E-LOTOS Tutorial with Examples

Alberto Verdejo
Dept. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid.
E-28040 Madrid. Spain.
jalberto@eucmos.sim.ucm.es

10th April 2000

This tutorial introduces E-LOTOS by describing all the features of the language, and by showing its expressive power. We show how E-LOTOS can be used to specify systems, their behaviour, and the values they manipulate, as well as how the specifier can modularize systems. The tutorial includes some *small* examples that show how E-LOTOS features are used to specify common problems, usual generic data types, well-known concurrent programming problems, as well as to describe hardware components.

E-LOTOS is still a draft, under revision. So this is still an unfinished document, it has to be completed when the ISO standard will be definitively approved.

Contents

1 Introduction

> If cars had improved at the rate of computers in the same time period, a Rolls Royce would now cost 10 dollars and get a billion miles per gallon. (Unfortunately, it would probably also have a 200-page manual telling how to open the door.)
>
> – Andrew S. Tanenbaum

LOTOS (Language Of Temporal Ordering Specification) [ISO89] is a Formal Description Technique[1] developed within ISO for the formal specification of networked and distributed systems. It is based on the intuitive and well known *black box* analogy [Mil89] where systems are described as black boxes with buttons, that represent their entire capability of communication. The basic idea is that systems can be specified by defining the temporal relation among the interactions that constitute the *externally* observable behaviour of a system. As stated in [Mil89] "the behaviour of a system is exactly what is observable, and to observe a system is exactly to communicate with it."

LOTOS is composed of a process algebra part (based on CCS [Mil89] and CSP [Hoa85]) to describe systems, and an algebraic language (ACT ONE [EM85]) to describe the abstract data types. LOTOS has proven very successful in specifying protocols and services (examples can be found in [vEVD89, Gam90, Mun91, Pec92, BL93, GH93]). However, LOTOS has several limitations related to its expressive power and structuring capabilities, besides user-friendliness. For these reasons, LOTOS is currently under revision in ISO [Que98], in the Work Item "Enhancements to LOTOS," giving rise to a revised language called E-LOTOS (Enhanced LOTOS). E-LOTOS has a similar structure to LOTOS. It has a behavioural process algebra part, which inherits some operators from LOTOS, generalizes others, and adds new operators. It also has a functional data definition part which allows constructive definitions of data types and functions for manipulating them, and is thought to be more user friendly.

Among the enhancements introduced in E-LOTOS, the most important ones are:

- the notion of quantitative time: in E-LOTOS we can define the exact time of events (by adding annotations to actions, see Section 2.1) or behaviours (by using **wait** statements, see Section 2.16)
- the data part has a new definition for data types and the construction of values of predefined types
- modularity allows the definition of types, functions, and processes in separate modules, controlling their visibility by means of module interfaces, and the definition of generic modules, useful for code reuse. One module can use another one by means of importation clauses.

[1] "Formal Description Techniques (FDTs) are methods of defining the behaviour of an (information processing) system in a language with formal syntax and semantics, instead of a natural language as English." [ISO89]

Although in this tutorial we describe all the operators of E-LOTOS, the original LOTOS operators and the new ones, now we concentrate on the latter. Among the new operators, we have:

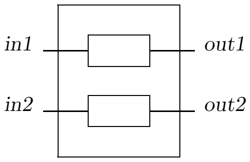
- the sequential composition operator that allows concatenation of two processes. This operator unifies both the action prefix operator and the enabling operator of LOTOS.
- the general parallel operator that allows the synchronization of n among m processes.
- the suspend/resume operator, which generalizes the disabling operator of LOTOS by allowing a disabled process to be resumed by the disabling process.
- operators to raise exceptions and to handle them.
- the renaming operator which allows renaming of actions and exceptions, and to modify their parameters.

E-LOTOS has several imperative features, which have been introduced in order to help the user of this programming paradigm to specify systems. Assignment, declaration of local variables, and several iterative operators are some of these features. We describe them in Section 2.19.

In the following two chapters we describe the Base Language, that is the part of E-LOTOS used to write the behaviour of processes, Chapter 2, and the data types, Chapter 3. In Chapter 4 we describe the Module Language, that is the language used to modularize the specifications. Chapter 5 includes several examples that show how E-LOTOS can be applied to specify different kinds of systems.

1.1 An example: two-position register

In order to show the basic ideas of E-LOTOS specification, let us begin with as simple example. Coming back to the idea of the black boxes, let us imagine that we want to specify a two-position register, graphically described by



At this level, we only know that the register has four gates (buttons of the black box) *in1*, *in2*, *out1*, and *out2*. A specification of the behaviour of this register would be a description of when the buttons (or gates) are available to be pressed. Generally, some gates are *input gates* (*in1* and *in2*) and values can go through them, and other gates are *output gates* (*out1* and *out2*). If the register has the following restrictions:

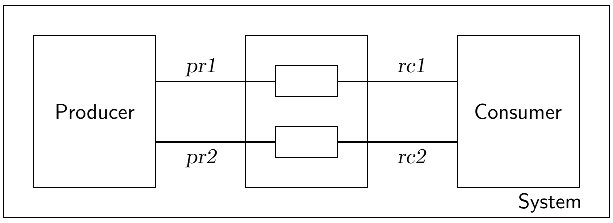
- the register is initially empty,
- both data inputs are needed before there is output,
- data through gate *in1* has to arrive before data through gate *in2*, and
- data have to be output in the same order,

we would describe it as

```
process Register [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1); in2(?x2); out1(!x1); out2(!x2)
  endvar
endproc
```

This example will be used during the tutorial and, then, all the details will be given (see Section 2.3). Note how the order of the communications through gates are expressed (using the sequential composition operator ;) and how the values communicated are represented with variables.

If we imagine other two black boxes representing a producer, that produces two values and saves them in the register, and a consumer that takes the values from the register, we can put them together, communicating:



The complete system is described in E-LOTOS as follows:

```
process System is
    hide pr1:data,pr2:data,rc1:data,rc2:data in
        conc
            Producer [pr1,pr2]()
        |[pr1,pr2]|
            Register [pr1,pr2,rc1,rc2]()
        |[rc1,rc2]|
            Consumer [rc1,rc2]()
        endconc
    endhide
endproc
```

where Producer and Consumer are processes specifying the behaviour of the producer and the consumer (see Section 2.7). They are put together with the Register, running in parallel, and communicating through gates. The whole system has no gate offered to the environment, because it represents a complete, closed system. All its behaviour is described and occurs internally.

1.2 Variables

Regarding the construction of values and the use of variables, E-LOTOS was thought as a functional language, in the sense that there is no idea of state and variables are given a value only once. When a variable is given a value, this value is substituted for the variable in the successive behaviour. However, this idea changed during the design of the language and, although values are still built with constructors as in functional language, E-LOTOS has *write-many* variables, that is, variables that can be assigned several times. Variables are declared with the **var** operator (Section 2.19). When a variable is declared, it is given a type and there is the possibility to give it an initial value. We have seen a variable declaration in the Register process:

```
var x1:data,x2 : data in
  in1(?x1); in2(?x2); out1(!x1); out2(!x2)
endvar
```

A variable can have a value of its declared type or a value of a subtype of its declared type, and this value can be changed during the execution of a specification.

The introduction of this kind of variable has affected the use of several operators. Thus, we will describe how variables can be used when several processes are composed by using any of these operators.

1.3 Time in E-LOTOS

Time is a very important aspect in concurrent, distributed systems. Although it is not present in classical process algebras, there have been many proposals to introduce time in process algebras [RR86, NS91, Yi91, BB93, Sch95]. The introduction of time in LOTOS has been dealt in [LL94, dFLL$^+$95].

The introduction of time is one of the most important enhancements of E-LOTOS. The specific features that E-LOTOS provides to manage time are three. The first one is that the specification writer can describe *when* the actions a process performs may occur, and this can be described with great flexibility, as we show in Section 2.1.

The second feature that E-LOTOS provides is the **wait** instruction, that, given a duration time d, represents that the process idles the time d.

The type time is the third feature introduced in E-LOTOS with respect to control of time. In E-LOTOS, no data type time is defined, but only the properties that it has to fulfill are enumerated. So, each *implementation* of the language can define a different time, provided that the following requirements are fulfilled. The properties are:

- the time domain is a commutative, cancellative monoid with addition + and unit 0. Thus, it satisfies the properties:
  - $d_1 + d_2 = d_2 + d_1$
  - if $d_1 + d = d_2 + d$ then $d_1 = d_2$
  - $d_1 + (d_2 + d_3) = (d_1 + d_2) + d_3$
  - $d + 0 = 0 + d = d$

  where d_1, d_2, and d are variables over the time domain.
- the order given by $d_1 \leq d_2$ if and only if $\exists d \, . \, d_1 + d = d_2$ is a total order.

For a description of the different time domains used in the literature see, for example, [NS91]. At first glance, it seems that it is better to have a continuous time domain. But, as shown in [LDV99], it has several problems, for example, we can define processes that freezes time. It is enough to have a discrete time domain and define the *unit of time* as small as we want. This is not a restriction decision since all computers are ruled by an internal discrete clock. In this tutorial we assume that the type time has been declared as a synonym of the type nat of natural numbers.

Although time is foreign to a process in the sense that the process cannot control time and has to *coexist* with it, the introduction of time has affected the meaning of the different operators of E-LOTOS. We will describe in the following sections how these operators behave in relation to time. When we say that a process *lets time pass*, we mean that the process can be idle (doing nothing but waiting) a concrete period of time. Thus, when we say that a process does not let time pass, we mean that the process *must* do something immediately. This is related to the notion of *urgency*. When an action is *urgent* it must be performed as soon as possible. This means that when an urgent action is enabled, it must be performed unless other action was performed without consuming time. Urgency is used to guarantee the progress of a system: if there were no urgent actions the systems could idle forever. For example, the special internal action **i** (Section 2.4) is urgent. We can specify which actions are urgent at certain level in order to ensure the evolution of the system. This is very useful when different components of a system are communicating, as we will see in Section 2.14, where the hiding operator is described. Not only actions may be urgent, there are other constructions in E-LOTOS, for example assignments, that are also urgent. Due to urgency, one could write specifications that *blocks* time. This is an undesirable and counterintuitive feature. The idea of time blocking is that a behaviour is performing internal actions urgently, thus time is not allowed to pass (see Section 2.6).

⌚ In E-LOTOS time is *deterministic*. As stated in [NS91], it is usually admitted that when a process P is idle (does not perform any action) for some duration d, then the resulting behaviour is completely determined from P and d. The avoidance of time nondeterminism has also affected several operators. We will describe how these operators have been affected.

We write the symbol ⌚ in the margin when we speak about time in relation with the operator which is being described.

2 Base Language for processes

In this chapter we are going to show the constructions of E-LOTOS used to describe the behaviour of systems. In Chapter 3 we will explain those constructions used to describe the values and types of data that these systems can manage.

We try to give this description in a progressive way, that is we begin with the simple operators and go to the more complex ones. In order to show examples, we sometimes have to include operators that are explained later.

All the constructions are illustrated with small examples that show a concrete use of each one. In Chapter 5 we will introduce more complex examples.

2.1 Actions

In E-LOTOS a concurrent system is specified as a process that is composed of other processes interacting with each other. This interaction between processes is carried out by means of *actions*, which represent synchronization or communication between processes through *gates*.

The E-LOTOS syntax to indicate that a process is carrying out an action is as follows:

$$G\ [\texttt{(}P_1\texttt{)}]\ [\texttt{@}P_2]\ [\ \texttt{[}\ E\ \texttt{]}\ ]$$

where the components between [ ] are optional (do not confuse with these brackets `[]` which are part of the syntax), G is the name of a gate, P_1 and P_2 are patterns (see Section 3.6) and E is a boolean expression.

The simplest action is a synchronization in which only the name of the gate, G, is indicated.

Value passing

If the action stands for a communication in which there is information exchange among the different processes that are communicating, this information is indicated by the first pattern P_1. Although the different patterns of E-LOTOS are shown later on in Section 3.6, now we will introduce some of them by means of examples. For example, if a process has a gate called *outP* and it wants to communicate on it the value 3, in E-LOTOS we should indicate it by using the behaviour[1]

$$outP\texttt{(!3)}$$

Although perhaps the first interesting example of this tutorial should be[2]

$$outP\texttt{(! "Hello, world!")}$$

[1] We use the word “process” to refer to both an abstract entity which can communicate with other processes and the declaration of this entity in E-LOTOS. The word “behaviour” refers to terms constructed by combining the different operators of the language in order to describe the behaviour of processes.

[2] `"Hello, world!"` is a constant of the predefined type `string`, Section 3.1.

If we want to communicate several values at a time we have to build what is called in E-LOTOS terminology a *record* of values, a list of transmitted values enclosed between parentheses. For example, if the process wants to communicate both values 3 and `"Hello, world!"`, we write

$$\mathit{outP}\texttt{(!(3,"Hello, world!"))}$$

When we write the record of values that a gate communicates, we can give a name to each value (that is, to each field of the record).[3] Thus we can write

$$\mathit{outP}\texttt{(!(}\mathit{value}\texttt{=>}\,3\texttt{,}\mathit{greeting}\texttt{=> "Hello, world!"))}$$

If the process has another gate called *inP*, on which data is received and is saved in the variable x, then we write

$$\mathit{inP}\texttt{(?}x\texttt{)}$$

If the process receives a record of values through gate *inP*, we can save the whole record value in the variable x as above, or we can use different variables to save the different fields of the record,

$$\mathit{inP}\texttt{((}\mathit{value}\texttt{=> ?}v\texttt{,}\mathit{greeting}\texttt{=> ?}g\texttt{))}$$

In order to prevent errors, in E-LOTOS there is the possibility (but not the obligation) of typing the different components which appear in a process specification. In particular, we can type the gates of a process so a gate can only communicate values of that type.

In the previous example, if the process receives only integers on the gate *inP*, we can write

$$\mathit{inP}\texttt{(?}x\texttt{:}\mathsf{int}\texttt{)}$$

A process that takes a record with an integer and a string looks like

$$\mathit{inP}\ \texttt{((?}x\texttt{:}\mathsf{int}\texttt{, ?}y\texttt{:}\mathsf{string}\texttt{))}$$

We can also specify a *selection predicate* in an action, which specifies the conditions that the transferred values have to fulfill. For example, if a process should receive only integers smaller than 10, we would write

$$\mathit{inP}\texttt{(?}x\texttt{:}\mathsf{int}\texttt{) [}x\texttt{<}10\texttt{]}$$

Timed constraints

It is assumed that the execution of an action takes no time, that is, actions are *atomic* and *durationless*. However, communications can be made sensitive to time by adding the `@` P_2 annotation, which pattern-matches the pattern P_2 to the time when the action happens, measured from the time when the communication was enabled. We can use this, joined with the use of selection predicates, to control the time when actions may be performed.

The behaviour

$$\mathit{inP}\texttt{(?}x\texttt{:}\mathsf{int}\texttt{) @?}t\texttt{ [}t\texttt{<}5\texttt{]}$$

specifies an action that receives an integer that is bound to the variable x provided that less than 5 units of time have passed, whereas the action

$$\mathit{inP}\texttt{(?}x\texttt{:}\mathsf{int}\texttt{) @!}5$$

[3] In this case, the gate has to be declared as having the corresponding record type, see Section 3.2.

can only occur 5 units of time after the action *inP* had been enabled. Let us explain how the different patterns of E-LOTOS can be used to represent different timed constraints. If we use the pattern ?t, then variable t will be bound to the time at which the action happens, and the selection predicate is which imposes the restriction (t<5); but if we use the pattern !5, then the value 5 is compared with the time at which the action happens, and the action will be possible only if this time is also 5.

The three parameters of a gate event are optional. When any of them is not present, a default value is used. So, the pattern P_1 is (), an empty record, by default. The pattern P_2 is **any**:time, that means any value of type time. The expression E is *true*, i.e., no conditions are required.

There is a special action, the internal action **i**, which will be studied in Section 2.4. Intuitively, it represents an action made by the process without the knowledge of its observer.

2.2 Sequential composition

We can compose two behaviours in sequence with the sequential composition operator ";". B_1;B_2 behaves first as B_1. When B_1 has finished then it continues as B_2. A sequential composition B_1;B_2 finishes when B_2 does.

With this new operator we can specify a behaviour with two gates, *inP* and *outP*, which receives an integer on gate *inP*, saves it in the integer variable x, and then sends this integer through gate *outP*:

```
inP(?x); outP(!x)
```

The sequential composition operator has greater precedence than any other binary operator. Thus, when we write B_1;B_2 [] B_3 (where [] is the selection operator, see next section) we mean (B_1;B_2) [] B_3. Anyway, we can use parentheses in order to clarify behaviours or to force precedence. Binary operators are right associative, so B_1;B_2;B_3 means B_1;(B_2;B_3).

Although the complete syntax of process declarations will be fully covered in Section 2.20, we are going to use it in examples to make them easier to understand. For the time being, it is enough to say that in a process declaration the *formal* gates via which the process can communicate are declared (like formal parameters are in a procedure declaration in Pascal) and also its behaviour (like a procedure body). In the process instantiation the *actual* gates are given.

Let us declare a process whose behaviour is as in our last example:

```
process Buffer1 [inP:int,outP:int] is
  var x:int in
    inP(?x); outP(!x)
  endvar
endproc
```

Note that in gate declarations each gate is typed with the type of values the gate can communicate.

Processes may be recursive. In this way, a process that is continuously receiving integers through a gate and sending them through another may be specified as

```
process Buffer2 [inP:int,outP:int] is
  var x:int in
    inP(?x:int); outP(!x); Buffer2[inP,outP]()
  endvar
endproc
```

In a process instantiation, the lists of actual gates and actual parameters (a process can also have a list of parameters) have to be given, although any of them may be empty. Thus, we have to write here Buffer2[*inP*,*outP*]() instead of only Buffer2[*inP*,*outP*].

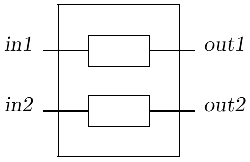
The behaviour B_1;B_2 lets time pass if B_1 does, or if B_1 has finished and B_2 lets time pass.

For LOTOS users, we have to note that LOTOS has two sequential composition operators: the action prefix operator (";") and the enabling operator (">>"). E-LOTOS has unified both of them in the sequential composition operator ";" we have just explained, and there is a particular case where the left behaviour consists of an action, for example *outP*(!3);Buffer2[*inP*,*outP*]().

2.3 Selection operator

The selection operator "[]" denotes a choice between two possible behaviours. In this way, the behaviour B_1 [] B_2 (where B_1 and B_2 are also behaviours) may perform actions either from B_1 or from B_2. The choice is solved (in principle) when B_1 [] B_2 interacts with its environment, which is defined by another behaviour. If the environment offers a first action of B_1, then B_1 is chosen and B_2 is forgotten. Otherwise, if the environment offers a first action of B_2 then B_2 is chosen and B_1 is forgotten. We see below what happens if the action offered by the environment is a first action of both B_1 and B_2.

For example, let us suppose we want to specify the two-position register that we introduce in Section 1.1, graphically described by



which can communicate with its environment through the gates *in1*, *in2*, *out1*, and *out2*. The register is initially empty and both data inputs are needed before there is output. In addition, data through gate *in1* has to arrive before data through gate *in2*; and data have to be output in the same order. The following process describes this behaviour

```
process Register1 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1); in2(?x2); out1(!x1); out2(!x2)
  endvar
endproc
```

where data is the type of the values the register can save.

But if the first value can be output without waiting for the second value, then we can specify the register behaviour with the selection operator as

```
process Register2 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1);
    ( in2(?x2); out1(!x1)
     []
      out1(!x1); in2(?x2)
    );
    out2(!x2)
  endvar
endproc
```

The E-LOTOS grammar forces the user to put parentheses where a behaviour is ambiguous,[4] so the user cannot mix binary operators without parentheses. Too many parentheses may lead to unreadable specifications, so E-LOTOS also has a *bracketed* syntax for these operators, suggested by Ed Brinksma in his thesis [Bri88]. The bracketed syntax for the selection operator is as follows:

$$\textbf{sel } B_1 \,\texttt{[]}\, \ldots \,\texttt{[]}\, B_n \textbf{ endsel}$$

where if some B_i use a binary operator (but ;), it has to use it with parentheses.

Nondeterministic choice

If we have the behaviour $B_1 \,\texttt{[]}\, B_2$ and the environment offers a first observable action of both, then the process is chosen nondeterministically. To show this let us see an example with vending machines.[5] Suppose we have a machine (Machine 1) whose behaviour is specified as

$$\text{Machine 1} \quad \equiv \quad (\ \textit{insert10};\ \textit{take\_coffee}\ )\ \texttt{[]}\ (\ \textit{insert5};\ \textit{take\_milk}\ )$$

This machine offers to the client (its environment) the choice between inserting a 10 unit coin or a 5 unit coin. If the client inserts a 10 unit coin then the machine evolves to a state where only coffee is offered, but if the client inserts a 5 unit coin, then the machine only offers milk. So, the client can (indirectly) choose coffee or milk by inserting the right coin.

Let us suppose now that we have another machine (Machine 2) that is like the last one but serves a more expensive milk, costing 10 units. If we modify (the behaviour of) Machine 1 getting

$$\text{Machine 2} \quad \equiv \quad (\ \textit{insert10};\ \textit{take\_coffee}\ )\ \texttt{[]}\ (\ \textit{insert10};\ \textit{take\_milk}\ )$$

then we will not obtain the desired behaviour. Now the client has to insert a 10 unit coin independently of what he wants. Once the client has inserted the coin, Machine 2 evolves to one of two possible states: one in which it only offers coffee, or one in which it only offers milk, and the client can take only what the machine offers. To which state the machine evolves depends on a nondeterministic choice. It is said that the machine has evolved *internally* (without knowledge of the environment) to one of these states. Therefore, Machine 2 is nondeterministic.

So, if we want to specify a machine that behaves as Machine 1, that is it offers coffee and milk after the client has inserted a coin but with the more expensive milk, we have to write instead:

$$\text{Machine 3} \quad \equiv \quad \textit{insert10};\ (\ \textit{take\_coffee}\ \texttt{[]}\ \textit{take\_milk}\ )$$

Once the client has inserted a coin, Machine 3 offers both coffee and milk, so the client can choose what he wants.

Another way of introducing nondeterminism is with internal actions, **i**, which we will present in the next section.

[4] The sequential composition operator is an exception to this rule, and this is the reason why we have said above that this operator has the greatest precedence.

[5] This example has been extracted from [LFHH90].

Variables and selection operator

Regarding variables, we have to say that in a selection B_1 [] B_2, variables which are modified only by B_1 or only by B_2 must be initialized, that is, they have to be given a value, before the selection, and the modification must preserve the type of the variables. In this way it can be assured that independently of the behaviour executed, the set of initialized variables after the selection is always the same, and we know their types. Variables which are modified by both behaviours may have been initialized previously or not, but both behaviours have to assign them a value of the same type.

Thus, the behaviours

```
inP(?x:int);
( inP(?x:bool)                            inP(?x:int)
 [] outP(!x)           and             [] inP(?x:bool)
)
```

are not allowed, because:

- in the first one, although variable x is initialized before the selection operator by means of the behaviour *inP*(?x:int), only one of the subbehaviours of the selection operator, *inP*(?x:bool), modifies it, and it changes the type of x;
- in the second one, both subbehaviours, *inP*(?x:int) and *inP*(?x:bool), modify variable x, but they do not give it the same type.

Note that although a variable has been declared, it may have been declared of the universal type **any** (see Section 3.2), so it can have values of any type.

Time determinism

Let us consider the following behaviour[6]

```
(?x:=5 [] ?x:=2); wait(1);P[...]
```

It is time nondeterministic: intuitively, when one unit of time has passed and process P is executed, variable x may have either the value 5 or the value 2.

Time nondeterminism is an undesirable feature, and E-LOTOS forbid it requiring that when we build a behaviour B_1 [] B_2, both behaviours B_1 and B_2 have to be "guarded," that is they have to perform an action on a gate (or raise an exception as we will see in Section 2.15) before they finish; this action decides which branch of the selection operator is selected.

The behaviour B_1 [] B_2 lets time pass if both B_1 and B_2 do so.

2.4 Internal action

The internal action, **i**, is an action that a process can carry out to evolve in an autonomous manner, without being observed by the environment. Together with the selection operator it is used to model nondeterminism. It is also used to represent hidden actions, as we will see in Section 2.14.

[6] We will describe the **wait** instruction in Section 2.16, and assignment in Section 2.19

Let us suppose now that our two-position register may lose information, that is, it is possible that data is accepted but then lose. So, once data has got in there are two possible cases: the data is offered or the data is lost, missing the chance of taking it out. We may describe this behaviour as follows

```
process Register3 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1:data);
    sel
      in2(?x2:data); out1(!x1)
    []
      out1(!x1); in2(?x2:data)
    []
      i; in2(?x2:data)
    endsel;
    sel out2(!x2) [] i endsel
  endvar
endproc
```

although we will see below (at the end of this section) that it is not right enough (due to urgency of internal action).

Once the first value has been accepted, there are three cases:

1. the second value is accepted and then the first is output,
2. the first value is output and then the second is accepted, or
3. the first value is lost and then the second is accepted.

It seems that the case where the second value is accepted and then the first value is lost lacks. But since losing a value is not observable by the environment, it is the same whether the first is lost before or after the second is accepted.

Nondeterministic behaviours

As noted above, internal actions can be used to specify nondeterminism because they can be executed in an autonomous way. The behaviour

coffee [] *milk*

offers the possibility of taking either coffee or milk. However, the behaviour

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

always offers coffee but only *may* offer milk. That is, if the environment asks for synchronization on gate *milk*, the synchronization could be accepted or not happen, in a nondeterministic way. Instead, if the client asks for synchronization on gate *coffee*, he always will succeed.

On the other hand, the behaviour

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

may not be able to synchronize on any particular gate, depending on the internal choice.

Urgency

The internal action **i** is *urgent*, that is it cannot idle. This means that when an internal action can be performed, it *must* be performed unless another action is performed without delay. So the process Register3 we have written above does not have the meaning we want. Due to the urgency of the internal action, if actions on gate *in2* or *out1* are not possible immediately by the environment, then the internal action must be carried out. So the behaviour specifies that if the second value is not accepted immediately or the first value is not output immediately, then the first one is lost, which is not the desired behaviour. In the behaviour

```
( i;coffee ) [] milk
```

if the environment wants to synchronize after some period of time (greater than 0) on gate *milk*, it will never have success.

We can specify the desired behaviour by introducing a nondeterministic internal choice between losing the value or not losing the value. The right Register4 process is:

```
process Register4 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1:data);
    ( i;(  in2(?x2:data); out1(!x1)
          [] out1(!x1); in2(?x2:data)
        )
      []
        i; in2(?x2:data)
    );
    (  i; out2(!x2)
      [] i
    )
  endvar
endproc
```

2.5 Successful termination

Successful termination in E-LOTOS is represented by the behaviour **null**. This behaviour terminates immediately (it is urgent), without doing anything.

2.6 Inaction and time block

There are two behaviours in E-LOTOS for representing anomalous processes: **stop** and **block**. Both behaviours do not do any communication, and they do not terminate either. The difference is that **block** will prevent the progress of time, whereas **stop** can delay for any time. Both represent a *pathological* behaviour, a deadlocked process, that is, an undesirable specification, a *broken* system. Therefore, they should not be used in a correct specification in parallel with other behaviours.

For example, the behaviour

```
stop || wait(1);outP(!3)
```

can perform a communication on gate *outP* after one unit of time, but it cannot finish (as we will see in the next section). However, the behaviour

```
block || wait(1);outP(!3)
```

cannot do anything, because it cannot idle one unit of time, since **block** *freezes* time.

The behaviour **block** can be defined with help of the urgency of the internal action:

```
process Block is
    i;Block[]()
endproc
```

2.7 Parallel composition operator

We are going to study in this section, and the following ones, five different ways of composing processes in parallel.

The first possibility is using the parallel operator, whose syntax is

$$B_1 \ |[\ G_1,G_2,\ldots,G_n\ ]|\ B_2$$

where B_i stands for behaviours and G_j for gates.

The behaviour B_1 |[$G_1,\ldots,G_n$]| B_2 offers observable actions of both B_1 and B_2 as long as they are not actions carried out on a gate in the given list $G_1, G_2, \ldots, G_n$. In order to carry out an action in the given list, both behaviours have to perform it simultaneously. If one behaviour can communicate on a gate in the list, but the other cannot, then the first one has to wait for the other, if it can. When both behaviours offer an action on the same gate, then communication will be possible if the patterns associated with both actions match (see Section 3.6). In such a case the whole behaviour will offer the action on the common gate, and once this happens both behaviours will continue their way until the next synchronization. This kind of synchronization is called *multi-way synchronization*, and it is useful when more than one behaviour have to synchronize on a common gate.

To show the use of this operator, we are going to add components to the register example, introducing two new processes: a producer that fills the positions and a consumer that empties them. These processes could be

```
process Producer [p1:data,p2:data] is
    (* produce first value *)
    p1(!val1);
    (* produce second value *)
    p2(!val2)
endproc
```

where *val*1 and *val*2 are two constants of type data, and

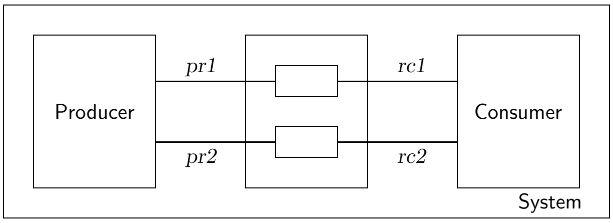
```
process Consumer [c1:data,c2:data] is
  var v1:data,v2:data in
    c1(?v1);
    (* consume first value *)
    c2(?v2)
    (* consume second value *)
  endvar
endproc
```

In E-LOTOS, comments are enclosed within (* and *).

Now we can put together these processes and the register to model the complete system. Our consumer only receives two values and is not prepared for loss, so we use the register version that does not lose values.

We use gates *pr1* (producer-register 1), *pr2*, *rc1* (register-consumer 1), and *rc2* to represent the communication between the processes. These gates, and the communications carried on them, are not interesting outside the *complete* system, so the environment cannot synchronize on them.

The system may be drawn as



The complete E-LOTOS system is:

```
process System1 is
    hide pr1:data,pr2:data,rc1:data,rc2:data in
        conc
            Producer [pr1,pr2]()
        |[pr1,pr2]|
            Register2 [pr1,pr2,rc1,rc2]()
        |[rc1,rc2]|
            Consumer [rc1,rc2]()
        endconc
    endhide
endproc
```

So the register must synchronize on its *input* actions (*pr1* and *pr2*) with the producer, and must synchronize on its *output* actions (*rc1* and *rc2*) with the consumer. Here we have used the bracketed syntax of the parallel composition operator,

$$\textbf{conc}\ B_1\ \texttt{|[...]|}\ \ldots\ \texttt{|[...]|}\ B_n\ \textbf{endconc}$$

(where the parallel composition operator is right associative, like any other binary operator) and the **hide** operator that, intuitively, is used to show that the gates that connect the register with the producer and the consumer are not observable for someone that sees the whole system. We will discuss this operator in Section 2.14.

In the behaviour B_1 |[G_1, ... ,G_n]| B_2, B_1 and B_2 must assign disjoint global variables. There is no *share memory* that processes running in parallel can use. Communication has to be explicit, that is, B_1 and B_2 can communicate only through gates $G_1, \ldots, G_n$. Besides, if both behaviours assigned the same variable, we would not know the last value when the whole behaviour finished.

The behaviour

$$B_1\ \texttt{|[}\ G_1\texttt{,}\ \ldots\ \texttt{,}G_n\ \texttt{]|}\ B_2$$

finishes when B_1 and B_2 do. It is said that both behaviours B_1 and B_2 must synchronize on their termination.

B_1 |[G_1, ... ,G_n]| B_2 lets time pass if both B_1 and B_2 do so, or if one of them has finished (and it is waiting for the other) and the other lets time pass.

2.8 Interleaving operator

When the list of gates on which two parallel processes have to synchronize is empty, E-LOTOS has an abbreviated form of the parallel operator, the *interleaving* operator, "`|||`"

$$B_1 \;\texttt{|||}\; B_2 \quad = \quad B_1 \;\texttt{|[]|}\; B_2.$$

As a consequence, this operator merges of the actions of both B_1 and B_2 in such a way that the actions belonging to each one of them remain in the same order.

As a matter of fact, there is an occasion when the processes have to synchronize: termination. As for the previous parallel operator, the interleaving B_1 `|||` B_2 of B_1 and B_2 finishes when both processes do it.

We may use this operator to model, in an easier way than we did before, the fact that the register can receive the second value before or after the first one is output:

```
process Register5 [in1:data,in2:data,out1:data,out2:data] is
  var x1 : data,x2:data in
    in1(?x1:data);
    (  in2(?x2:data)
      |||
       out1(!x1)
    );
    out2(!x2)
  endvar
endproc
```

The bracketed syntax of the interleaving operator is as follows:

$$\textbf{inter}\; B_1 \;\texttt{|||}\; \ldots \;\texttt{|||}\; B_n \;\textbf{endinter}$$

Being an abbreviation, the interleaving operator has the same features as the parallel operator regarding variables and time.

2.9 Synchronization operator

This operator is used when the two behaviours composed in parallel have to synchronize on every observable action. They also have to synchronize on the termination. Its unbracketed syntax is

$$B_1 \;\texttt{||}\; B_2$$

and the bracketed syntax is

$$\textbf{fullsync}\; B_1 \;\texttt{||}\; \ldots \;\texttt{||}\; B_n \;\textbf{endfullsync}$$

Note that

$$B_1 \;\texttt{||}\; B_2 \quad = \quad B_1 \;\texttt{|[}\; \text{“all gates of } B_1 \text{ and } B_2\text{”} \;\texttt{]|}\; B_2.$$

Using both this operator and the interleaving operator we can specify again the complete system that puts together the producer, the consumer and the register. Producer and consumer have no common gate, so their parallel evolutions are independent, and we can put them together using the interleaving operator. The resulting behaviour has to synchronize on all gates with the register, so we compose them with the synchronization operator, in the following way:

```
process System2 is
  hide pr1:data,pr2:data,rc1:data,rc2:data in
      (  Producer [pr1,pr2]()
        |||
          Consumer [rc1,rc2]()
      )
    ||
      Register5 [pr1,rc1,pr2,rc2]()
  endhide
endproc
```

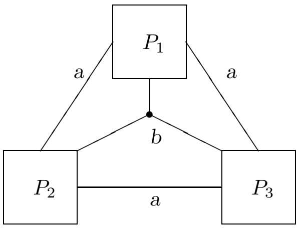
Regarding assignments of variables and time, the synchronization operator behaves like the parallel operator.

2.10 General parallel operator

The three parallel operators we have seen ("||", "|||", and "|[...]|") are not easy to use when we want to describe some possible networks of communicating processes. Let us suppose we have a set of m processes, $P_1, P_2, \ldots, P_m$, and we want to specify a synchronization scheme where, in order to execute any action on a given gate, we need the collaboration of *any collection* of n of these processes. When $n = 2$ this means that any process P_i can communicate on that gate with any other process P_j $(i \neq j)$.

It is not easy to specify in E-LOTOS this kind of "n among m" synchronization using only the kind of operators we have already introduced. To solve this problem, in E-LOTOS a new generalized parallel operator has been introduced, which directly expresses networks of processes.

With this new operator we can specify, in an easy way, systems like the one graphically represented by



where on gate a two processes have to synchronize, and on gate b all the three processes have to synchronize, in the following way:

```
par a#2 in
      [a,b] -> P1
   || [a,b] -> P2
   || [a,b] -> P3
endpar
```

i.e. pairs of processes may synchronize on gate a, but all processes synchronize on gate b.

The syntax of this new operator is:

```
par $G_1$#$n_1$, ... ,$G_p$#$n_p$ in
   [$\Gamma_1$] -> $B_1$
|| ...
|| [$\Gamma_m$] -> $B_m$
endpar
```

where n_i's are positive natural numbers, $p \geq 0$ and Γ_i's are gate lists $\Gamma_i = G_{i1}, \ldots, G_{ir_i}$.

The intuitive meaning of this operator is as follows: the whole behaviour can perform an action on a gate G, if:

- There is a component B_i such that gate G is not in its *synchronization list* Γ_i which performs it; or
- There are some components synchronizing in the following way:
  - if gate G is in the *list of degrees* (G_1#n_1, ... ,G_p#n_p) with degree n, then n whatever components B_i such that $G \in \Gamma_i$ synchronizes on G.
  - if gate G is not in the list of degrees, then all the behaviours B_i having G in their synchronization list Γ_i synchronizes on it.

The behaviour finishes when all the behaviours B_i have finished.

Regarding variables and time, the general parallel operator behaves as the other parallel operators: all the behaviours B_i have to assign pairwise disjoint variables, and it lets time pass if all B_i do, or if some of them have finished and all the unfinished ones let time pass.

2.11 Parallel over values operator

Sometimes we have a behaviour that may depend on a variable, and we want to put together, running in parallel, some instantiations of this common behaviour, each one with a different value of this variable. For example, we may have a process that describes the behaviour of a node of a network which is identified by a unique identifier, and we may want to build a network of several nodes, each one with a different identifier.

We can specify this kind of behaviour in E-LOTOS with the parallel over values operator. Its syntax is

par P **in** N ||| B **endpar**

where P is a pattern, N is a list of values (the type of N has to be List, a predefined type of E-LOTOS, see Section 3.1) these values can be matched against P, and B is a behaviour that may depend on the variables in P. The represented behaviour is the interleaving of a series of instantiations of B, one for each value of N. For example, if we have a process Node[...](id:IdType, ...), we can specify a network of five nodes with different identifiers in the following way:

```
par ?$x$ in [1,2,3,4,5] |||
  Node[...]($x$, ...)
endpar
```

In other parallel operators, we have said that it is not allowed that two behaviours composed in parallel assign to the same variable. Here, it is the same behaviour which is used as a template to make several instantiations. If this template behaviour assigned to a global variable, then the different instantiations would do, and we would have several behaviours in parallel assigning to the same global variable. In order to forbid this situation, the template behaviour in a parallel over values cannot assign any global variable. Of course, it can assign local variables declared inside the template behaviour. In this way, each instantiation would have its own local variables.

Regarding time, the parallel over values operator lets time pass if all the instantiations do, or if some of them have finished and all the unfinished instantiations let time pass.

2.12 Disabling operator

The disabling operator "[>" models permanent interruption of a process by another. So the behaviour B_1 [> B_2 models the fact that at any point during the execution of B_1 there is a choice between doing a next action of B_1 or a first action of B_2. Once an action of B_2 is carried out, B_2 continues evolving and the remaining behaviour of B_1 is forgotten. But if B_1 finishes without interruption, the whole behaviour finishes and B_2 is not able to interrupt any more.

Coming back to the register example, we can use the disabling operator to model the fact that the register fails after 50 units of time if it has not yet finished its "normal" behaviour:

```
process Register6 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    ( in1(?x1);
    ( in2(?x2) ||| out1(!x1) );
    out2(!x2) )
    [> ( wait(50); i )        (* the register fails *)
  endvar
endproc
```

where we have used the internal action **i** to represent the *internal* decision of the register to fail, and the **wait** operator (Section 2.16) to express when the register fails.

The disabling operator also has a bracketed syntax which is as follows:

$$\textbf{dis } B_1\,\texttt{[>}\ldots\texttt{[>}B_n \textbf{ enddis}$$

Regarding variables, the disabling operator behaves like the selection operator. That is, in the behaviour B_1[>B_2, when only B_1 or B_2 modifies a variable it must be initialized before B_1[>B_2; and if a variable is modified by both of them, they have to assign a value of the same type.

In order to keep internal action urgent and time deterministic, behaviour B_2 is required to be guarded in the disabling behaviour B_1 [> B_2. For technical reasons, B_1 is not required to be guarded.

The behaviour B_1[>B_2 lets time pass if both B_1 and B_2 do.

2.13 Suspend/Resume operator

The suspend/resume operator "[X>" is an extension of the disabling operator (Section 2.12) which allows the resumption of the interrupted behaviour. If during the evolution of the behaviour B_1 [X> B_2 B_1 is interrupted by B_2, then B_1 is suspended until B_2 resumes it through exception X. Then B_1 continues its evolution with the possibility of being interrupted again. B_2 is always restarted after a resumption.

Although exceptions and their handling will be seen in Section 2.15, it is enough to know now that an exception X can be raised with **signal** X.

Going on with our register example, if it can recover itself after an unexpected failure and continue as nothing wrong has happened, then we could specify it as follows:

```
process Register7 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    ( in1(?x1);
    ( in2(?x2) ||| out1(!x1) );
    out2(!x2) )
    [Ok> ( wait(50); i; Repair[ ](); signal Ok )
  endvar
endproc
```

With this operator we can specify more complex interruption mechanisms, where, for example, a behaviour controls the evolution of others. Let us consider the next process

```
( ( B1 [cont1> stop1; start1;signal cont1 ) ||| (B2 [cont2> stop2; start2;signal cont2 ) )
|[stop1, stop2, start1, start2]|
B3
```

The behaviour B_3 controls B_1 and B_2 through gates *stop1*, *start1*, *stop2*, and *start2*. B_3 can stop the evolution of B_1 using the gate *stop1* and can restart it using *start1*. Using instead *stop2* and *start2* it can control B_2.

Regarding variables and time, the suspend/resume operator behaves like the disabling operator.

2.14 Hiding operator

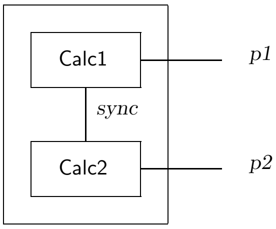
The hiding operator allows the abstraction of the internal operation of a process. It hides those actions that are considered internal to it, and which are considered of no interest at a certain level of detail. To do it, the hiding operator transforms observable actions into internal actions, hiding them from the environment, and making them urgent actions as we will see below.

The syntax of the hiding operator is

```
hide G1[:T1], ... ,Gn[:Tn] in
  B
endhide
```

where the G_i's are gates declared in B by the **hide** operator, so they can be used only by B. Types T_i are **any** by default, that is, if we do not declare the type of a gate, it can communicate values of any type.

Let us suppose that the producer used before is composed of two *subprocesses*, each of which produces a value.



In order to produce the values in the desired order, the components have to synchronize, but this synchronization (or the way it is done) is not interesting from the point of view of the whole system, so we hide it. Using the **hide** operator the producer would become

```
process Producer2 [p1:data,p2:data] is
    hide sync in
        Calculation[p1, sync](1)
      |[ sync ]|
        Calculation[p2, sync](2)
    endhide
endproc
```

where we have used two different instantiations of the same process Calculation, given by

```
process Calculation [pd:data,sync](turn:int) is
    case turn is
          !1 -> pd(!val1); sync
        | !2 -> sync; pd(!val2)
    endcase
endproc
```

where *val*1 and *val*2 are two constants of type data.

The process Calculation has a parameter *turn* of type int which is used to distinguish whether the process has to communicate the calculated value either before or after it synchronizes on gate *sync*. The **case** operator (Section 3.6) is used to define conditional behaviours. In this case, if *turn* is equal to 1 then the process first communicates the value and then offers a synchronization on gate *sync*, and if *turn* is 2 the process first synchronizes and then communicates its value.

Urgency

The behaviour **hide** $G_1 : T_1, \ldots, G_n : T_n$ **in** B **endhide** lets a time d pass if B lets time d pass and B cannot offer a communication on a hidden gate $(G_1, \ldots, G_n)$ in a time less than d. So, the **hide** operator transforms the actions on hidden gates into *urgent* actions, that is, they are performed as soon as they are enabled. Thus, when we hide the gate *sync* in the example above, we achieve two aims: first, this action cannot be observed outside the producer, and, second, communication on that gate must be performed as soon as possible, because *sync* is an urgent action. We will see another example in Section 2.16.

2.15 Exceptions and their handling

Exceptions are nowadays recognized as a desirable programming feature for dealing with errors and other abnormal situations. Concerning parallel languages, the importance of exceptions combined with concurrency has been pointed out by Berry in relation to the language ESTEREL [Ber93]. That proposal was adapted to the framework of process algebras by Nicollin and Sifakis in their "Algebra of Timed Processes," ATP [NS94]. However, none of the previously standardized Formal Description Techniques (SDL, LOTOS, and ESTELLE) supports exceptions. In E-LOTOS, exceptions and their handling have been introduced through the **trap** operator (handling) and **raise** and **signal** instructions (raising).

Beginning with the **trap** operator, its syntax is:

```
trap
  exception $X_1$[($P_1$)[:$T_1$]] is $B_1$ endexn
    ...
  exception $X_n$[($P_n$)[:$T_n$]] is $B_n$ endexn
  [exit [$P_{n+1}$] is $B_{n+1}$ endexit]
in $B$
endtrap
```

where B stands for the "normal" behaviour and $X_1, \ldots, X_n$ are the exceptions that can be raised from B. Each exception X_i can be raised together with a value of type T_i (which is () by default) that will be trapped with the pattern P_i (which is () by default); and will be handled by behaviour B_i that defines how the behaviour evolves after the exception has been raised (and trapped). If the **exit** clause is present, when the behaviour B finishes and returns a value, this value is matched against the pattern P_{n+1} and the behaviour B_{n+1} is executed.

Taking the words of H. Garavel in [GS96], the **trap** operator behaves like a "watchdog." The normal behaviour B evolves until it raises an exception X_i. Then, B is aborted and the exception handler associated with X_i starts its execution, once the value raised together with the exception have been assigned to the corresponding pattern.

An exception may be raised with the **signal** or **raise** instructions whose syntax is

signal X [(E)]

raise X [(E)]

These are quite similar. Both raise an exception (possibly with associated values E), and if the exception is trapped then there is no difference. But if the exception is not trapped,[7] then the behaviour after a **signal** instruction can be executed, whereas if the exception has been raised with a **raise**, then the whole behaviour is blocked. In fact, **raise** X(E) is syntactic sugar for **signal** X(E); **block**.

In [GS96] some algebraic properties of the **trap** operator are shown, together with the relationship to other operators which are derived from it.

We can use exceptions for example when our register fails. If it cannot recover itself, it finishes raising an exception *Error*.

```
process Register8 [in1:data,in2:data,out1:data,out2:data] raises [Error] is
  var x1:data,x2:data in
     ( in1 ?x1:data;
       ( in2 ?x2:data ||| out1 !x1 );
       out2 !x2 )
     [> ( wait(50); i; signal Error )
  endvar
endproc
```

Note how it is indicated with **raises** [*Error*] that this process can raise an exception.

If the producer receives the message that an error has occurred through gate *anError*, then we can handle exception *Error* with a communication with the producer through this gate:

```
   Producer3[pr1,pr2,anError]()
  |[pr1,pr2,anError]|
```

[7] In order to raise an exception X, it must be declared as an exception. So we might think that we can only raise exceptions declared in a **trap**, but as we will see in Section 4.4 a specification can also declare exceptions that are not trapped and are propagated to the top level, that is, to the level of the environment of the system.

```
trap
  exception Error is anError endexn
in
  Register8[pr1,pr2]()[Error]
endtrap
```

2.16 Delay instruction

We have already seen how to make gates sensitive to the time at which events occur (see Section 2.1). Now we present another construction related to time: the delay operator **wait**.

The behaviour

$$\textbf{wait}(E)$$

is idle while the time indicated by the expression E (which must have type time) passes, and then it finishes.

For example, suppose our register has a delay such that a received value may not be output before three units of time have passed since it was received. We could specify it as

```
process Register9 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    hide canGetIn2 in
         in1(?x1); CanGetIn2; wait(3); out1(!x1)
       |[canGetIn2]|
         canGetIn2; in2(?x2) ; wait(3); out2(!x2)
    endhide
  endvar
endproc
```

By using the **hide** operator and a synchronization on gate *canGetIn2*, we have described in a different way that data received through gate *in1* has to get in before data through gate *in2*. In this way it is easier to specify that three units of time will pass before the corresponding value is output.

If we do not know exactly the delay of the register, but we know a lower bound *min* on this delay, we can specify that the time between input and output is not less than this value:

```
in1(?x1);
?t := any time [min <= t]; wait(t);
out1(!x1)
```

where we have used a nondeterministic assignment, whose meaning will be explained in Section 2.19. We have that once a value has been accepted through gate *in1*, an arbitrary value of type time greater than or equal to *min* is assigned to variable *t*. We have to wait for this time, and then a communication on gate *out1* is offered.

Let us see now another simple, classical example[8] which uses both a time annotation on actions and the **wait** operator. We have a sender process that, after having transmitted a message, waits for an acknowledgment. If this acknowledgment does not arrive during a certain time then the sender retransmits the same message; otherwise, it waits for another message to be transmitted. The process that specifies the sender behaviour is as follows:

[8] This example has been adapted from [LL94].

```
process Sender1 [ req:message, trans:message, ack ] is
  var m:message in
    req(?m) ;
    Send_Message1 [ trans, ack ] (m) ;
    Sender1 [ req, trans, ack ]
  endvar
endproc
```

where we have defined

```
process Send_Message1 [ trans:message, ack ] (m:message) is
  var t:time in
    trans(!m) ;
    (  ack @?t  [ t<waiting_time ]
    [] wait(waiting_time) ; Send_Message1 [ trans, ack ] (m)
    )
  endvar
endproc
```

Note that if the acknowledgment does not arrive before *waiting_time* units of time have passed, the action on gate *ack* becomes impossible, and only a new transmission (of the same value) is possible (via the recursive call to Send_Message1).

Suppose that we want to separate the time constraints from the rest of the behaviour. We can have a process Timer that controls these constraints and interacts with the sender:

```
process Timer [ setT:time, reset, timeout ] is
  var t:time in
    setT(?t) ;
    (  reset
    [] wait(t) ; timeout
    ) ;
    Timer [ setT, reset, timeout ] ()
  endvar
endproc
```

and modified processes:

```
process Sender2 [ req:message, trans:message, ack, setT:time, reset, timeout ] is
  var m:message in
    req(?m) ;
    Send_Message2 [ trans, ack, setT, reset, timeout ] (m) ;
    Sender2 [ req, trans, ack, setT, reset, timeout ]
  endvar
endproc
```

and

```
process Send_Message2 [ trans:message, ack, setT:time, reset, timeout ] (m:message) is
  trans(!m) ; setT(!waiting_time) ;
  (  ack; reset
  [] timeout; Send_Message2 [ trans, ack, setT, reset, timeout ] (m)
  )
endproc
```

And then the whole sender could be like

```
  Sender2 [ req, trans, ack, setT, reset, timeout ]
|[setT, reset, timeout]|
  Timer [ setT, reset, timeout ]
```

The problem is that in this last behaviour, the synchronization on gates *setT*, *reset*, and *timeout* is forced, but it is not forced the time at which they occur. We have to hide these gates in order to make the actions on them urgent actions, that is, the synchronizations on these gates occur as soon as they are possible. Thus, the sender can be

```
process Sender3 [ req:message, trans:message, ack ] is
  hide setT:time, reset, timeout in
      Sender2 [ req, trans, ack, setT, reset, timeout ]
    |[setT, reset, timeout]|
      Timer [ setT, reset, timeout ]
  endhide
endproc
```

2.17 Renaming operator

In E-LOTOS we can rename actions by means of an explicit renaming operator which allows to rename observable actions into observable actions, and exceptions into exceptions, and also allows us to modify the offers of the actions performed by a behaviour.

For example, let us suppose we have two behaviours B_1 and B_2 which we cannot modify since they belong to a private library of processes. If we compose them in parallel and they have to communicate with each other, we have a problem if the messages sent by any of them, say B_1, are not in the same format as B_2 is expecting. For example, B_1 could send a message $G(!E)$ through the gate G but B_2 could expect messages of the form $G(!f(header, E, trailer))$, where f is a function to build packets from messages by adding them a header and a trailer. We can solve this problem by means of the renaming operator.

An explicit renaming operator is also useful for verification purposes, when it is desirable to rename all non-interesting (for the verification of the property to be checked) observable actions into some particular action, different from the internal action (so hiding cannot be used).

The syntax of this operator is as follows:

```
rename
  gate $G_1[(P_1)[:T_1]]$ is $G'_1[(P'_1)]$
    ...
  gate $G_m[(P_m)[:T_m]]$ is $G'_m[(P'_m)]$
  signal $X_1[(P''_1)[:T'_1]]$ is $X'_1[(E_1)]$
    ...
  signal $X_n[(P''_n)[:T'_n]]$ is $X'_n[(E_n)]$
in $B$
endren
```

This represents the behaviour B where the gates $G_1, \ldots, G_m$ and the exceptions $X_1, \ldots, X_n$ are renamed into $G'_1, \ldots, G'_m$ and $X'_1, \ldots, X'_n$, respectively. These gates and exceptions are declared by the **rename** behaviour, so they can be used only in B. P_i's are patterns used as follows: values associated to renamed gates or exceptions are bounded to variables in P_i's, and then these variables can be used in the corresponding new gates (patterns P'_i) or exceptions (expressions E_i).

The simplest renaming is the one that only renames one gate into another:

```
rename
  gate inP(?a):int is intro(!a)
in
  B
endren
```

In this way, if behaviour B offers a communication on gate *inP* of an integer, then this value is bound to a and the whole behaviour will offer a communication on gate *intro* of the same integer a.

However, gate renaming may be more powerful than a simple change of name. We can also use it to change the structure (type of values) of actions offered by a process. For example we can remove a record field:

```
rename
  gate inP ((x => ?a, y => ?b)):recordXY is inP(!(x => a))
in
  B
endren
```

where recordXY is a type defined by the user (in E-LOTOS anonymous types cannot be used) as

```
type recordXY is
  (x => int, y => bool)
endtype
```

Note that there are two gates called *inP*: the left one is the new declared gate that can be used in B; and the right one is a gate which have to be declared outside this **rename**, and cannot be used by B. If B performs the communication *inP* `(!(`x `=> 3,`y `=>` *true*`))` on (local) gate *inP*, the variable a will be bound to the value 3, and the variable b will be bound to the value *true*, and the whole rename behaviour will perform the communication *inP* `(!(`x `=> 3))` on (global) gate *inP*.

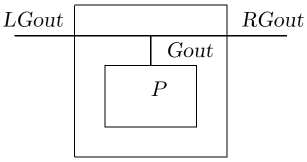
We can add a field:

```
rename
  gate inP ((x => ?a:int)):recordX is inP (!(x => a,y => true))
in
  B
endren
```

We can even change the values that are being communicated. For example, we can solve the problem at the beginning of this section in the following way

```
 rename
  gate G (?e):message is G(!f(header,e,trailer))
 in
  B1
 endren
||
 B2
```

We can also split a gate G into two gates G_1 and G_2, depending on the values G carries on. For example, let us suppose we have defined a process P that sends values through a gate *Gout*. It indicates whether they have to go to the right or to the left side, using actions like *Gout*`(!(`*left*`,`*dat*`))` or like *Gout*`(!(`*right*`,`*dat*`))` (where *left* and *right* are values of an enumerated type (see Section 3.1) and *dat* is a variable with a value of type data). Suppose we want to use process P in a context where we have two different gates, *LGout* to send values to the left, and *RGout* to send values to the right,



we can use the **rename** operator as follows:

```
rename
  gate Gout ((!left,?d:data)):any is LGout(!d)
  gate Gout ((!right,?d:data)):any is RGout(!d)
in
  P[Gout]()
endren
```

We can also do the opposite, that is, we can merge two gates $G1$ and $G2$ in a single gate G:

```
rename
  gate G1(?a):int is G(!(a,true))
  gate G2(?a):int is G(!(a, false))
in
  B
endren
```

Exceptions can be renamed in the same way. Let us imagine a process that raises a generic exception *Error* together with the number of the produced error. If at some time the association between numbers and errors is changed, and we have a function g that defines this change, we can rename the exception *Error* in order to vary its parameter as function g indicates:

```
rename
  signal Error (?e):int is Error (g(e))
in
  B
endren
```

2.18 Conditional operator

As in almost every programming language, E-LOTOS includes a conditional **if** – **then** – **else** operator. Its syntax is:

```
if E then B
  (elsif E then B)*
  [else B]
endif
```

where E must be a boolean expression. The behaviour associated with the first expression E that evaluates to *true* describes how the **if** – **then** – **else** behaves. The behaviour associated with the **else** clause is **null** by default, and therefore this is the behaviour when no expression E is *true*, and there is no else clause. For instance, we can use this instruction for communicating the absolute value of variable x through gate *outP*:

```
if x >= 0 then  outP(!x)
else  outP(!-x)
endif
```

2.19 Imperative features

In E-LOTOS several imperative features have been introduced in order to make the job of specifying systems easier to the user of this programming paradigm.

One of these features is assignment. E-LOTOS has *write-many* variables, that is, variables that can be assigned several times. The static semantics of the language forbids reading a variable unless it has a value. We can assign values to a variable by using the assignment instruction, whose syntax is

$$P \;\texttt{:=}\; E$$

where P is a pattern which can be pattern-matched against the value returned by expression E. The result of the assignment is the creation of a binding between the variable (or variables) in the pattern and the expression value. In this way, the assignment

$$\texttt{?}x \;\texttt{:=}\; 3$$

binds the value 3 to variable x. On the other hand, the assignment

$$\texttt{(?}x\texttt{, ?}y\texttt{) := (5, }\mathit{true}\texttt{)}$$

associates variable x with 5 and variable y with *true*.

There is another kind of assignment, the *nondeterministic assignment*, whose syntax is:

$$P \;\texttt{:=}\; \mathbf{any}\; T \;[\texttt{[}\; E \;\texttt{]}]$$

By means of it we bind the variables in pattern P with any value of type T that satisfies the condition E, which is *true* by default. For example, the assignment

$$\texttt{?}x \;\texttt{:=}\; \mathbf{any}\; \mathsf{int} \;\texttt{[}\; x \texttt{ < } 10 \;\texttt{]}$$

binds variable x to an integer less than 10.

Another imperative feature introduced in E-LOTOS is the possibility of declaring variables, using the **var** operator (as we have seen in several examples), in order to restrict the scope of variables, and therefore possibly hiding more global variables with the same name. The syntax of this operator is:

```
var V1:T1[:=E1], ... ,Vn:Tn[:=En] in
  B
endvar
```

where the declared variables $V_1, \ldots, V_n$ may be initialized with the corresponding expressions E_j.

The occurrences of the identifiers naming the declared variables in behaviour B refer to these new variables, hiding in the scope those variables with the same name that already exist. Changes of the values of these (local) variables do not modify in anyway the more general ones. On the other hand, they will not be visible outside the scope of the **var** operator.

For example, the behaviour

```
?x := 3;
var x : int in
  ?x := 5; ?y := x
endvar;
inP(!(x,y))
```

offers on the gate *inP* the pair (3,5).

As we said, declared variables may be initialized, and so the following behaviour is equivalent to the previous one:

```
?x := 3;
var x : int := 5 in
  ?y := x
endvar;
inP(!(x,y))
```

A variable V declared of type T can be assigned a value of any subtype T' of T.

E-LOTOS has also several iterative operators: **loop**, **while**, and **for**. The **loop** operator represents an infinite loop whose execution can only be stopped raising an exception with the instruction **break**. It has two versions. The first one, whose syntax is

loop B **endloop**,

represents a loop where the behaviour B is continuously executed until the predefined exception *inner* is raised (by means of **break** command, see below). The second version has syntax

```
loop X [: T] in
  B
endloop
```

and represents a *breakable* loop that can be interrupted with the exception X.

We raise this exception[9] with a **break** command, with syntax

break [X [(E)]]

where X is the name of the exception associated with the loop we can stop. (If it has no name, the default *inner* exception will be used.) E, the value associated with the exception, represents the result that the loop returns.

Note that exception X is declared together with the loop, and so we do not need to introduce an explicit handler for it. The handling of this exception consists of breaking (stopping) the loop, and returning the associated value, if any.[10]

For example, using the **loop** operator we can define the recursive (infinite) process that continuously receives and sends integers:

```
process Buffer3 [inP:int,outP:int] is
  var x:int in
    loop
      inP(?x); outP(!x)
    endloop
  endvar
endproc
```

[9] And also the predefined *inner* exception, but in this case no name has to be given.

[10] In the definition of the semantics [Que98] we see that a **trap** is included.

On the other hand, the buffer that receives and sends integers until it receives a 0 can be specified with a breakable **loop**:

```
process Buffer4 [inP:int,outP:int] is
  var x:int in
    loop End in
      inP(?x); outP(!x)
      if x=0 then break End endif
    endloop
  endvar
endproc
```

E-LOTOS has also a *conditional* loop, the **while** instruction, whose syntax is:

```
while E do
  B
endwhile
```

where E is a boolean expression, and B is a behaviour which is executed until the expression E evaluates to *false*. For example, we can add all the elements in a list of integers, xs, with type **list of** int (see Section 3.1), and send the result through a gate *outP* as follows:

```
var total : int := 0,e:int in
  while not(isempty(xs)) do
    ?e := head(xs);
    ?total := total + e;
    ?xs := tail(xs)
  endwhile;
  outP(!total)
envar
```

where we have used some functions for manipulating lists, to be described in Section 3.1.

The last iterative construction of E-LOTOS is the **for** loop, whose syntax is:

```
for E1 while E2 by E3 do
  B
endfor
```

It represents the execution of the expression E_1 and then, while expression E_2 evaluates to *true*, the execution of behaviour B followed by the evaluation of expression E_3. Thus, the **for** loop is just syntactic sugar for

```
E1;
while E2 do
  B;E3
endwhile
```

2.20 Process declaration and instantiation

We can use process declarations to give a name to a behaviour and to abstract the names of actual gates, parameters, and exceptions. Thus, processes are parameterized by a list of formal gates, a list of formal variables (parameters), and a list of formal exceptions.

The syntax of a process declaration is as follows:

$$\begin{array}{ll}\textbf{process}\ \mathsf{P} & [\ \texttt{[}\ G_1[\texttt{:}T_1]\texttt{,}\ldots\texttt{,}G_n[\texttt{:}T_n]\ \texttt{]}\ ] \\ & [\ \texttt{(}\ [\textbf{in}|\textbf{out}]V_1\texttt{:}T'_1\texttt{,}\ldots\texttt{,}[\textbf{in}|\textbf{out}]V_m\texttt{:}T'_m\ \texttt{)}\ ] \\ & [\textbf{raises}\ \texttt{[}\ X_1[\texttt{:}T''_1]\texttt{,}\ldots\texttt{,}X_p[\texttt{:}T''_p]\ \texttt{]}] \\ \quad\textbf{is}\ B & \\ \textbf{endproc} & \end{array}$$

where

- P is a process identifier;
- $\texttt{[}\ G_1[\texttt{:}T_1]\texttt{,}\ldots\texttt{,}G_n[\texttt{:}T_n]\ \texttt{]}$ is the list of gates the process uses, which is empty (`[]`) by default;
- T_i is the type of gate G_i, that is, the type of values gate G_i can communicate. It is **any** by default, that is, a gate can communicate values of any type by default[11];
- $\texttt{(}\ [\textbf{in}|\textbf{out}]V_1\texttt{:}T'_1\texttt{,}\ldots\texttt{,}[\textbf{in}|\textbf{out}]V_m\texttt{:}T'_m\ \texttt{)}$ is the list of formal parameters, where each parameter is either an input parameter (by default), **in**, or an output parameter, **out**;
- $\texttt{[}\ X_1[\texttt{:}T''_1]\texttt{,}\ldots\texttt{,}X_p[\texttt{:}T''_p]\ \texttt{]}$ is the list of formal exceptions, empty by default;
- T''_j is the type of values associated with exception X_j, which is `()` by default. That is, by default an exception has no value associated; and
- B is the *body* of the process, the behaviour that describes how the process behaves.

Declared processes may be called by using process instantiations, whose syntax is:

$$\Pi\ \texttt{[}[GPL]\texttt{]}\ \texttt{(}[APL]\texttt{)}\ [\texttt{[}XPL\texttt{]}]$$

where

- GPL is the gate parameter list and it may be either in *positional* form, $G'_1\texttt{,}\ldots\texttt{,}G'_n$, or in *named* form, $G_{i_1}\ \texttt{=>}\ G'_{i_1}\texttt{,}\ldots\texttt{,}G_{i_s}\ \texttt{=>}\ G'_{i_s}$ (G_{i_j} are the formal gates and G'_{i_j} the actual gates). When the named form is used, it is not needed to give all the parameters by using the keyword **...** at the end, and in this case, the absent actual parameters will be added with the same names as the formal ones. For example, a process with header

  process Register `[` *in1*`:data,`*in2*`:data,`*out1*`:data,`*out2*`:data` `]` **is**

  can be instantiated as follows:

  Register `[` *in1* `=>` *inGate1*`,`*out1* `=>` *outGate1*`,`**...** `]` `()`

  and this instantiation is equivalent to this one:

  Register `[` *inGate1*`,`*in2*`,`*outGate1*`,`*out2* `]` `()`

- APL is the list of actual parameters where each parameter can be an expression (input parameter) or a pattern (output parameter), and the list may be either in positional or named form; and
- XPL is the exception parameter list, which may also be either in positional or named form.

An instantiation of a process Π stands for the body of the process where actual gates, parameters, and exceptions are substituted for formal ones.

[11] This is made for compatibility with LOTOS.

3 Base Language for data types

In this chapter we describe how the data types used in the behaviour language described in the previous chapter can be defined in E-LOTOS, and how values of these types are used.

3.1 Data types

As we have said, the part regarding the declaration and use of data types is one of those that has been changed more in E-LOTOS with respect to its predecessor LOTOS.

In LOTOS the abstract data type specification language ACT ONE [EM85] is used to declare new data types and to represent their values (called value expressions in LOTOS). This language is not too user-friendly and suffers from several limitations such as the semi-decidability of equational specifications, the lack of modularity, and the inability to define partial operations.

In E-LOTOS, ACT ONE has been replaced by a new language in which data types are declared in a similar way to functional languages (ML, Haskell), and where some facilities for the representation of values are given.

Predefined types. In E-LOTOS there is a set of predefined types with associated operations which are specified in the "Predefined Library" described in Chapter 7 of [Que98]. Predefined types are:

- bool: with constants *true* and *false*, and operations not, and and or among others.
- nat: the natural numbers. In E-LOTOS, we have a specific syntax (similar to the usual one) to generate values of this type (and also for the rest of the numeric types), so that we can write 1, 7, 2258... instead of having to use the constructors, to write expressions like

  `Succ(Succ( ...  Succ(0)...)).`

  E-LOTOS also provides arithmetic operations, like +, −, ∗, div, mod, and comparison operations, like >, >=, ! =, ...
- int: the integer numbers with a conventional syntax like naturals, and with arithmetic and comparison operations. There is also an operation to transform a positive integer to the corresponding natural (nat), and another to transform a natural to the corresponding integer (int)
- rational: the rational numbers (i.e. integer/integer). In addition to arithmetic and comparison operations, there are functions like round(r), to get the nearest integer value of r, ceil(r), to get the least integer value greater than or equal to r, and floor(r), to get the greatest integer value less than or equal to r.
- float: the real numbers, with arithmetic, comparison, and trigonometric operations.

- char: the character values, written between quote symbols (e.g. 'a'). They represent the ISO Latin-1 characters. There are functions like tolower, toupper, isalpha, isdigit, islower, isupper.
- string: sequences of characters between double quotes (e.g. ''Hello, world!''). There are operations to determine the number of characters of a string (length), to concatenate two strings (concat), to get any prefix or suffix of a string, and to get the character in any position. There are also comparison operations, and operations to convert any natural, integer, or float to a string.
- List: lists of values of any type. It is defined as a new data type (see below)

```
type List is
  nil | cons(any,List)
endtype
```

The user can define lists of a concrete type as we explain below.

Predefined type schemes. In E-LOTOS there is also a set of type schemes that are translated to usual type and function declarations. They are used to make easier the definition of typical types, as suggested by the "rich term syntax" of [Pec94].

We can define an *enumerated* type ET with values $V_1, \ldots, V_n$ as

```
type ET is
  enum V_1, ..., V_n
endtype
```

This type has several predefined functions such as a comparison operation (==), and operations to get the next value (succ) or the previous value (pred) of a given one.

For example, we can define a type colour with some colours:

```
type colour is
  enum Blue, Red, Green, Yellow, Pink
endtype
```

which is translated into a new data type (see below) with a constructor for each enumerated value:

```
type colour is
  Blue | Red | Green | Yellow | Pink
endtype
```

The language also allows record types to be easily defined and dealt with. It is possible to declare a record by giving the list of its fields together with their corresponding types. For example, we can define the record type

```
(name => string, address => string, age => nat)
```

and we can access each field with the "." notation, for example *rec.name*, provided variable *rec* has the above type.

We can also define sets of values of a given type T, with syntax

```
type ST is
  set of T
endtype
```

and express extensionally values of this type, i.e. by giving the list of their elements

$$\{e_1,\ \ldots\ ,e_n\}$$

where e_i are expressions of type T. This scheme has predefined functions to calculate the union (union), difference (diff), and intersection (inters) of two sets; to calculate the number of elements (card) of a set; and to know if an element belongs (isin) to a set; and to know if a set is empty (isempty).

We can define lists of elements of type T in the following way

```
type LT is
  list of T
endtype
```

and write list values with syntax

$$[e_1,\ \ldots\ ,e_n]$$

where e_i are expressions of type T. This scheme has predefined functions to get the first element (head) of a list, to remove the first element of a list (tail), to get the nth element of a list (nth), to concatenate (concat) two lists, and to get the number of elements (length) in a list.

By using these functions we can write a behaviour that adds the elements of a list of integers, xs, and communicates the result through gate *outP*, but without destroying the given list:

```
var total : int := 0, len : int := length(xs),e:int,n:nat in
  for ?n := 1 while n<=len by ?n := n+1 do
    ?e := nth(xs,n);
    ?total := total+e;
  endwhile;
  outP(!total)
envar
```

The predefined operations on these types and their implementations are shown in Chapter 7 of [Que98].

User defined types. The user can define two kinds of types: type synonyms and new data types.

A type synonym declaration simply declares a new identifier for an existing type. For example, we can define the type Complex to represent complex numbers as a record with two float fields:

```
type Complex is
  (real => float, imag => float)
endtype
```

The general syntax to declare a type synonym is

```
type S is (RT) endtype
```

where S is a type identifier, and *RT* is a record type. This is a non-empty list of type identifiers separated by commas, where each type can be labelled with the name of the corresponding record field. In the previous example the record has two fields: *real* and *imag*, both of type float.

It is also possible to declare a type synonym renaming an existing type, with syntax

```
type S renames T endtype
```

where T is a type identifier. For example, the type time can be declared as a synonym of the predefined type nat with the declaration

```
type time renames nat endtype
```

Type equality is *structural*, not by *name*. Thus, with the above definitions, we can use Complex and the anonymous type (*real* => float, *imag* => float) as the same type, and also nat and time.

The declaration of a new data type consists of the enumeration of all the constructors for that type, each one with the types of its arguments, separated with "|". The concrete syntax to declare a new data type is:

```
type S is
  C_1(RT_1) | ... | C_n(RT_n)
endtype
```

For example, we can define a type dest which represents the destination of a message when it reaches a router with two exits, one on the left and one on the right:

```
type dest is
  left | right
endtype
```

Also, we can define the type of messages, data messages or acknowledgment messages (for error handling), as follows:

```
type pdu is
  send(packet,bit) | ack(bit)
endtype
```

We can define recursive data types, by using in some constructors the type which is being declared, as in the predefined type List described above.

The base language does not allow to declare parameterized types. This is left for the module system (see Section 4.3).

3.2 Type expressions

We have seen in the previous section how the user can *declare* new types with the facilities of the language. Now we are going to see how to write type expressions, that is, what the user can write when a type is needed. For example, when the user may want to type a gate or a local variable in a behaviour.

The simplest type expression is a type *identifier*. We have already seen several examples: nat and bool are predefined type identifiers, Complex is a type synonym identifier, and dest is a new datatype identifier.

Type identifiers are the unique type expressions the user can use in order to type an E-LOTOS element. For example, when the user types a gate of a process, $G : T$, T must be a predefined or user-declared type identifier, or one of the special type identifiers, **none** and **any**, that we present below. It is the same when we want to declare a variable or a pattern type. Thus, at the user level, in E-LOTOS there are no anonymous types. For instance, the user is *not* able to declare

$$G : (\textit{real} \Rightarrow \mathsf{float}, \textit{imag} \Rightarrow \mathsf{float}),$$

although, of course, it is possible to declare

$$G\ \texttt{:}\ \mathsf{Complex}$$

provided the Complex type is already declared.

We have two special types:

- **none** is the empty type that has no values, used to give functionality to processes that never stop.
- **any** represents a wildcard type which is used to type gates which can communicate data of any type, in order to achieve compatibility with LOTOS.

We can also write record type expressions *RT*, although as we have seen the user can only use them in the declaration of type synonyms. Note that a record type expression in *not* a type, and it becomes a (anonymous) type when we enclose it between parentheses, (*RT*). We have also seen that each component of a record can be associated with a field name. But when we give name to a field, we have to give name to all the fields. For example, in the record type expression

$$(x\ \texttt{=>}\ \mathsf{float}, y\ \texttt{=>}\ \mathsf{float})$$

x and y are the names of the record fields. In this way, the order of the different fields is not important, but, of course, the names of the fields are. However, it is optional to give name to the different fields of a record, so a correct record type expression is

$$(\mathsf{int},\mathsf{float}),$$

which represents a record with an integer and a float. In this case the order of the fields *is important.* Really, this expression is syntactic sugar for

$$(\$1\ \texttt{=>}\ \mathsf{int}, \$2\ \texttt{=>}\ \mathsf{float})$$

where E-LOTOS *invents* default names for the fields, $\$1, \$2, \ldots$, when the user does not give them. And ($1 => float,$2 => int) is a different type.

In E-LOTOS we can write *extensible* record types, with the keyword **etc** at the end of the record, to mean any set of fields. So the record type

$$(\mathit{message}\ \texttt{=>}\ \mathsf{data},\ \mathbf{etc})$$

represents a type of records with *at least* one field called *message* of type data. This means that we have as values of this type all the record values that have *at least* a field with that name and type. The notion of extensible records is related to record subtyping that we detail in the next section.

3.3 Subtyping

E-LOTOS has a built-in subtyping relation between record types. A record type is subtype of another record type if the former has at least all the fields the latter has. So the record type (**etc**) is a supertype of any other record type. And the extensible record type

$$(\mathit{message}\ \texttt{=>}\ \mathsf{data},\ \mathbf{etc})$$

has as subtypes, for example, the record types

$$(\mathit{message}\ \texttt{=>}\ \mathsf{data},\ \mathit{de}\ \texttt{=>}\ \mathsf{dest},\ \mathbf{etc}) \quad \text{and} \quad (\mathit{message}\ \texttt{=>}\ \mathsf{data},\ \mathit{de}\ \texttt{=>}\ \mathsf{dest})$$

We denote that type T is a subtype of type T' by $T \sqsubseteq T'$. Whenever $T \sqsubseteq T'$ we can use values of type T where values of type T' are required.

Let us see an example that shows the usefulness of subtyping. Let us suppose we have to specify a Router process that receives packets through an input gate, and it has to direct them to one of its output gates depending on the packet destination. So the only restriction is that packets that get into the router have to have a field indicating their destination. We can declare an extensible record type

```
type packet is
  (de => dest, etc)
endtype
```

and then a process specifying the router is the following:

```
process Router [inP:packet, leftP:packet, rightP:packet] is
  var p:packet in
    loop
      inP(?p);
      case p.de is
          left -> leftP(!p)
        | right -> rightP(!p)
      endcase
    endloop
  endvarendproc
```

The empty type **none** is subtype of any type T, **none** $\sqsubseteq T$, and it is equivalent to any record with a field of type **none** (since that field cannot have any value, the whole type has no values). The universal type **any** is supertype of any type T, that is, $T \sqsubseteq$ **any**.

3.4 Expressions

In contrast to LOTOS [ISO89], where there is a separation between processes and functions, E-LOTOS considers functions as a kind of process. A function in E-LOTOS is any process with the following characteristics: it is deterministic; it cannot communicate (i.e. it has no gates), and so its only capabilities are to return values and raise exceptions; and it has no behaviour over time (i.e. a function is an immediately exiting process). Therefore, the expression (sub)language is very similar to that of behaviours that we have been introducing, once the elements related to these characteristics are removed.

The simplest expressions are *normal forms.* A normal form is an expression which cannot be reduced any further. The following examples are normal forms:

- a *primitive* constant, such as 5, *true*, or `"Hello, world!"`
- a variable, such as x or *total*
- a constructor of a new data type applied to normal forms, such as *right*, *nil*, *cons*(5, *nil*), and *send*(*aPacket*, *aBit*)
- a record of normal forms, such as (*real* => 4.5, *imag* => 8) or (7, 9) (that is syntactic sugar for ($1 => 7, $2 => 9))

Starting from these simple expressions we can construct more complex ones using the operators of E-LOTOS.

Expressions can raise exceptions, provided they are in the scope of an application of the **trap** operator where they are handled, as we saw in Section 2.15:

```
trap
  exception $X_1$[($P_1$)[:$T_1$]] is $E_1$ endexn
    ...
  exception $X_n$[($P_n$)[:$T_n$]] is $E_n$ endexn
  [exit [$P_{n+1}$] is $E_{n+1}$ endexit]
in $E$
endtrap
```

However, note that the handler behaviours have been substituted by expressions here.

For example, let us consider the function Head that returns the head of a list:

```
function Head (xs:intlist) : int raises [Hd] is
  var x:int := 0 in
    case xs is
         nil -> signal Hd;0
       | cons(?x,any:intlist) -> x
    endcase
  endvar
endfunc
```

Function Head declares that it may raise an exception, whose *formal* name is *Hd*. Inside the function, pattern-matching is used to distinguish whether the integer list *xs* is empty or not (patterns and pattern matching will be detailed in Section 3.6). When the list is empty the function raises the exception *Hd* (and if the exception *Hd* is not trapped, the function Head returns 0, because all the branches in a **case** statement have to return something of the same type if they finish). In this way, the user of the function has to give the handler of the exception. For example, suppose we do not want an error to be produced when the head of an empty list is consulted. Instead a 0 is to be returned:

```
trap
    exception Hd0 is 0 endexn
in
    Head (xs) [Hd0]
endtrap
```

that returns 0 if the list *xs* is empty, and otherwise returns the head of the list. The exception *Hd0* is the *actual* exception in the Head function call.

We can also rename exceptions raised by an expression (in this case there is no sense in renaming gates as expressions do not have gates) with all the renaming power we saw in Section 2.17. Now the syntax is

```
rename
  signal $X_1$[($P_1''$)[:$T_1'$]] is $X_1'$[($E_1$)]
    ...
  signal $X_n$[($P_n''$)[:$T_n'$]] is $X_n'$[($E_n$)]
in $E$
endren
```

The expression language also has a conditional expression **if** – **then** – **else**; and some imperative features, such as:

- assignment, P `:=` E, that produces bindings between the variables in the pattern P and values of E. That expression does not return a value but produces bindings (between variables in P and the value returned by expression E).
- sequential composition, E_1 ; E_2, which returns bindings produced by E_1 not overridden by those of E_2, bindings produced by E_2 and the value returned by E_2.
- declaration of local variables, with syntax

  var V_1:T_1[:=E_1], . . . ,V_n:T_n[:=E_n] **in**
    E
  endvar

  For example, the expression

  var x:int **in**
    ?x:=E; $x * x$
  endvar

  returns the same value as the expression $E * E$ does, and it does not produce bindings because variable x is local to the **var** declaration, and its binding does not go out of this declaration.
- different iterative constructions such as **loop**, **while**, and **for**.

Finally, E-LOTOS includes some operators that can only be applied to expressions:

- boolean conjunction E_1 **andalso** E_2 that evaluates E_1; if its value is *true* then the value of E_2 is returned, otherwise *false* is returned. E_1 and E_2 must be of type bool, that is, they have to return a value of this type.
- boolean disjunction E_1 **orelse** E_2 that evaluates E_1; if its value is *false* then the value of E_2 is returned, otherwise *true* is returned. E_1 and E_2 must be of type bool.
- equality operation, E_1 = E_2. E_1 and E_2 do not need to have the same type.
- inequality operation, E_1 <> E_2. E_1 and E_2 do not need to have the same type.
- select field operation, E.V. Assuming that expression E returns a record value with a field called V, E.V returns the value associated with this field.
- explicit typing, E:T, that returns the value of expression E only if it is of type T. This is controlled by the static semantics at compilation time.

3.5 Function declaration and instantiation

Much as we can declare processes (Section 2.20), we can declare functions in order to give a name to an expression E, and abstract the names of variables (or subexpressions) and exceptions in E.

The syntax of a function declaration is as follows:

function F [([**in**|**out**]V_1:T_1, . . . ,[**in**|**out**]V_m:T_m)] [:T]
  [**raises** [X_1[:T'_1], . . . ,X_p[:T'_p]]]
is E
endfunc

where

- F is a process identifier;

- ([**in**|**out**]V_1:T_1, ... ,[**in**|**out**]V_m:T_m) is the list of formal parameters, empty () by default;
- T is the type of the value returned by F, and it is an empty record () by default;
- [X_1[:T'_1], ... ,X_p[:T'_p]] is the list of formal exceptions, empty [ ] by default; and
- expression E is the body of the function F.

A function call is an expression. Its syntax is

$$F[\texttt{(}APL\texttt{)}][\texttt{[}XPL\texttt{]}]$$

where APL is the list of actual parameters and XPL is the exception parameter list, which may be in either positional or named form.

We can declare also *infix* functions

function F **infix** [([**in**|**out**]V_1:T_1,[**in**|**out**]V_2:T_2)] [:T]
[**raises** [X_1[:T'_1], ... ,X_p[:T'_p]]]
is E
endfunc

which can be called as follows:

$$AP_1\ F\ AP_2[\texttt{[}XPL\texttt{]}]$$

We can also declare *values*, which represent constant functions without parameters, with syntax

value V : T **is** E **endval**

where V is the value identifier, T is a type identifier, and the value of expression E (which has to be of type T) is what value V returns.

3.6 Patterns and pattern matching

We have already used patterns in several examples: assignments P:=E, actions G P_1 @ P_2, etc. Patterns are matched against values and can produce bindings on variables. Now we are going to describe the different kinds of pattern in E-LOTOS. A pattern has one of the following forms:

- a variable, ?x. If we try to match ?x against a value N of type T, the pattern matching succeeds if the variable x has been declared with type T' and T is a subtype of T'. In this case, x is bound to N.
- an expression, !E. This pattern can be matched against the value N only if N is the value of the expression E. In this case, the pattern matching does not produce bindings.
- a wildcard pattern, **any**:T, that matches against any value of type T without producing bindings.
- a record pattern, (RP), where RP is a list of patterns separated by commas, either in a *named* form V_1 => P_1, ... ,V_n => P_n (where $V_1, \ldots, V_n$ are the field names of the record) or in a *positional* form P_1, ... ,P_n. The pattern matching of (RP) against the value N will succeed if N has the form (RN) and RN is a list of values (possibly preceded by field names) that match against the corresponding patterns in RP. This pattern matching produces the bindings produced by matching each pattern of RP against the corresponding value in RN. Patterns in RP must bind disjoint variables.

For example, the record pattern (*real*=>?re,*imag*=>?im) matches the value (*real*=>2.5,*imag*=>0.3) binding variable re to the value 2.5 and variable im to the value 0.3.

In a record pattern RP we can use the keyword **etc** at the end of RP for representing all the unspecified fields. So, the pattern (*real* => ?re,**etc**) may be matched against the value (*real* => 2.5,*imag* => 0.3) producing a binding between the variable re and the value 2.5.

- a constructor application, C[(RP)], where RP stands for the arguments of the constructor C (if needed). The pattern C(RP) matches the value N if N has the form C(RN) and RN matches RP. The bindings are those produced by matching each pattern against the corresponding value. For example, *cons*(?x,?l) or *cons*(!3,**any**:List).
- an explicit typing, P:T. The pattern P:T matches the value N if N has type T and P matches against N. For example, the pattern ?l:intlist matches the value *cons*(4,*nil*).

Patterns are also used in the **case** operator, whose syntax is

```
case E[:T] is
    P_1[[E_1]] -> B_1
      ⋮
  | P_n[[E_n]] -> B_n
endcase
```

where expressions E_i are boolean expressions, which are *true* by default.

The value of the expression E is matched sequentially against each of the clauses P_1 [E_1], . . . , P_n [E_n]. A value N matches a clause P_i [E_i] if the value matches the pattern P_i and E_i evaluates to *true* in the context of variables bound by the pattern-matching. The behaviour B_i associated with the first clause that matches N describes how the whole behaviour continues. If there is no clause that matches N then the predefined exception *Match* is raised.

For example, we can define a process that receives messages of type pdu and behaves in a different manner depending on the kind of message:

```
process P [ inP:pdu, . . . ] is
  var p:packet in
    inP(?p);
    case p is
        send(?pa:packet,?b:bit) -> (* handle data *)
      | ack(?b:bit) -> (* handle acknowledgment *)
    endcase
  endvar
endproc
```

There is also the possibility of pattern-matching between two patterns, for example, in the communication

$$inP(?x)\ ||\ inP(!3)$$

In this case the pattern ?x is pattern-matched against the pattern !3. The pattern-matching between two patterns succeeds if there is a value N such that both patterns can be matched against N. In the example this value is 3.

Although strange, we can put together two behaviours that can receives values on a gate *inP*,

$$inP(?x)\ ||\ inP(?y)$$

In this case, a value of the type of values that G can communicate is “*invented*” and both x and y are bound to this value.

4 Module Language

When large specifications are developed, it is useful to encapsulate several related data types, functions, and processes so that they can be regarded as a single unit (a *module*) with which one can work. One can also want to combine different modules, to control what objects a module shows, and to build generic modules that are parameterized by other modules.

LOTOS has a limited form of modularity, whose modules only encapsulate types and operations but not processes, and do not support abstraction (every object declared in a module is exported outside),[1] E-LOTOS has a new modularization system, which allows

- to define a set of related objects (types, functions, and processes),
- to control what objects the module exports (by means of interfaces),
- to include within a module the objects declared in other modules (by means of import clauses),
- to hide the implementation of some objects (by means of opaque types, functions, and processes), and
- to build generic modules.

In order to facilitate this modularization, a separation between the concept of module *interface* and *module* definition is made. An interface declares the visible objects of a module and what the user needs to know about them (the name of a data type or a function header, for example). A module gives the definition (or implementation) of objects (visible or not).

A specification in modular E-LOTOS is a sequence of interface (Section 4.1) and module (Section 4.2) or generic module (Section 4.3) declarations, besides a specification declaration (Section 4.4).

4.1 Interfaces

As we have said above, an interface defines the visible part of the objects (types, functions, and processes) declared within a module.

The syntax of an interface declaration is as follows:

interface *int-id* [**import** $\mathit{int\text{-}exp}_1$ **,** ... **,** $\mathit{int\text{-}exp}_n$] **is**
  i-body
endint

where *int-id* is an interface identifier, $\mathit{int\text{-}exp}_1, \ldots, \mathit{int\text{-}exp}_n$ are interface expressions that we will describe below, and *i-body* is the body of the interface.

In the body of an interface, the visible parts of types, functions and processes are given. The visible part of a type is its name and, maybe, its implementation. If the implementation of a type

[1] A critical evaluation of LOTOS data types from the user point of view can be found in [Mun91].

is not given, the type is called *opaque*, and only the functions declared in the interface can modify it. An opaque type declaration is as follows:

type T

If the type is not opaque, it is declared as we saw in Section 3.1.

For functions and processes, only their headers are visible. Thus, an interface has function headers like

function F [([**in**|**out**]V_1:T'_1, ... ,[**in**|**out**]V_m:T'_m)] [:T] [**raises** [X_1[:T''_1], ... ,X_p[:T''_p]]]

or processes headers like

process P [[G_1[:T_1], ... ,G_n[:T_n]]] [([**in**|**out**]V_1:T'_1, ... ,[**in**|**out**]V_m:T'_m)]
[**raises** [X_1[:T''_1], ... ,X_p[:T''_p]]]

We can also declare *values*, which represent constant functions without parameters, with syntax

value V : T

where V is the value identifier, and T is a type identifier.

The types of formal parameters must be predefined, imported or declared by the interface.

For example, we can define an interface of a module that implements our register:

```
interface Register_Interface is
  type data
  process Register [in1:data,in2:data,out1:data,out2:data]
endint
```

As we have seen, an interface may import other interfaces, and the imported interfaces are specified by means of interface expressions. An interface expression may be

- an interface identifier *int-id′*, that represents all the identifiers declared in the interface *int-id′*; or
- a renaming of an interface

  [*int-id′* **renaming** (*reninst*)]

  representing the identifiers declared in the interface *int-id′* renamed by the renaming *reninst*, explained below; or
- an explicit interface body (possibly renamed)

  [*i-body* [**renaming** (*reninst*)]]

  that represents the identifiers declared in the interface body *i-body*.

In the last two cases *reninst* is a list of renaming of

- types, **types** S'_1:=S_1, ... ,S'_n:=S_n,
- constructors and functions, **opns** F'_1:=F_1, ... ,F'_m:=F_m,
- processes, **opns** P'_1:=P_1, ... ,P'_q:=P_q, or
- values, **values** V'_1:=V_1, ... ,V'_s:=V_s.

The primed identifiers are the old names (of *int-id′* or of *i-body*) being renamed, and the unprimed identifiers are the new names (which can be used in *int-id*). For example,

```
[Register_Interface renaming (proc Register := Register_Nat)]
```

represents the declarations in the interface Register_Interface where the process Register now is named Register_Nat.

The imported identifiers have to be all different, so we have to use renaming when there are conflicts.

We can define an interface for our system composed by the register, the producer, and the consumer, as follows

```
interface System_Interface import Register_Interface is
  process Producer [p1:data,p2:data]
  process Consumer [c1:data,c2:data]
endint
```

All visible objects of an interface (including the imported ones) are visible from outside and may be imported in other interfaces. Importing is transitive through interfaces.

4.2 Modules

A module specifies the implementation of a set of (related) types, functions, and processes. The objects that a module exports, i.e. that may be imported by other modules, are controlled by means of interfaces.

In E-LOTOS a module declaration is as follows:

module *mod-id* [: *int-exp*] [**import** $\textit{mod-exp}_1, \ldots, \textit{mod-exp}_n$] **is**
  m-body
endmod

where *mod-id* is a module identifier; *int-exp* is an interface expression (Section 4.1) that declares the visible objects of *mod-id*, so that other modules that import *mod-id* only can use the objects declared in *int-exp*[2]; $\textit{mod-exp}_1, \ldots, \textit{mod-exp}_n$ are module expressions that we will describe below; and *m-body* is the body of the module, which is a sequence of type declarations (Section 3.1), function declarations (Section 3.5), and process declarations (Section 2.20).

The objects imported by a module have to have all different identifiers, so renaming is needed when an identifier is used more than once.

For example, we can specify the module that implements a version of the register (a register that communicates natural numbers) as follows:

[2] By default, the interface of a module is the set of objects imported from other modules and the objects declared by the module body.

```
module Register_Mod: [Register_Interface renaming (proc Register := Register_Nat)] is
  type data renames nat endtype
  process Register_Nat [in1:data,in2:data,out1:data,out2:data] is
    var x1:data,x2:data in
    in1(?x1 : data);
      ( in2(?x2 : data)
      |||
      out1(!x1) );
      out2(!x2)
    endvar
  endproc
endmod
```

A module expression may be a module identifier possibly restricted by an interface expression and renamed:

mod-id [: *int-exp*] [**renaming** (*reninst*)]

or an instantiation of a generic module as we will see in Section 4.3.

The module that implements the objects declared in System_Interface may be

```
module System_Mod import Register_Mod is
  value value1:data is 8 endval
  value value2:data is 15 endval
  process Producer [p1:data,p2:data] is
      p1(!value1); p2(!value2)
  endproc
  process Consumer [c1:data,c2:data] is
    var v1 : data,v2 : data in
      c1(?v1 : data); c2(?v2 : data)
    endvar
  endproc
endmod
```

4.3 Generic Modules

Genericity is a useful mechanism to construct re-usable specifications. In E-LOTOS, a generic module declaration is as follows:

```
generic gen-id ( mod-id$_1$:int-exp$_1$, ... ,mod-id$_n$:int-exp$_n$ ) [: int-exp]
 [import mod-exp$_1$, ... ,mod-exp$_m$] is
   m-body
endgen
```

gen-id is a generic module identifier. The *mod-id*$_i$ are module identifiers, which are called the formal parameters of the generic module, and whose visible objects are declared in the interface expressions *int-exp*$_i$. *mod-exp*$_j$ are module expressions as defined in Section 4.2. *m-body* is the body of the generic module, where the objects declared in the *int-exp*$_i$, and the imported ones, may be used.

For example, instead of implementing a different register depending on the values that the register can communicate, we can define a generic module that implements a *generic* register parameterized by the type of the values the register can communicate:

```
interface Data is
  type data
endint
generic Register_Gen (D:Data) is
  process Register [in1:data,in2:data,out1:data,out2:data] is
    var x1 : data,x2 : data in
      in1(?x1 : data);
      ( in2(?x2 : data)
      |||
      out1(!x1) );
      out2(!x2)
    endvar
  endproc
endgen
```

In order to use a generic module we have to instantiate it, by providing actual parameters, which must be modules that match the corresponding interface expression. A module matches an interface whether it implements at least the objects declared in the interface.

The syntax of a generic module instantiation is as follows:

```
gen-id ( mod-id$_1$ => mod-exp$_1$ , ... ,mod-id$_n$ => mod-exp$_n$ ) [: int-exp] [renaming ( reninst )]
```

where *mod-id*$_i$ are module identifiers that must be the names of the formal parameters of *gen-id*, *mod-exp*$_i$ are module expressions that must match the corresponding interface expressions in the declaration of *gen-id*. *int-exp* is an interface expression and *reninst* is a renaming clause as we saw in Section 4.1.

A generic module instantiation is a module expression, and may be the body of a module, or may appear in an importation clause.

We can instantiate our generic register in order to build a register of natural numbers:

```
module Mod_Register_Nat is
  Register_Gen (D => NaturalNumbers renaming (types nat := data))
    renaming (proc Register := Register_Nat)
endmod
```

4.4 Specification

The entry point of an E-LOTOS description is the *specification* declaration, with syntax

```
[top-dec]
specification Σ [import mod-exp$_1$ , ... ,mod-exp$_n$] is
  [gates $G_1$:$T_1$ , ... ,$G_m$:$T_m$]
  [exceptions $X_1$:$T'_1$ , ... ,$X_p$:$T'_p$]
  (behaviour B | value E )
endspec
```

where

- *top-dec* is a sequence of interfaces, modules, and generic modules declarations;
- Σ is the name of the specification;
- *mod-exp*$_i$ are module expressions that define the imported modules;

- $G_1 : T_1$, ..., $G_m : T_m$ is the list of gates of the whole system;
- $X_1 : T'_1$, ..., $X_p : T'_p$ is the list of exceptions that the system can raise to its environment; and
- the body of the specification may be a behaviour B or a value E.

For example we can define the specification for the system composed of a producer, a register, and a consumer:

```
specification System import System_Mod is
  behaviour
    hide pr1:data, pr2:data, rc1:data, rc2:data in
      conc
        Producer [pr1, pr2] ()
      | [pr1, pr2] |
        Register_Nat [pr1, pr2, rc1, rc2] ()
      | [rc1, rc2] |
        Consumer [rc1, rc2] ()
      endconc
    endhide
endspec
```

5 Examples

In this chapter we will discuss some well-known examples where the main E-LOTOS features are illustrated.

5.1 Global clock

In this section we are going to specify a *global clock*, that is, a clock that measures time since it is started until it is stopped, and to which other processes running in parallel can ask what time it is. The clock is always able to communicate what time it is, and to be stopped. The process Clock is as follows:

```
process Clock[stopClock,whatTime:time](gtime:time) is
  var gt:time:=0,t:time:=0 in
      whatTime(?gt) @?t [gt=gtime+t];Clock[...](gtime+t)
    []
      stopClock
  endvar
endproc
```

We use the parameter *gtime* to measure the global time. Initially, it is set to 0 (in the initial call), and then its value always represents the time when the last question "What time is it?" was answered. When another question is asked, the clock performs the action

```
whatTime(?gt) @?t [gt=gtime+t]
```

where *gt* represents the value communicated, and although it is like an input (?*gt*), really it is the unique value such that $gt = gtime + t$ where t is the time measured since the action was enabled, that is, since the last question was answered. Therefore, *gt* represents the global time (new value for *gtime*).

At any moment, the clock can be stopped with action *stopClock*. This is necessary if the clock is running in parallel with a process that finishes and we want the whole system (the process and the clock) to finish.

5.2 FIFO queue

Now, we are going to specify a generic FIFO (*first-in, first-out*) queue, which we will use in Section 5.4.

First, we specify the characteristics that the elements of the queue must fulfill in an interface:

```
interface Data is
```

```
  type elem
endint
```

Now, we specify the generic module that describes the queue operations.

```
generic GenQueue(D:Data) is
  type queue is
      Empty
    | Add(queue,elem)
  endtype
  function addQueue(q:queue,e:elem):queue is
    Add(q,e)
  endfunc
  function front(q:queue):elem raises [EmptyQueue] is
    var e:elem is
      e:=any elem;
      case q in
          Empty -> signal EmptyQueue; e
        | Add(Empty,?e) -> e
        | Add(Add(?q,?e),any:elem) -> front(Add(q,e))
      endcase
    endvar
  endfunc
  function delete(q:queue):queue raises [EmptyQueue] is
    var e1:elem,e2:elem is
      case q in
          Empty -> signal EmptyQueue; Empty
        | Add(Empty,any:elem) -> Empty
        | Add(Add(?q,?e1),?e2) -> Add(delete(Add(q,e1)),e2)
      endcase
    endvar
  endfunc
  function isEmpty(q:queue):bool is
    case q in
        Empty -> true
      | Add(any:queue,any:elem) -> false
    endcase
  endfunc
endgen
```

And now we can instantiate this generic queue to make, for example, a queue of natural numbers:

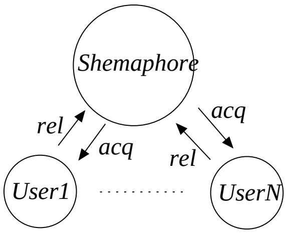
```
module NatQueue is
  GenQueue(D => NaturalNumbers renaming(types nat := elem))
endmod
```

5.3 Random semaphore

We study now the classical example of the *critical section problem*, which is one of the classic concurrent programming problems. Let us suppose that a set of n processes (*users*) have to access a shared resource in mutual exclusion, i. e. the resource can be used by at most one user at a time. So the behaviour of each user should be

1. non-critical section
2. entry protocol
3. *critical section*
4. exit protocol
5. go to 1.

In order to ensure mutual exclusion, we introduce a *semaphore* process which all the users have to synchronize with, before and after entering the critical section. In this first attempt, the semaphore allows to entry to the critical section of *any* of the processes that are waiting, provided none is already in. So the semaphore allows one of those users to enter the critical section and then that user notifies its exit from the critical section. The structure of the whole system could be



We can model a user as a process that continuously behaves as described above. All the users notify its release through the same gate *rel* (release), and the semaphore uses the same gate to allow access to every user, *acq* (acquire). In order to distinguish users each one will have a different identifier (implemented as a parameter). A user has to provide this identifier each time to acquire or release the resource. An E-LOTOS specification of a user may be:

```
process User1 [acq:id,rel:id](myid:id) is
  loop
    (* non-critical section *)
    acq(!myid);
    (* use shared resource *)
    rel(!myid)
  endloop
endproc
```

where type id may be a synonym of the predefined nat type:

```
type id renames
  nat
endtype
```

We can also specify the infinite behaviour of the user with a recursive process:

```
process User2 [acq:id,rel:id](myid:id) is
  (* non-critical section *)
  acq(!myid);
  (* use shared resource *)
  rel(!myid);
  User2 [acq,rel](myid)
endproc
```

As another example, we can modify this process by making explicit the state of the user, i. e. whether it is in or out the critical section. We could write:

```
process User3 [acq:id,rel:id](myid:id) is
  var st:state := outside in
    loop
      case st is
          outside -> (* non-critical section *)
                     acq(!myid);
                     ?st := inside
        | inside -> (* use shared resource *)
                     rel(!myid);
                     ?st := outside
      endcase
    endloop
  endvar
endproc
```

where we have used a new type, state, that has two different values, *inside* and *outside*:

```
type state is
   inside | outside
endtype
```

The Semaphore process uses the same gates as users do. When the resource is free, the semaphore waits for someone who wants to enter, and keeps in variable *usr* who has entered. Then, the resource is not free and the semaphore allows the current user (*usr*) to release the resource. When this user does so, the semaphore repeats its behaviour. The specification of this process is as follows:

```
process Semaphore[P:id, V:id] is
  var usr:nat in
    P(?usr);
    V(!usr);
    Semaphore[P, V]()
  endvar
endproc
```

We can model the entire system by instantiating several times the User1 process (each one with a different identifier) and composing them in parallel with an instantiation of the Semaphore process. We hide the gates that stand for synchronization between the users and the semaphore, in order to make them urgent and because they are of no interest from the point of view of the environment. So the behaviour of the system can be specified with:

```
hide acq:id,rel:id in
  par ?usr in [1,2,3,4,5] |||
    User1[acq,rel](usr)
  endpar
||
  Semaphore[acq,rel]
endhide
```

And the complete E-LOTOS specification would be:

```
module CriticalSection1 is
  type id renames
```

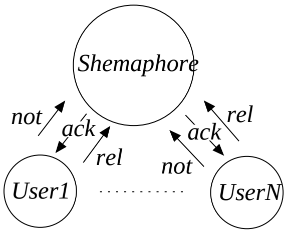
```
      nat
    endtype
    process User1 [acq:id,rel:id](myid:id) is
      loop
        (* non-critical section *)
        acq(!myid);
        (* use shared resource *)
        rel(!myid)
      endloop
    endproc
    process Semaphore[P:id,V:id] is
      var usr:nat in
        P(?usr);
        V(!usr);
        Semaphore[P,V]()
      endvar
    endproc
  endmod

  specification RandomSemaphore import CriticalSection1 is
  behaviour
    hide acq:id,rel:id in
      par ?usr in [1,2,3,4,5] |||
        User1[acq,rel](usr)
      endpar
    ||
      Semaphore[acq,rel]
    endhide
  endspec
```

In this example, when there are several users waiting to access the shared resource, it is not known which of them will succeed. From this point of view, the semaphore is *random.* Any of the waiting users can be the next to access the resource. In the next section the semaphore will take care of the order in which the waiting users access the resource.

5.4 FIFO semaphore

In this section we discuss the same problem as in the last section, but in this case the waiting users will be served in the order they arrived at the critical section. In order to achieve this, each user has to notify its intention to access the critical section. So, in this case there is a new communication between the users and the semaphore:



When a user notifies to the semaphore its wish to access the critical section, the semaphore keeps the user identifier in a queue of identifiers. When the resource is free, the semaphore allows access to the user whose identifier is at the front of the queue. So, we need a queue of identifiers. We can use the generic queue of Section 5.2, by doing the following instantiation:

```
module NatQueue is
  GenQueue(D => DataTypes renaming(types id := elem))
endmod
```

where DataTypes is the module that specifies the new data types required in this example, and id should be among them.

We have to modify also the User2 process in order to notify to the semaphore its intention to access the resource:

```
process User4 [not:id,acq:id,rel:id](myid:id) is
  (* non-critical section *)
  not(!myid);acq(!myid);
  (* use shared resource *)
  rel(!myid);
  User4 [...](myid)
endproc
```

The semaphore has to allow always a user notification, and it has to insert the user identifier in the queue; if the resource is free and there is some user waiting, then it has to allow access to the user in the front of the queue; and if the resource is not free, it has to wait for the current user to go out. The semaphore specification in this case is:

```
process Semaphore[not:id,acq:id,rel:id](free:bool,q:queue,usrIn:nat) is
  var usr:nat:=0 in
    trap
      exception EmpQ is stop endexn
    in
      sel
          not(?usr);Semaphore[...](free,addQueue(q,usr),usrIn)
        []
          acq(!front(q)[EmpQ]) [free and not(isEmpty(q))];
          Semaphore[...](false,q,front(q)[EmpQ])
        []
          rel(!usrIn)[not(free)];Semaphore[...](true,delete(q)[EmpQ],0)
      endsel
    endtrap
  endvar
endproc
```

In this case, we have used a different kind of specification than in, for example, User3. We have used a recursive process and, instead of using several **if** − **then** − **else** instructions, we use a selection where each branch begins with an action with a selection predicate that specifies when it is possible to execute this branch. For example, the first branch begins with the action *not*(?*usr*) (with default selection predicate [*true*]), that is, it is always possible to take a notification from a user. The second branch begins with

acq(!front(*q*)[*EmpQ*]) [*free* and not(isEmpty(*q*))]

that is, the user who is the first of the queue can access the resource provided that the resource is free and there are users waiting. In fact, the condition not(isEmpty(*q*)) is not necessary, without care of an exception being raised. If q is empty, the pattern `!front(q) [EmpQ]` fails and it cannot match against any value, so the action is not offered.

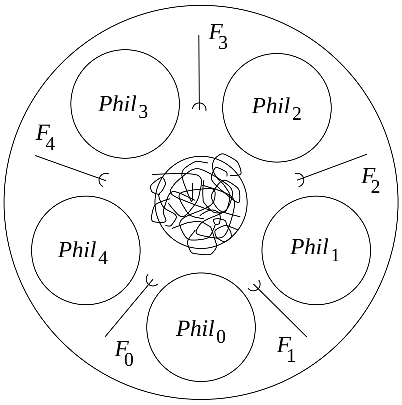
Finally, in the specification of the entire system, the only needed modification is the inclusion of the new gate *not*:

```
specification RandomSemaphore import CriticalSection2 is
behaviour
  hide not:id, acq:id, rel:id in
    par ?usr in [1,2,3,4,5] |||
      User4 [...] (usr)
    endpar
  ||
    Semaphore [...] (true, Empty, 0)
  endhide
endspec
```

where CriticalSection2 is the module that includes the declaration of types and processes used.

5.5 Dining philosophers

This problem, originally stated and solved by E.W. Dijkstra [Dij65], is set in a monastery where five monks are dedicated philosophers. Each philosopher has a room in which he can engage in thinking. There is also a common dining room, with a circular table with five plates, each labeled by the name of the philosopher who uses it, as the following figure shows:



To the left of each philosopher there is laid a fork, and in the center stands a large bowl of spaghetti, which is constantly replenished. A philosopher is expected to spend most of his time thinking, but when he feels hungry, he goes to the dining room, takes a seat, eats, and then returns to his room to think. However, the spaghetti is so entangled that two forks are needed simultaneously in order to eat.

The problem is to devise a *ritual* (protocol) that will allow the philosophers to eat. Each philosopher may use only the two forks adjacent to his plate. The protocol must satisfy the following requirements:

- mutual exclusion, that is, two philosophers cannot use the same fork simultaneously;
- freedom from deadlock and lockout, that is, absence of starvation – literally!

Our first solution[1] consists in the philosophers having to ask for the forks in order to take them. In this way, each fork controls ensures only one philosopher (one that is adjacent) takes it at any time. The process that specifies the behaviour of a philosopher may be:

```
process Philosopher[sits_down,picks_up,puts_down,gets_up](id:nat) is
  (* think *)
  sits_down(!id);
  picks_up(!(id,id));  picks_up(!(id,right(id)));
  (* eat *)
  puts_down(!id);
  puts_down(!(id,right(id)));
  gets_up(!id);
  Philosopher[...](id)
endproc
```

The process is parameterized by the identifier of the philosopher that represents. When the philosopher sits down, he tries to pick up the fork on this left, *picks_up*(!(*id*,*id*)) (where the second value represents the number of the fork which is picked up), and when he has picked up it, he tries to pick up the fork in his right (right(*id*) represents the identifier of the fork which is on the right of the philosopher *id*). When the philosopher has two forks he eats, and then he puts down the forks and gets up.

The process that specifies the behaviour of the forks would be:

```
process Fork[picks_up,puts_down](id:nat) is
    picks_up(!(id,id));puts_down(!(id,id));Fork[...](id)
   []
    picks_up(!(left(id),id));puts_down(!(left(id),id));Fork[...](id)
endproc
```

That is, a fork allows to be picked up by the philosopher with the same identifier or by the philosopher who is on its left (left(*id*)).

And the behaviour of the dining room with five philosophers and five forks may be specified as follows:

```
hide picks_up,puts_down in
    par ?phi in[1,2,3,4,5] |||
      Philosopher[...](phi)
    endpar
  |[ picks_up,puts_down  ]|
    par ?fo in[1,2,3,4,5]  |||
      Fork[...](fo)
    endpar
endhide
```

But this solution may become deadlocked. For example, if all the philosophers get hungry at the same time, they all sit down, pick up their left fork, and try to pick up the other fork, which is not free.

The problem may be solved by adding a *butler* (which acts as the semaphore in the previous example) whose permission the philosophers have to ask for sitting down, and to whom they must communicate that they get up. The butler never allows more than four philosophers to be seated simultaneously. His behaviour may be specified with the following E-LOTOS process:

[1] As we will see below, this is not really a good solution.

```
process Butler [ sits_down, gets_up ] (seated:nat) is
   sits_down(?id) [ seated<4 ] ; Footman [ ... ] (seated + 1)
  []
   gets_up(?id) ; Footman [ ... ] (seated − 1)
endproc
```

Now, we have to add the footman to the whole system:

```
hide sits_down, picks_up, puts_down, gets_up in
   ( par ?phi in [1,2,3,4,5] |||
       Philosopher [ ... ] (phi)
     endpar
  |[ picks_up, puts_down ]|
     par ?fo in [1,2,3,4,5] |||
     Fork [ ... ] (fo)
     endpar )
  |[sits_down, gets_up ]|
   Footman [ ... ] (0)
endhide
```

This solution is free from deadlock, as explained in [Hoa85].

5.6 Readers and writers

This is again a mutual exclusion problem, but now there are two kinds of processes, readers and writers, sharing a resource, for example a database. Readers only examine information in the database, whereas writers modify it. A writer must have exclusive access to the database, otherwise the information could be corrupted. But any number of readers can access the database at a time, provided that there is no writer modifying it.

In this solution, there will be a manager that controls the right access to the database. In a first attempt, readers and writers have to wait for the manager's permission to access the database; when they have it, they can access the database, and then they have to notify their going out to the manager. We can specify the reader and writer behaviours in the following way:

```
process Reader1 [accR:id, abdR:id] (myid:id) is
   (* other things *)
   accR(!myid) ;
   (* read *)
   abdR(!myid) ;
   Reader1 [...] (myid)
endproc

process Writer1 [accW:id, abdW:id] (myid:id) is
   (* other things *)
   accW(!myid) ;
   (* write *)
   abdW(!myid) ;
   Writer1 [...] (myid)
endproc
```

The manager has a boolean parameter, *writing*, that controls whether there is a writer updating the database, and an integer parameter, *readers*, that controls how many readers are reading the database. When a reader wants to access it, the manager will give permission if there is no writer. If a writer wants to access it, the manager will give permission only if there is no writer and no reader. The specification of the manager is:

```
process Manager1 [accR:id,abdR:id,accW:id,abdW:id] (writing:bool,readers:nat) is
  var r:nat:=0,w:nat:=0 in
    sel
        accR(?r) [not(writing)] ; Manager1 [...] (false, readers + 1)
      []
        accW(?w) [not(writing) and (readers = 0)] ; Manager1 [...] (true, readers)
      []
        abdR(?r) ; Manager1 [...] (writing, readers − 1)
      []
        abdW(?w) ; Manager1 [...] (false, readers)
    endsel
  endvar
endproc
```

The complete system with 8 readers and 2 writers is specified as follows:

```
specification ReadersWriters1 import ReadWriteMod is
behaviour
  hide accR:id,accW:id,abdR:id,abdW:id in
      ( par ?r in [1,2,3,4,5,6,7,8] |||
          Reader1 [accR,abdR] (r)
        endpar
      |||
        par ?w in [1,2] |||
          Writer1 [accW,abdW] (w)
        endpar )
    ||
      Manager1 [accR,accW,abdR,abdW] (false,0)
  endhide
endspec
```

where ReadWriteMod is the module that contains the declarations of the processes Reader1, Writer1, and Manager1.

This solution suffers from the *starvation problem.* Once the readers begin using the database, they can monopolize it, without allowing any writer to access it. We can solve this problem if the manager does not allow new reader accesses when there is a writer waiting to update the database. In this way, readers will be finishing (*readers* will become 0) and the writer will be able to access.

In order to introduce this new idea, the writers have to notify their interest of accessing the database, so a new gate, *noteW*, that allows communication between writers and the manager has to be included. The manager will have a new parameter, *wwriters*, to count how many writers are waiting. The specification of the readers is as before, but writers and the manager have to be modified as follows:

```
process Writer2 [noteW:id,accW:id,abdW:id] (myid:id) is
  (* other things *)
  noteW(!myid) ; accW(!myid) ;
  (* write *)
  abdW(!myid) ; Writer2 [...] (myid)
endproc
```

```
process Manager2[accR:id,abdR:id,noteW:id,accW:id,abdW:id]
                (writing:bool,readers:nat,wwriters:nat) is
  var r:nat:=0,w:nat:=0 in
    sel
        accR(?r)[not(writing) and (wwriters=0)];
        Manager2[...](writing, readers + 1, wwriters)
      []
        noteW(?w); Manager2[...](writing, readers, wwriters + 1)
      []
        accW(?w)[not(writing) and (readers=0)];
        Manager2[...](true, readers, wwriters − 1)
      []
        abdR(?r)[readers>0]; Manager2[...](writing, readers − 1, wwriters)
      []
        abdW(?w)[writing]; Manager2[...](false, readers, wwriters)
    endsel
  endvar
endproc
```

where this process will be called from the specification as

Manager2[...](*false*, 0, 0).

This second attempt has introduced the reverse problem: now writers can be always accessing the database, not allowing readers to use it. We can solve this problem if the manager stops the notification of new writers once it knows there are readers waiting. The writers that have already notified their intention can access the database sequentially, but no new writers can notify. In this way the parameter *wwriters* will become 0 and at least one reader will be able to access.

In this attempt, a new gate, *noteR*, will be included, through which readers can notify to the manager their wish to access the database. And the manager has to count how many readers are waiting (parameter *wreaders*). The specification of the reader has to be changed to

```
process Reader3 [noteR:id,accR:id,abdR:id](myid:id) is
  (* other things *)
  noteR(!myid); accR(!myid);
  (* read *)
  abdR(!myid) Reader3 [...](myid)
endproc
```

The writer is the same as in the second attempt, and the new manager is:

```
process Manager3[noteR:id,accR:id,abdR:id,noteW:id,accW:id,abdW:id]
                (writing:bool,readers:nat,wwriters:nat,wreaders:nat) is
  var r:nat:=0,w:nat:=0 in
    sel
        noteR(?r); Manager3[...](writing, readers, wwriters, wreaders + 1)
      []
        accR(?r)[not(writing) and (wwriters = 0)];
        Manager3[...](writing, readers + 1, wwriters, wreaders − 1)
      []
        noteW(?w)[readers > 0 or (wreaders = 0)];
        Manager3[...](writing, readers, wwriters + 1, wreaders)
      []
        accW(?w) [not(writing) and (readers = 0)] ;
        Manager3 [...] (true, readers, wwriters − 1, wreaders)
      []
        abdR(?r) ; Manager3 [...] (writing, readers − 1, wwriters, wreaders)
      []
        abdW(?w) ; Manager3 [...] (false, readers, wwriters, wreaders)
    endsel
  endvar
endproc
```

The manager could be modified not only to know how many readers and writers are waiting, but *who* of them are waiting (as we did in Section 5.4, by means of a queue in a semaphore), so it can control the order in which they are served.

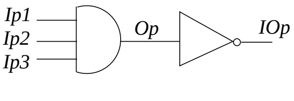
5.7 Specifying digital logic

In this section, we show how we can deal with the specification of digital logic components in E-LOTOS. It is entirely based on the work [TS94] by K. J. Turner and Richard O. Sinnott, and the subsequent work [JT97], by Ji He and K. J. Turner. There, the specification and validation of digital logic components and circuits using LOTOS are addressed, with the philosophy that it should be easy for the hardware engineer to translate a circuit schematic into a LOTOS specification, and then to analyze and verify the properties of this specification. Because of this, a component library (DILL) is introduced.

Here, our aim is simply to show how these jobs can be dealt in E-LOTOS by means of an example: a specification of a full adder.

First, we introduce some basic ideas. Digital signals are going to be modeled as two-level voltages, specified as constants of the E-LOTOS data type Bit. The constant *bit1* represents logic 1, constant *bit0* represents logic 0, and constant *bitX* represents an unknown, arbitrary or "do not care" value, used as the initial state of every signal (both inputs and outputs). This data type is specified in a module we will see below, with the definition of several logical operations on signals.

Each basic logic gate (*Inverter*, *And*, *Or*, *XOr*, ...) is modeled as an E-LOTOS process. For example, And3 [*Ip1*, *Ip2*, *Ip3*, *Op*] is an E-LOTOS specification of an *And* gate with three inputs. An E-LOTOS gate (for example, *Ip1*) models a physical wire or pin, and an E-LOTOS action (for example, *Ip1*(! *bit1*)) models a signal change on the wire (here, from logic 0 to logic 1). Larger circuits can be built from basic logic gates using E-LOTOS parallel behaviours. For example, an And3 gate followed by an Inverter,



can be modeled as:

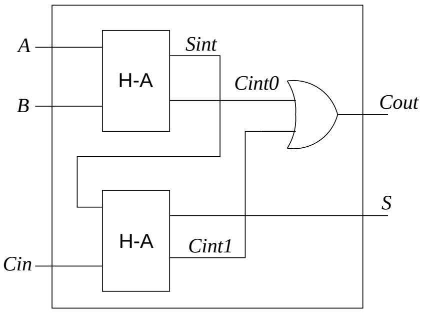
```
And3 [ Ip1, Ip2, Ip3, Op ] ()  |[Op]|  Inverter [ Op, IOp ] ()
```

Thus, connecting several wires and pins is modeled as synchronization at E-LOTOS gates. We package the specification of a circuit into an E-LOTOS process in order to reuse it. The E-LOTOS gates of the process are the inputs and outputs of the circuit, and all the other E-LOTOS gates are hidden. For example, the circuit above can be specified as:

```
process And3Inverter [ Ip1:Bit, Ip2:Bit, Ip3:Bit, IOp:Bit ] is
```

```
    hide Op:Bit in
        And3[Ip1,Ip2,Ip3,Op]()
      |[Op]|
        Inverter[Op,IOp]()
    endhide
  endproc
```

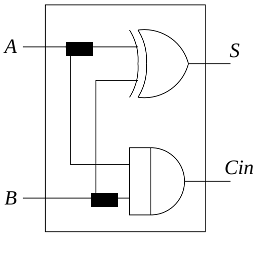
After this introduction, now we are going to specify a full-adder, as explained in [Flo94]. A full-adder accepts three inputs including an input carry and generates a sum output and an output carry. A full-adder can be built from two half-adders and an Or2 gate, as the following logic diagram shows:



Thus, we can specify a process Full − Adder as follows

```
  process Full − Adder[A,B,Cin,S,Cout] is
    hide Sint:Bit,Cint0:Bit,Cint1:Bit in
        (
          Half − Adder[A,B,Sint,Cint0]()
        |[Sint]|
          Half − Adder[Sint,Cin,S,Cint1]()
        )
      |[Cint0,Cint1]|
        Or2[Cint0,Cint1,Cout]()
    endhide
  endproc
```

A half-adder accepts two binary digits on its inputs and produces two binary digits on its outputs, a sum bit and a carry bit. A half-adder can be built from an And2 gate and a XOr2 gate, connected in the following way:



We can specify it in E-LOTOS with the following process:

```
process Half − Adder [A:Bit,B:Bit,S:Bit,Cout:Bit] is
    Xor2[A,B,S]()
  |[A,B]|
    And2[A,B,Cout]()
endproc
```

We have reached the logic gate level. We have to specify now the logic gates Or2, XOr2, and And2. As is done in [TS94, JT97], we can specify a process that implements logic gates with two inputs and which is parameterized by the binary logical operation that it must implement. The process is Logic2:

```
process Logic2[Ip1:Bit,Ip2:Bit,Op:Bit](bOp:BitOp) is
  var bIn1:Bit:=bitX, bIn2:Bit:=bitX, bOut:Bit:=bitX, bOutNew:Bit:=bitX in
    loop
      sel
          Ip1(?bIn1)
        []
          Ip2(?bIn2)
        []
          ?bOutNew:=Apply2(bOp,bIn1,bIn2);
          sel
            Op(?bOut2) [(bOutNew = bitX) and (bOut = bitX) and(bOut2 <> bitX)];
            ?bOut:=bOut2
          []
            Op(!bOutNew) [(bOutNew <> bitX) and (bOutNew <> bOut)];
            ?bOut:=bOutNew
          endsel
      endsel
    endloop
  endvar
endproc
```

Variables $bIn1$, $bIn2$, and $bOut$ save the state of the pins. The first two branches of the outer selection describe the possibility of new inputs. If the value of an input pin changes, it is saved in the corresponding variable. The third branch describes the output. Variable $bOutNew$ saves the value of the application of the binary logical operation bOp to the values of the inputs. The second branch of the inner selection describe the case when the inputs are known (that is, they are not $bitX$) and then the output is also known ($bOutNew$<>$bitX$), and there is a change in the output ($bOutNew$<>$bOut$). The first branch describes the case when the output is unknown, but the output pin takes a known value from outside, for example, because there is feedback.

And now, by instantiating this process we can specify the needed logic gates:

```
process Or2[Ip1:Bit,Ip2:Bit,Op:Bit] is
  Logic2[Ip1,Ip2,Op](orOp)
endproc
process XOr2[Ip1:Bit,Ip2:Bit,Op:Bit] is
  Logic2[Ip1,Ip2,Op](xorOp)
endproc
process And2[Ip1:Bit,Ip2:Bit,Op:Bit] is
  Logic2[Ip1,Ip2,Op](andOp)
endproc
```

Finally, we have to specify the Bit data type, with the logical operations.

```
module Bit_Mod is
type Bit is
  bit1 | bit0 | bitX
endtype

function not(b:Bit):Bit Bis
  case b is
      bit0 -> bit1
    | bitX -> bitX
    | bit1 -> bit0
  endcase
endfun

function or(b1:Bit,b2:Bit):Bit is
  case (b1,b2) is
      (?b:Bit,bit0) -> b
    | (bit0,bitX) -> bitX
    | (bitX,bitX) -> bitX
    | (bit1,bitX) -> bit1
    | (?b:Bit,bit1) -> bit1
  endcase
endfun

function and(b1:Bit,b2:Bit):Bit is
  case (b1,b2) is
      (?b:Bit,bit0) -> bit0
    | (bit0,bitX) -> bit0
    | (bitX,bitX) -> bitX
    | (bit1,bitX) -> bitX
    | (?b:Bit,bit1) -> b
  endcase
endfun

function xor(b1:Bit,b2:Bit):Bit is
  case (b1,b2) is
    (?b:Bit,bit0) -> b
    | (?b:Bit,bitX) -> bitX
    | (?b:Bit,bit1) -> not(b)
  endcase
endfun

type BitOp is
  orOp | andOp | xorOp
endtype

function Apply2(bOp:BitOp,b1:Bit,b2:Bit):Bit is
  case bOp is
      orOp -> or(b1,b2)
    | andOp -> and(b1,b2)
    | xorOp -> xor(b1,b2)
  endcase
endfun
endmod
```

Bibliography

[BB93] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3:142–188, 1993.

[Ber93] Gérard Berry. Preemption in concurrent systems. In *Proceedings of FSTTCS 93*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93, Berlin, 1993. Springer Verlag.

[BL93] Rezki Boumezbeur and Luigi Logrippo. Specifying telephone systems in Lotos. *IEEE Communications Magazine*, pages 38–45, August 1993.

[Bri88] Ed Brinksma. *On the Design of Extended LOTOS, a Specification Language for Open Distributed Systems.* PhD thesis, University of Twente, November 1988.

[dFLL$^+$95] D. de Frutos, G. Leduc, L. Léonard, L. Llana, C. Miguel, J. Quemada, and G. Rabay. Time extended LOTOS. In *Working draft on Enhancements to LOTOS.* ISO/IEC JTC1/SC21/WG1, 1995.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University Eindhoven, 1965.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[Flo94] Thomas L. Floyd. *Digital Fundamentals.* Macmillan Publishing Company, 1994.

[Gam90] Mark Gamble. The CCSDS protocol validation programme inter-agency testing using LOTOS. In Juan Quemada, Jose A. Mañas, and Enrique Vázquez, editors, *Proc. Formal Description Techniques III.* North-Holland, Amsterdam, Netherlands, November 1990.

[GH93] Hubert Garavel and Rene-Pierre Hautbois. Experimenting with LOTOS in the aerospace industry. In Teodor Rus and Charles Rattray, editors, *Theories and Experiences for Real-Time System Development*, Computing: Vol 2. World Scientific, 1993.

[GS96] Hubert Garavel and Mihaela Sighireanu. On the introduction of exceptions in LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 469–484. Chapman & Hall, October 1996.

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

[ISO89] ISO. *Information Processing Systems – Open Systems Interconnection – LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* IS-8807. International Standards Organization, Geneva, September 1989.

[JT97] He Ji and Kenneth J. Turner. Extended DILL: Digital logic in LOTOS. Technical Report CSM-142, Department of Computing Science and Mathematics, University of Stirling, UK, November 1997.

[LDV99] Luis Llana-Díaz and Alberto Verdejo. Time and urgency in E-LOTOS. Submitted for publication, March 1999.

[LFHH90] Luigi M. S. Logrippo, Mohammed Faci, and Mazen Haj-Hussein. An introduction to Lotos: Learning by examples. Technical Report TR-90-14, University of Ottawa, Ottawa, Canada, March 1990.

[LL94] Guy Leduc and Luc Lonard. A formal definition of time in LOTOS. ISO/IEC, 1994. Source: JTC1/SC21/WG1/Q48.6.

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

[Mun91] Harold B. Munster. LOTOS specification of the MAA standard, with an evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Computer Aided Design*, volume 575 of *Lecture Notes in Computer Science*, pages 376–398, 1991.

[NS94] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[Pec92] Charles Pecheur. Using LOTOS for specifying the CHORUS distributed operating system kernel. *Computer Communications*, 15(2):93–102, March 1992.

[Pec94] Charles Pecheur. A proposal for data types for E-LOTOS. Technical report, University of Liège, October 1994. Annex H of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[Que98] Juan Quemada, editor. Final committee draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG7 Project 1.21.20.2.3., May 1998.

[RR86] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *Proc. ICALP 86*, pages 314–323. Springer-Verlag, 1986. LNCS 226.

[Sch95] S. A. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, 1995.

[TS94] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.

[vEVD89] Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz. *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project.* North-Holland, New York, 1989.

[Yi91] Wang Yi. CCS + time = an interleaving model for real time systems. In J. Leach Albert, B. Monien, and M. Rodríguez, editors, *Proc. ICALP 91*, pages 217–228. Springer-Verlag, 1991. LNCS 510.