cc2 ]
    )
where  . . .
```

Note that the parallel composition operators have provided us with a powerful new tool to better specify the concepts introduced above. For example, a channel that can take and deliver, in any order, can be written as:

pc1; cc1; **exit** ||| pc2; cc2; **exit**

while a similar channel, which can also lose an element, can be written as:

pc1; (cc1; **exit** [] **i**; **exit**) ||| pc2; (cc2; **exit** [] **i**; **exit**)

2.8 Disable operator

The LOTOS *disable* operator *[>* models an interruption of a process by another process. So, ***P1*** *[>* ***P2*** means that, at any point during the execution of ***P1***, there is a choice between executing one of the the next actions from ***P1*** or one of the the first actions from ***P2***. Once an action from ***P2*** is chosen, ***P2*** continues executing, and the remaining actions of ***P1*** are no longer possible. If ***P1*** terminates unsuccessfully, the first actions from ***P2*** are offered, while if ***P1*** terminates successfully, ***P2*** does not start execution. For example,

a; b; **exit** [> c; d; **stop**

is equivalent to

a; (b; (**exit** [] c; d; **stop**) [] c; d; **stop**) [] c; d; **stop**

As a final requirement, let us assume that the channel may also fail at any time.

To model this fact, we put a disable [**>** at the end of the channel's behavior expression. We use the internal action **i**, to express an internal decision by the channel.

```
process Channel [ pc1, pc2, cc1, cc2] : exit :=
(
        (* same behavior expression as in Section 2.6 *)
)
[>  (   i;  (* channel goes down *)
        exit
    )
endproc
```

Also, to ensure that the ***Consumer*** terminates successfully when the ***Channel*** goes down, we must disable the existing behavior of the ***Consumer*** with an **exit** which is always ready to synchronize with the **exit** after the [> in the ***Channel***. The ***Consumer*** then becomes:

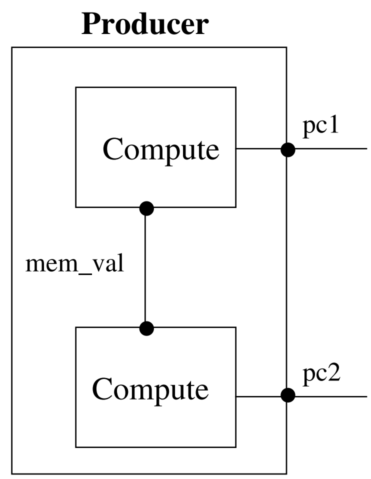
```
process Consumer [ cc1, cc2 ] : exit :=
    (
        (*  same behavior expression as in Section 2.6  *)
     )
     [>  exit
endproc
```

Note that the ***Consumer*** behavior is still equivalent to the one of Section 2.6. The difference would appear if the ***Consumer*** enabled itself recursively.

A corresponding change is necessary in the ***Producer.***

2.9 Hiding operator

The ***hide*** operator allows abstracting from the internal functioning of a process, by hiding actions that are internal to it. In particular, when the top-down approach is used, the designer can compose the system using LOTOS operators while hiding the details of interprocess communications that are irrelevant at a higher level of abstraction. For the sake of illustration, let us assume that the process ***Producer*** is composed of two subprocesses (actually two instances of the same process)



process ***Producer*** [pc1, pc2] **: exit :=**
hide mem_val **in**
(
Compute [pc1, mem_val]
|[mem_val]|
Compute [mem_val, pc2]
)
where
process ***Compute*** [v1, v2] : **exit** :=
v1; v2; **exit**
endproc
endproc

The expansion of ***Producer*** is: *pc1;* ***i****; pc2;* ***exit***, which is the behavior that we wish to model. It offers the observable action *pc1,* it performs an internal action ***i*** which represents a hidden action (the result of synchronization between *v2* and *v1* relabelled *mem_val*), then it offers action *pc2.* Note that the cooperation of the environment is required for both actions *pc1* and *pc2*, but not for ***i***. Generally speaking, an action requires cooperation of all processes that must synchronize on that action by virtue of the parallel composition operators, unless a ***hide*** hides it from external processes. If an action is not hidden with respect to the environment, cooperation of the environment is also required. The attentive reader will have noted that the absence of hiding in our ***Producer_Consumer*** specifications makes it necessary for the environment to participate in actions on gates *pc1, pc2, cc1, cc2*. This can be prevented by hiding these four gates at the top level of the specification.

As an additional example of the ***hide***, note the following:

(**hide** b **in** a; b; c; **exit**) || a; c; **exit**

is equivalent to *a; **i**; c; **exit*** because *b* is turned into an internal action which does not synchronize. Also,

hide b **in** (a; b; c; **exit** || a; c; **exit**)

is equivalent to *a; **stop***.

2. 10 Process Instantiation

Let us restate the problem requirements to illustrate process instantiation in LOTOS.

Formulation 4: Give a recursive LOTOS specification for the producer-consumer problem with a one-slot channel. The channel may lose messages and may go down at any time (in which case, a deadlock occurs).

```
specification Pro_Cha_Con [ Put, Get] : noexit

behavior
    (   Producer [ Put ]   |||   Consumer [ Get]  )
    ||
    ( Channel [Put, Get ] [>  Channel_Down )

    where
        process Channel_Down : noexit :=
            i;  (* channel goes down *)        stop
        endproc

        process Producer [ Put ] : noexit  :=
            Put;     Producer [ Put ]
        endproc

        process Consumer [ Get ] : noexit  :=
            Get;     Consumer [ Get ]
        endproc

        process Channel [ Put, Get] : noexit  :=
          Put;
            (Get;   exit    []   i; (*  lose  msg *)      exit)
            >> Channel [Put, Get ]
        endproc
endspec
```

3. Full LOTOS

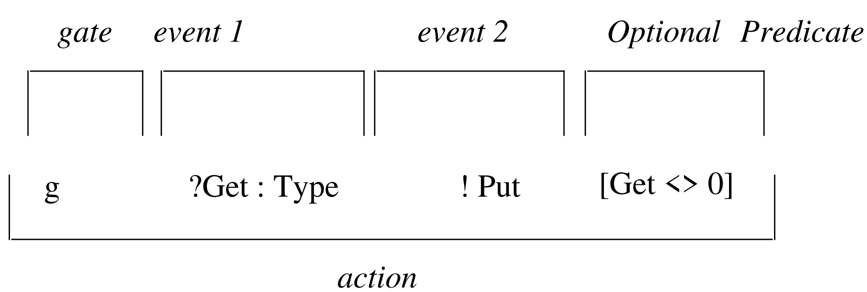
In full LOTOS, it is possible to describe process synchronization involving the exchange of data values. Data structures and value expressions are defined by using the abstract data type specification language ACT ONE [EM 85][MRV 91]. We do not describe ACT ONE, but we use some data types (called sorts in LOTOS) and value expressions. In our examples, we use only two sorts: Booleans and Natural numbers, whose domains are *true, false* and *0*, *1*, *2*, ... respectively. We also use some operations on these sorts.

The reader should note that, for the sake of clarity, in this section we shall take some licences with LOTOS syntax, mostly concerning the representation of natural numbers and the use of some operation symbols. Hopefully these "licences" will become accepted syntax in the expected enhancements of the language.

LOTOS variables are, more properly, *value identifier*s, i.e., place holders for value expressions to be generated during execution. The definition of a variable in LOTOS has the form *Var_Name: Sort* where *Var_Name* is a variable name that can take any value expression of sort *Sort*. For example*, X: Nat* is a variable whose values are in the domain of *Nat*. The expression *1 + 2* has a value *3* which is in the domain of *Nat*.

3.1 Actions in full LOTOS

In basic LOTOS, we defined an action to be synonym with gate. In full LOTOS an action, an example of which is shown in the diagram below, is formed of three components: *a gate*, a list of *events*, and an optional *predicate*. Processes synchronize their actions, provided that they name the same gate, that the lists of events are matched, and that the predicates, if any, are satisfied. An event can either *offer* (!) or *accept* (?) a value. Predicates establish a condition on the values that can be accepted/offered.



As an example, consider the following action *a*:

g ?X:Nat !1 [X =< 2]

This is a LOTOS action that occurs at gate *g* and expects from the environment a value for *X* of sort *Nat* restricted to be less than or equal to *2*, while at the same time offering the value *1*. This is equivalent to offering a choice between the following three actions:

g ! 0 ! 1
g ! 1 ! 1
g ! 2 ! 1

We express this by the notation:

tuples(a) = { !0 !1, !1 !1, !2 !1 }.

Of course, *tuples* can be an infinite set, e.g.

tuples(g ?x:nat).

3.2 Synchronization with value establishment

Interprocess synchronization with value exchange occurs when two or more processes agree on a single tuple of values to be established on a gate. This is the case of matching actions. More precisely, if a set of processes, say $\boldsymbol{P_1, \ldots, P_n}$, is composed in parallel and **G** represents the list of common gates on which synchronization must occur, then these processes can synchronize if the following conditions hold:

1. $\boldsymbol{P_1, \ldots, P_n}$ are all offering actions $a_1, \ldots, a_n$ respectively;
2. The gate of actions $a_1, \ldots, a_n$ is the same and is in $\boldsymbol{G}$; and
3. T = tuples(a_1) $\cap \ldots \cap$ tuples(a_n) $\neq \emptyset$.

For example if $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ are two communicating processes, and $\boldsymbol{P_1}$ offers action *a* above, and, at the same time, $\boldsymbol{P_2}$ offers the following action *b*:

g ?Y:Nat ?Z:Nat [Z >= Y and Z < 2]

then,

tuples(*b*) ={ !0 !0, !0 !1, !1 !1 }

and

T = tuples(*a*) $\cap$ tuples(*b*) = {!0 !1, !1 !1 } $\neq \emptyset$

This means that synchronization can occur with the environment on one of two actions, i.e.

g !0 !1 or g !1 !1

This will cause all variables appearing in the actions to acquire the corresponding value, and these values will be replaced for the variables throughout the behavior expression. Suppose that the first action is chosen, then *0* is established as the value for *X* in action *a* and *Y* in action *b*. As a consequence, *X* becomes *0* in the behavior expression containing action *a* and also *Y* becomes *0* in the behavior expression containing action *b*.

3.3 Behavior expressions in full LOTOS

In this section we provide an introduction to full LOTOS by constructing a new version of the producer-consumer problem.

Formulation 5: Give a Full LOTOS specification for the process ***Consumer***. The consumer accepts a single message, with an even sequence number, then ***exits***.

The consumer specification becomes:

```
process Consumer [In_Ele]:exit:=
    In_Ele   ? Msg:Nat   ? Msg_Seq:Nat       [(Msg_Seq mod 2) = 0];
    exit
endproc
```

The action on the second line is described in terms of one gate *In_Ele*, two events *?Msg:Nat* and *?Msg_Seq:Nat,* which represent the readiness of the process to accept two values, and the predicate *[(Msg_Seq mod 2) = 0]* which restricts the sequence number to be even. This process is equivalent to the following infinite one:

```
process Consumer [ In_Ele] : exit:=
    In_Ele   ?Msg:Nat   !0;  exit
    []
    In_Ele   ?Msg:Nat   !2;  exit
    []
    In_Ele   ?Msg:Nat   !4;  exit
    []
    .  .  .  .  .
endproc
```

3.3.1 Successful Termination with parameters

This is denoted by ***exit***$(E_1, ... ,E_n)$. This behavior offers the action: $\delta\ !E_1\ \ ...\ \ !E_n$. $E_1, ... ,E_n$ are value expressions, the results of the process that executes the ***exit***. These results are passed to the enabled behavior if the process enables another process. The following ***Consumer*** specification reflects the behavior of a consumer that accepts a single message with an even sequence number, then terminates successfully with that sequence number.

```
process Consumer [ In_Ele] : exit (Nat):=
    In_Ele   ?Msg:Nat  ?Msg_Seq:Nat      [(Msg_Seq mod 2) = 0];
    exit(Msg_Seq)
endproc
```

Note the syntax of the ***exit*.** The number of parameters, their order and their types in the ***exit*** that appears in the process definition must be compatible with those in the ***exit*** that appears in the process body. The expansion of this specification is an infinite one, i.e.:

```
process Consumer [In_Ele] : exit(Nat):=
    In_Ele   ?Msg:Nat    !0;    exit(0)
    []
    In_Ele    ?Msg:Nat    !2;    exit(2)
    []
    .   .   .   .   .   .
endproc
```

Processes that are composed by way of any of the parallel composition operators must synchronize on δ, in order to successfuly terminate. That is, all processes must offer ***exit***s with the same number of parameters and these must match in the same way as events do in an action synchronization, described earlier. A parameter that can match with any value of sort *S* is denoted by ***any*** *S*. For example if ***P1*** and ***P2*** are composed in parallel, and ***P1 exit***s with ***exit** (**any** Boolean, 1)* producing the action δ *?dummy:Bool !1* and ***P2 exit***s with ***exit** (true, **any** Nat)* producing δ *!true ?dummy:Nat,* then these two actions will match and their environment must offer an action that matches the resulting action of ***P1*** and ***P2*** which is δ*!true!1*. Such matching actions could be provided by other ***exits***, or by an ***accept***, as will be seen in Section 3.3.4.

The corresponding changes for the ***Channel*** and ***Producer*** are straightforward.

3.3.2 Process Instantiation with Parameters

Formulation 6: Modify the consumer so that it receives messages with sequence numbers and counts the number of messages with odd sequence numbers and messages with even sequence numbers. It **exit**s with the total number of messages received, which may be zero.

```
process Consumer[In_Ele](Odd_Num_Msg, Even_Num_Msg:Nat):exit(Nat, Nat):=
    In_Ele   ?Msg:Nat   ?Msg_Seq:Nat           [(Msg_Seq mod 2) = 0];
    Consumer[ In_Ele] (Odd_Num_Msg, Even_Num_Msg + 1)
    []
    In_Ele   ?Msg:Nat   ?Msg_Seq:Nat            [(Msg_Seq mod 2) <> 0];
    Consumer[ In_Ele] (Odd_Num_Msg + 1, Even_Num_Msg)
    []
    exit(any Nat, Odd_Num_Msg + Even_Num_Msg)
endproc
```

3.3.3 Guarded behavior

A behavior expression can be preceded by a guard that must be true in order for the former to be enabled. For example

```
g?x:Nat [x =< 5];
    ([X = 2 or X = 3] -> g  !X  !0; stop
    []
    [X < 3]              -> g  !X  !1;    stop)
```

is equivalent to

```
   g!0; g!0!1; stop
[] g!1; g!1!1; stop
[] g!2;  (g!2!0; stop [] g!2!1; stop)
[] g!3; g!3!0; stop
[] g!4; stop
[] g!5; stop
```

The ***Consumer*** can be specified equivalently using guards as follows:

```
process Consumer[In_Ele](Odd_Num_Msg, Even_Num_Msg:Nat):exit(Nat,Nat):=
    In_Ele ?Msg:Nat ?Msg_Seq:Nat ;
    (
      [(Msg_Seq mod 2) = 0] ->
```

```
        Consumer[ In_Ele] (Odd_Num_Msg, Even_Num_Msg + 1)
      []
      [(Msg_Seq mod 2) <> 0] ->
        Consumer[ In_Ele] (Odd_Num_Msg + 1, Even_Num_Msg)
    )
    []
    exit(any Nat, Odd_Num_Msg + Even_Num_Msg)
endproc
```

In general,

$a[P_1];\ B_1\ []\ a[P_2];\ B_2\ []\ ...\ []\ a[P_n];\ B_n$

where P_i stands for a predicate and B_i stands for a behavior expression, is equivalent to

```
a;
(   [P1] -> B1
[]
    [P2] -> B2
[]
    .   .   .
[]
    [Pn] -> Bn
)
```

if and only if exactly one of P_1 or P_2 ... or P_n is true for any values agreed on by an action *a.* This is the case in our example, where any message sequence agreed on must be even or odd.

3.3.4 Sequential Composition with Value Passing (enable with accept)

This has the form:

B_1 >> **accept** $X_1{:}S_1$, ... ,$X_n{:}S_n$ **in** B_2

B_1 will be executed until it terminates. If it ***exit***s, then the next behavior expression is B_2 where the variables X_1 to *Xn* in B_2 are substituted for the value results of B_1 (i.e. the ***exit*** parameters). The number and sorts of the values that are passed at the successful termination of B_1 must be the same as those of the variable declarations in the ***accept*** statement. The enable produces an internal action as seen previously.

Here is an example of a producer that generates a message and sends it with a sequence number or may ***exit*** with the number of messages sent.

process ***Producer*** [Out_Ele](Msg_Seq:Nat) : **exit**(Nat, Nat) :=
(***Generate_Ele*** >> accept Msg:Nat in
Out_Ele !Msg !Msg_Seq;
Producer[Out_Ele](Msg_Seq + 1)
)
[]
exit(Msg_Seq, **any** Nat)
endproc

The process ***Generate_Ele*** has no external gates, its mission is to generate internally a value that is going to be accepted by the ***Producer*** and sent to the ***Consumer***.

process ***Generate_Ele*** : **exit**(Nat) :=
exit(**any** Nat)
endproc

3.3.5 Summation on values

This has the form

choice X:S [] B

which is equivalent to

$[t_1/X]B$ []... [] $[t_n/X]B$

where $t_1,...,t_n$ are all possible value expressions of sort S. $[t_i/X]B$ is the resulting behavior by substituting t_i for X in B.

For example

choice x: Nat []
[x mod 2 = 0] -> g!x; **stop**

is equivalent to

```
  g  !0; stop
[]
  g  !2; stop
[]
  g  !4; stop
[]
.  .  .  .
```

and therefore to

```
g   ?x  [ x mod 2 = 0 ];  stop
```

The ***choice*** operator, in conjunction with the internal action, can be used in order to specify nondeterministic choice. For example,

```
choice  x:Nat [] [x mod 2 = 0] -> i;     g!x;    stop
```

specifies nondeterministic choice of just one even integer.

An elegant example of the usefulness of the choice operator is given in [Tur 88a], where a sorting process is specified in LOTOS as

```
input ?UnsortedList: NatList;
choice SortedList: NatList []
     [IsPermuted (SortedList,  UnsortedList)  and   IsOrdered (SortedList)] ->
                              output  !SortedList
```

(of course, definition of the data type operators *IsPermuted* and *IsOrdered* must be provided to complete this specification).

Nested choices can be used in order to impose different constraints on the set of offerings, depending on some predicates. For example:

```
g?x:Nat;
choice y:Nat []
     [y > x]-> (choice w,z:Nat []
                    [w * z = y]-> g!z; exit
                 []   [prime(y)]-> i; g!y; exit
              )
```

The process partially specified above first selects a value for *x* in collaboration with the environment, and then can: either be ready to offer any *z* that is a factor of any *y* greater than *x*; or decide to offer a nondeterministically chosen prime *y* greater than *x*.

Here is another version for ***Consumer*** using the ***choice*** operator, which is equivalent to the previous two specifications.

```
process Consumer[In_Ele](Odd_Num_Msg, Even_Num_Msg:Nat):exit(Nat,Nat):=
   (
   choice Msg_Seq:Nat []
    [(Msg_Seq mod 2) = 0] ->
      In_Ele ?Msg:Nat !Msg_Seq ;
      Consumer[ In_Ele] (Odd_Num_Msg, Even_Num_Msg + 1)
    []
    [(Msg_Seq mod 2) <> 0] ->
      In_Ele ?Msg:Nat !Msg_Seq ;
      Consumer[ In_Ele] (Odd_Num_Msg + 1,  Even_Num_Msg)
   )
   []
   exit(any Nat, Odd_Num_Msg + Even_Num_Msg)
endproc
```

To complete the specification of producer-consumer, we specify an unreliable one-place channel which may lose messages, as specified by an internal action ***i***, or simply ***exit***.

```
process Channel [ In_Ele, Out_Ele ] : exit (Nat, Nat) :=
   In_Ele ?Msg:Nat  ?Msg_Seq:Nat;
   (
    Out_Ele !Msg ! Msg_Seq;                (* deliver message *)
    Channel[ In_Ele, Out_Ele]
    []
    i;                                     (* lose message *)
    Channel[ In_Ele, Out_Ele]
   )
   []
   exit(any Nat, any Nat)
endproc
```

The global behavior of the producer-consumer can be specified as

```
specification Producer_Consumer  [ Ele1, Ele2  ] : exit(Nat, Nat)
```

behavior

__Producer__ [Ele1](0)

|[Ele1]|

__Channel__ [Ele1, Ele2]

|[Ele2]|

__Consumer__ [Ele2](0,0)

where

(* Definitions of *__Producer__*, *__Consumer__*, *__Generate_Ele__*, and *__Channel__* go here. *)

endspec

Or equivalently it can be specified as

specification *__Producer_Consumer__* [Ele1, Ele2] : **exit** (Nat, Nat)

behavior

(*__Producer__* [Ele1](0)

|||

__Consumer__ [Ele2](0,0))

||

__Channel__ [Ele1, Ele2]

where

(* Definitions of *__Producer__*, *__Consumer__*, *__Generate_Ele__*, and *__Channel__* go here *)

endspec

The above two specifications are equivalent since *Ele1* and *Ele2* are the only gates used by *__Producer__*, *__Consumer__* and *__Channel__*. The *__exit__* in the *__Producer__*, *__Consumer__* and *__Channel__* can only be executed when the three processes are ready to offer *__exit__* at the same time, that is when *__Producer__* offers

exit(Number_Of_Messages_Sent, **any** Nat)

and the *__Consumer__* offers

exit(**any** Nat, Number_Of_Messages_Received)

and the *__Channel__* offers

exit(**any** Nat, **any** Nat)

the resulting ***exit*** due to the synchronization of the above three ***exit***s is

exit(Number_Of_Messages_Sent,Number_Of_Messages_Received)

If this specification was part of a larger one, process ***Producer_Consumer*** could enable another process, which could determine the number of messages lost by the ***Channel*** by computing:

Number_Of_Messages_Sent - Number_Of_Messages_Received

4. Priority of Operators

As in most languages, a priority of operators exists in LOTOS in order to reduce the number of parentheses needed. In order of decreasing priority, we have:

; -> [] |[L]| [> >> **hide** **par** **choice** **let**

operators of equal priority associate to the right.

5. Further Readings

Two unpublished tutorials which have achieved different degrees of distribution are [ISO1 89] [Tur 88b], and we should also mention [Hog 90], which is in German. [ISO4 89] will soon become available as an ISO technical Report. It contains substantial examples of LOTOS specifications. The series of books *Protocol Specification, Testing, and Verification* and *Formal Description Techniques* published yearly by North-Holland, contains many papers on LOTOS and related subjects. The book [VVD 89] is a collection of research papers on LOTOS produced within the European community's SEDOS project.

The reader must have observed that in this paper we repeatedly showed different ways of writing behaviorally similar LOTOS specifications. In fact, several *specification styles* can be used in LOTOS. This point is developed further in [VSVB 91].

6. LOTOS Applications

Although protocols have been and still are the main area of utilization of LOTOS, a number of papers have been published recently that show that the language eventually may be useful well beyond that area.

Among others: [FLS 90] shows the application of the language to the formal specification of telephone call processing. [Va 89] shows an application to the specification of security mechanisms. [P 90] discusses the application to the specification of a distributed operating system. [HL 91] shows how LOTOS can be used for the specification of distributed algorithms.

7. LOTOS Tools

A number of software tools has been developed for supporting the use of LOTOS. Two were mentioned in Section 1.2. LOLA [QPF 89] is a "parameterized expander". TOPO [MM 89] is a translator from LOTOS to C. TETRA [BB 89][BDD 90] is a "trace checker", i.e. it checks whether a given event trace can be executed by a given specification. Among the verification tools we note: Squiggles, a tool to verify strong and weak observation equivalence between basic LOTOS specifications having finite-state representations [BC 89]; and CAESAR/ALDEBARAN, a model-checking tool using an intermediate Petri Nets representation [GS 90]. We refer again to the books mentioned in Section 4 for a more complete view of the tools available.

Acknowledgment. Funding sources for our work include the Natural Sciences and Engineering Research Council of Canada, the Telecommunications Research Institute of Ontario (Design of Validation Environments project) and the Canadian Department of Communications. We are indebted to Souheil Gallouzi, Jacques Sincennes and the anonymous referees for their useful comments.

REFERENCES

[BB 87] Bolognesi, B., and Brinksma, E. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems* 14 (1987) 25-59. Also reprinted in [VVD89] 23-73.

[BB 89] Bochmann, G.v., and Bellal, O. Test Result Analysis in Respect to Formal Specifications, Proc. 2nd Int. Workshop on Protocol Test Systems, Berlin, Oct. 1989, pp.272-294.

[BC 89] Bolognesi, T., and Caneve, M. Equivalence Verification: Theory, Algorithms, and a Tool. In [VVD] 303-326.

[BDD 90] Bochmann, G.v. , Desbiens, D., Dubuc, M., Ouimet, D., and Saba, F. Test Result Analysis and Validation of Test Verdicts. To appear in the Proceedings of the Workshop on Protocol Test Systems, McLean, Virginia, (Oct. 1990).

[EM 85] Ehrig, H., Mahr, B., *Fundamentals of Algebraic Specification 1*, Springer-Verlag, Berlin, 1985.

[FLS 90] Faci, M., Logrippo, L., and Stepien, B. Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Approach. To appear in Computer Networks and ISDN Systems.

[GS 90] Garavel, H., and Sifakis, J. Compilation and Verification of LOTOS Specifications. In: Logrippo, L., Probert, R.L., and Ural, H. (eds.) *Protocol Specification, Testing, and Verification, X.* North-Holland, 1990, 379-394.

[Hoa 85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hog 89] Hogrefe, D. *Estelle, LOTOS und SDL.* Springer Verlag, 1989

[HL 91] Haj-Hussein, M., and Logrippo, L. Specifying Distributed Algorithms in LOTOS. To appear in the Proceedings of Computer Networks, Wroclaw 1991.

[ISO1 89] International Organization for Standardization. IS 8807: LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behavior (1989).

[ISO2 89] International Organization for Standardization. ISO/IEC JTC1/SC6 N 6116: Revised Text of ISO/DTR 10023 - Formal Description of ISO 8072 in LOTOS (1990)

[ISO3 89] International Organization for Standardization. IS 9074: Estelle, A Formal Description Technique Based on an Extended State Transition Model (1989).

[ISO4 89] International Organization for Standardization. ISO/IEC JTC 1/SC21 N 3252: Guidelines for the Application of Estelle, LOTOS and SDL (1989).

[LOBF 88] Logrippo, L., Obaid, A., Briand, J.P., and Fehri, M.C. An Interpreter for LOTOS, a

Specification Language for Distributed Systems. *Software-Practice and Experience*, 18 (1988) 365-385.

[Mil 80] Milner, R. A Calculus of Communicating Systems. *Lecture Notes in Computer Science No.92*, Springer-Verlag, 1980.

[Mil 89] Milner, R. *Communication and Concurrency*. Prentice-Hall, 1989.

[MM 89] Mañas, J.A., and de Miguel-More, T. From LOTOS to C. In: K.J.Turner (ed.) *Formal Description Techniques*, North-Holland, 1989, 79-84.

[MRV 91] deMeer, J., Roth, R., and Vuong, S. Introduction to Algebraic Specifications Based on the Language ACT ONE, this issue.

[P 90] Pecheur, C. An Overview of the LOTOS Specification of Chorus V3. Report No. S.A.R.T. 89 - 03 -13, Université de Liège, B28, Département de Systèmes et Automatique, Mai 1989.

[QPF 89] Quemada, J., Pavón, S., and Fernandez, A. Transforming LOTOS Specifications with LOLA. In: Turner, K.J. (ed.) *Formal Description Techniques*, North-Holland, 1989, 45-54.

[Tur 88a] Turner, K. Constraint-Oriented Style in LOTOS. In: *Proc. of the British Computer Society Workshop on Formal Methods in Standards*, Didcot, April 1988.

[Tur 88b] Turner, K., The Formal Specification Language LOTOS: A Course for Users, Course notes, University of Stirling, June 1988.

[V 88] van Eijk, P. Software Tools for the Specification Language LOTOS. Doctoral Thesis, Universiteit Twente (1988).

[Va 89] Varadharajan, V. Use of a Formal Description Technique in the Specification of Authentication Protocols. Computer Standards and Interfaces 9 (1989/90), 203-205.

[VCA 89] Vissers, C. A., LOTOS Background, in [VVD 89] 15-22.

[VSVB 91]Vissers, C. A., Scollo, G., van Sinderen, M., and Brinksma, E. On the Use of Specification Styles in the Design of Distributed Systems. To appear in Theoretical Computer Science.

[VVD 89] van Eijk, P., Vissers, C.A., and Diaz, M. *The Formal Description Technique LOTOS.* North-Holland, 1989.

Pictures and vita of M. Faci and L. Logrippo should be in North-Holland files (see your reference COMNET 00866).

Mazen Haj-Hussein holds a Bachelor and Master degree in the Computer Science Departement of the University of Ottawa, where he is now working towards his PhD.

The picture of Mazen will follow with the galley proofs.