3 Introduction to LOTOS[1]

LOTOS was developed to define implementation-independent formal standards of OSI services and protocols. 'LOTOS' stands for Language Of Temporal Ordering Specification because it is used to model the order in which the events of a system occur. LOTOS has two very clearly separated parts. The first part provides a behavioural model derived from process algebras, principally from CCS (*Calculus of Communicating Systems*, Milner (1989)) but also from CSP (*Communicating Sequential Processes*, Hoare (1985)). The second part of the language allows specifiers to describe abstract data types and values, and is based on the abstract data type language ACT ONE (Ehrig and Mahr (1985)).

These two aspects of LOTOS guided the organisation of this chapter. Section 3.1 introduces basic concepts of the language. Section 3.2 presents the process algebra aspect of LOTOS, called **Basic LOTOS**. Section 3.4 presents the ACT ONE basis of data typing. Finally, Section 3.5 indicates how these aspects are combined as **Full LOTOS**.

The syntax and semantics of LOTOS are defined in the relevant standard (ISO (1989l)). The definition has four main parts: the syntax, the static semantics, the algebraic semantics of data types, and the dynamic semantics of behaviour expressions. A description of the detailed semantics of LOTOS is beyond the scope of this introductory chapter.

3.1 Basic Concepts

Specification languages have been developed to allow modelling of systems for the purpose of analysis and design. LOTOS uses the concepts of process, event and behaviour expression as basic modelling concepts.

3.1.1 *Processes*

Systems and their components are represented in LOTOS by **processes**. A process displays an **observable behaviour** to its environment in terms of

[1] Chapter 3 is by J. Quemada, L. Ferreira Pires, J. A. Mañas, A. Azcorra and T. Robles.

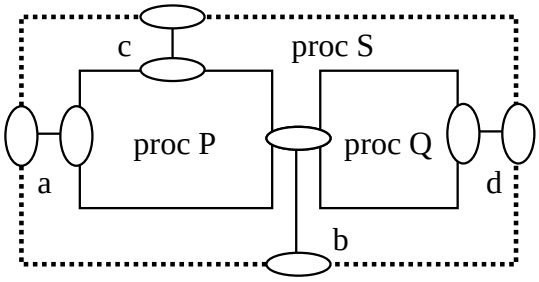

Figure 3.1: Process Structure Example

permitted sequences of observable actions. A process appears as a **black box** to its environment since the environment has no information about its internal structure and mechanisms. LOTOS processes interact with each other through **gates**.

Figure 3.1 illustrates a LOTOS process structure. In this figure, process S represents a system that interacts with its environment via gates a, b, c and d. These gates model the logical or physical attachment points between a system and its environment. Process S is the composition of processes P and Q. Process P interacts with its environment through gates a, b and c, while process Q interacts with its environment through gates b and d. Processes P and Q interact through their common gate b, and are therefore considered to belong to the environment of each other.

Figure 3.1 could correspond to the following LOTOS process definitions:

```
process S [a, b, c, d] : noexit :=

  P [a, b, c] |[b]| Q [b, d]            (* behaviour expression *)

  where

  process P [t, u, v] : noexit :=
    t; (u; stop [] v; stop)             (* behaviour expression *)
  endproc (* P *)

  process Q [x, y] : noexit :=
    x; y; stop                          (* behaviour expression *)
  endproc (* Q *)

endproc (* S *)
```

The identifiers *S*, *P* and *Q* in the example designate the corresponding processes in the LOTOS text. *P [a, b, c]* represents an instantiation of process *P*, while *Q [b, d]* represents an instantiation of process *Q*. Notice that the gate structure of these processes is explicitly described in the process instantiation through gate identifier lists (*[a, b, c]* and *[b, d]*). The operator |[*b*]| between *P* and *Q* states that these processes interact through their common gate *b*.

Process declarations are delimited by the reserved words **process** and **endproc**. Process definitions have a process identifier, a formal gate list, an optional parameter list, the process functionality, and a behaviour expression. All these language elements are discussed later. Note that process *P* in the example is declared with formal gate list *[t, u, v]* which is renamed to *[a, b, c]* when the process is instantiated (called).

In LOTOS, a **specification** is a special kind of process, namely the one that represents the whole system. The difference between a process and a specification is only syntactic, as will be explained later.

3.1.2 *Events*

LOTOS specifications describe observable behaviour of systems. The observable behaviour is the set of all possible sequences of interactions in which the system is allowed to participate. Therefore the concept of interaction is fundamental in the LOTOS model. Interactions are represented in LOTOS by **events**. Events are atomic, instantaneous and synchronous instances of interaction. Each event is associated with a gate, namely the gate at which the event takes place.

Events model real-life occurrences in an abstract way. The degree of detail required in the model determines the events to be considered. For example, data transmission through an interface could be modelled by a single event representing the whole transmission. Alternatively, the beginning and the end of the transmission could be modelled as distinct events, or events could correspond to the transmission of individual bits. All these models represent the same data transmission at different degrees of detail.

Events can only occur if *all* processes that are supposed to participate in it are ready to interact. When an event takes place, all the processes involved in the event synchronise and have a common view of the interaction. This common view is interpreted as synchronisation and communication, and guarantees that processes participating in interactions have access to the same interaction parameter values. The concept of structured event introduced in Section 3.5 clarifies these ideas.

3.1.3 *Behaviour Expressions*

The observable behaviour of a system is described in LOTOS by means of a language construct in which the sequences of allowed events are defined. This

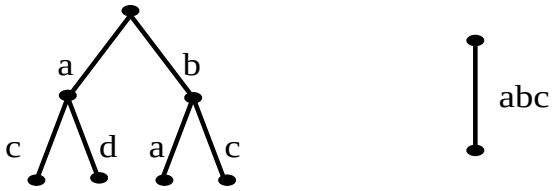

Figure 3.2: Examples of Behaviour Trees

construct is called a **behaviour expression** that defines sequences precisely in terms of the LOTOS semantic model.

Behaviour expressions can be represented graphically as **behaviour trees**. This representation is helpful in visualising the sequence of events and their dependencies, but its practical use is limited to simple cases of behaviour. Figure 3.2 depicts two examples of behaviour trees. Time in these diagrams runs down the page. Nodes represent states of a system. Arcs between nodes represent transitions between states, and are labelled with the corresponding actions.

3.2 Basic LOTOS

LOTOS without ACT ONE is called **Basic LOTOS**. Experience shows that Basic LOTOS is easier to understand than Full LOTOS; furthermore, Basic LOTOS can be generalised to Full LOTOS in a quite straightforward way.

3.2.1 *Basic Constructs*

The basic LOTOS operators are the ones which allow the representation of any finite behaviour. Other operators are introduced in order to cope with structuring, readability, and repetitive behaviour. The basic operators deal with inaction (**stop**), action prefix (';'), and choice ('[]').

Inaction

Inaction (specified with **stop**) models a situation in which a process is unable to interact with its environment. Inaction can be used to describe **deadlock**, i.e. a situation in which no more interactions are possible. Inaction is a degenerate case of behaviour. It is a kind of null behaviour expression (like an empty set) since it models the *absence* of behaviour. In behaviour trees, inaction corresponds to a node that does not lead to further branches.

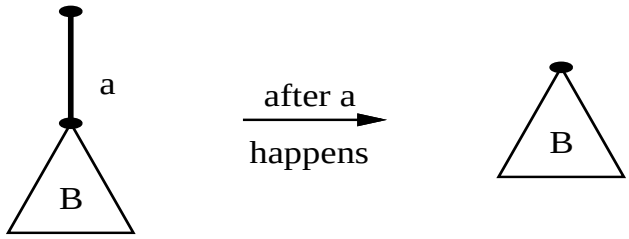

Figure 3.3: Action Prefix

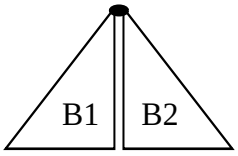

Figure 3.4: Choice Construct

Action Prefix

When representing observable behaviour, it is often necessary to indicate that an event occurs before other events. The **action prefix construct** is used when an event must occur before the following behaviour expression. The syntax element that represents an event in LOTOS is called an **action denotation**. Figure 3.3 depicts the behaviour tree associated with the generic behaviour expression *a; B* where *a* is an action denotation and *B* is a behaviour expression.

Some behaviour expressions illustrating action prefix are:

Error; (Crash; **stop**)

TelephoneFriend; Conversation; **stop**

Choice

The **choice construct** selects one of two alternative behaviours. Given behaviour expressions B_1 and B_2, B_1 *[]* B_2 behaves as B_1 or B_2 depending on whether the next event is the initial one of B_1 or B_2 respectively. Figure 3.4 shows a graphical representation of the choice operator.

Consider modelling the behaviour of a drink dispenser that accepts 10 and 20 cent coins. Introducing a 10 cent coin results in a cup of tea, while introducing a 20 cent coin results in a cup of coffee. This could be specified as:

(Coin10c; Tea; **stop**) [] (Coin20c; Coffee; **stop**)

In the behaviour expression above, B_1 is *(Coin10c; Tea;* **stop***)* and B_2 is *(Coin10c; Tea;* **stop***)*. If the environment (a user of the drink dispenser) offers *Coin10c*, behaviour B_1 can be selected; after the interaction, only the *Tea* event can occur, meaning that only tea can be delivered. Similarly, the *Coin20c* event will select behaviour B_2, and only coffee can be delivered.

3.2.2 *Internal Event*

So far, observable behaviour has been described in terms of events that represent interactions between a system and its environment. However, a system may make an internal unobservable decision that affects its future behaviour; the *effect* of the decision may be significant.

The **internal event**, represented by **i**, is a special LOTOS event that models occurrences or decisions that are internal and therefore invisible to the environment. Although invisible, the occurrence of an internal event may modify the subsequent externally observable behaviour. Examples of real-life occurrences that may be modelled as internal events are timeout, system failures, and lack of resources. The following represents a timeout situation:

(ExpectedAction; **stop**) [] (**i**; TimeOut; **stop**)

Non-determinism

From the point of view of the environment, **non-deterministic behaviour** occurs when there are multiple possibilities for behaviour that cannot be controlled by the environment. In such a case, the system may react in different ways on different occasions to the same sequence of interactions.

Non-determinism can be modelled in LOTOS in three ways, combining choice and the internal event. Suppose that P and Q are behaviour expressions and that a is an event:

(a; P) [] (**i**; Q) (* case 1 *)

(a; P) [] (a; Q) (* case 2 *)

(**i**; P) [] (**i**; Q) (* case 3 *)

Case 1 says that event a can take place initially, unless the internal event occurs and makes the system behave like Q. In this case the environment has some chance to influence the system if it acts before the internal event. Note that the concept of time is immaterial in LOTOS, so 'acting before' refers to an interpretation of the internal event as a timer or some internal activity that takes time. Case 2 says that the system decides at the occurrence of the interaction which one of the a events will take place. Case 3 says that the system will decide internally whether it should behave according to P or Q. In both cases 2 and 3, the environment cannot influence the outcome of the decision.

The following example deals with the behaviour of an airline reservation system:

```
RequestSeat;
(
  i; SeatConfirmed; stop
[]
  i; NoSeatsAvailable; stop
[]
  i; SystemNotAvailable; stop
)
```

The result of a request for reservation of seats on a flight is completely unpredictable from the point of view of a client, since a normal client does not know how the system is organised and whether there are seats available in advance. The inability of the client to influence the system is modelled with internal events.

3.2.3 *Recursion*

A behaviour is very often repetitive or even infinite. With the language constructs introduced so far this is not possible since the specification text must be finite. However, LOTOS allows repetition of behaviour to be modelled with **recursive** process instantiations.

Consider again the drink dispenser of Section 3.2.1. After one coin has been inserted, this machine can deliver tea or coffee and then stop. This is probably not the way the drink dispenser machine should operate since a dispenser that delivered only one cup of tea or coffee would be a little impractical! A more realistic dispenser could be specified as:

```
process DrinkDispenser [Coin10c, Coin20c, Tea, Coffee] : noexit :=
  Coin10c; Tea; DrinkDispenser [Coin10c, Coin20c, Tea, Coffee]
[]
  Coin20c; Coffee; DrinkDispenser [Coin10c, Coin20c, Tea, Coffee]
```

endproc (* DrinkDispenser *)

This example uses recursion to indicate that the behaviour is repetitive: after a cup of tea or coffee has been delivered then more can be dispensed.

Another example to illustrate recursion is a communications polling device. This example deals with only the polling operation, and supposes the existence of three stations:

process PollingOperation [StationA, StationB, StationC] : **noexit** :=
  StationA;
  PollingOperation [StationB, StationC, StationA]
endproc (* PollingOperation *)

Here the actual gate list of the recursive instantiation is cycled one place relative to the formal gate list in the process definition. This allows a cyclic repetition of events at gates *StationA*, *StationB* and *StationC*, characterising the polling.

3.2.4 *Exit*

The model of behaviour termination introduced by inaction (**stop**) is very rough since defines only abrupt termination. This kind of termination does not allow a sequence of processes, for example, which is sometimes a useful concept. LOTOS models successfully terminating processes though the **exit** construct. The interpretation of **exit** is that a special termination event[2] takes place before **stop**; this special event indicates successful termination and is distinct from any ordinary event.

Consider the following example of a login procedure:

process LoginProcedure [LoginReq, LoginConf, LoginAbort] : **exit** :=
  LoginReq;
  (
    i; LoginConf; **exit**
  []
    i; LoginAbort; LoginProcedure [LoginReq, LoginConf, LoginAbort]
)
endproc (* LoginProcedure *)

In the example, the occurrence of *LoginAbort* is followed by a new login attempt, but the occurrence of *LoginConf* is followed by the special termination event to indicate successful termination of the login procedure. Section 3.2.7 indicates how **exit** can be used in the context of sequential composition of processes.

[2]The special event denoting successful termination is called δ.

3.2.5 *Parallel Composition*

Very often specifiers have to structure specifications or to represent design structures. These structures, in the case of distributed systems, are generally parallel instances of functionality. In LOTOS, behaviour expressions can be combined using the **parallel construct**.

Consider the general behaviour expression B_1 $|[g_1,\ldots,g_n]|$ B_2 in which B_1 and B_2 are behaviour expressions and $g_1,\ldots,g_n$ is a gate identifier list. In this behaviour expression, events at gates that belong to the gate identifier list can occur only with the participation of both B_1 and B_2. Events at gates that do not belong to the gate identifier list can occur with the participation of either B_1 or B_2 alone.

Return to the example of Figure 3.1. The behaviour expression:

P [a, b, c] |[b]| Q [b, d]

says that behaviour expressions *P [a, b, c]* and *Q [b, d]* synchronise at gate *b*. In this case the behaviour expressions are process instantiations, but may not be in general. As another example, consider the following behaviour expression:

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

Events at gates *a* and *c* in the example above can occur only with the participation of both behaviour expressions; events take place at the other gates (by inspection, *b* and *d*) with the participation of only one of the behaviour expressions.

Special Cases

Two special cases of the parallel operator can be identified by considering the extreme cases of pure interleaving and full synchronisation. These two extreme cases can be represented by language shorthands.

In the case of **pure interleaving**, interactions of the two behaviour expressions are completely interleaved, i.e. they always occur with the participation of only one of the behaviour expressions. This corresponds to the case in which the gate identifier list is empty; the shorthand notation for pure interleaving is '|||'. An example of this is the behaviour expression:

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

The equivalent[3] behaviour expression in terms of the action prefix and choice operator can be deduced by inspection:

[3]The behaviour expressions are considered to be **observationally equivalent** to each other, and so represent the same semantic model. The formalisation of the equivalence relation used falls outside the scope of this chapter.

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

In the case of **full synchronisation**, interactions at any gate of the two behaviour expressions are fully synchronised, i.e. they occur with the participation of both behaviour expressions. Syntactically this is the case in which the gate identifier list contains all the gate identifiers appearing in both behaviour expressions; the shorthand notation for full synchronisation is '||'. An example of this is the behaviour expression:

```
  (a; stop [] b; c; stop)
||
  (a; stop [] i; b; c; stop [] d; stop)
```

The equivalent behaviour expression in terms of the action prefix and choice operators is:

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

Note that behaviour expressions do not synchronise on internal events. This is due to the fact that internal events are not observable.

Multi-way Synchronisation

The concept of synchronisation occurring between two instances of activity can be generalised to more than two instances, i.e. **multi-way synchronisation**. LOTOS supports multi-way synchronisation through the combinational character of the parallel construct.

As an example, take process instances *P [a, b, c]*, *Q [a, c]* and *R [a, b]*. The following expression says that events at gate *a* can happen only with the participation of all three processes:

```
P [a, b, c] |[a]| Q [a, c] |[a]| R [a, b]
```

This example says that events at gate *a* in *Q* and in *R* can happen only with the participation of both processes. *Q [a, c] |[a]| R [a, b]* is a new behaviour expression, so events at gate *a* in *P* and in this new behaviour expression (and thus in *Q* and *R*) can happen only with the participation of both behaviour expressions. The conclusion is that interactions at gate *a* in the example above can happen only with the participation of all processes, *P*, *Q* and *R*.

3.2.6 *Hiding*

The parallel composition presented so far has a drawback: behaviour expressions cannot be 'protected' from interactions with their environment. This problem becomes clear with the example in Figure 3.1. There the environment of process S interacts with processes P and Q. (Notice the parallel operator $|[b]|$.) To say that events at gate b represent some kind of internal (unobservable) interaction, it is necessary to be able to 'hide' this gate from the environment. This can be done using the **hide construct**. The example of Figure 3.1 can be rewritten as follows:

```
process S [a, c, d] : noexit :=

  hide b in
    P [a, b, c] |[b]| Q [b, d]

  where
  ...
```

In this new specification of process S, events at gate b are not accessible to the environment. The **hide** operator contains a gate identifier list in which the specifier indicates which gates are unobservable by the environment of the behaviour expression. In the example the list has a single element, b. Observe that the gate list in the declaration of process S does not have b, indicating that gate b is no longer accessible to the environment of S.

3.2.7 *Sequential Composition*

Action prefix was introduced to allow the explicit representation of temporal ordering of events. Nevertheless, it is very often necessary to represent temporal ordering of behaviours. This occurs when the system presents well-determined phases. In Lotos, temporal ordering of phases can be represented using sequential composition of behaviour expressions with the **enabling construct** '>>'.

Consider the generic behaviour expression $B_1 >> B_2$. The behaviour expression B_2 is **enabled** by the behaviour expression B_1 if the special δ event (**exit**) occurs in B_1. As explained earlier, **exit** indicates successful termination and allows the sequential composition of behaviour expressions. Notice that if B_1 does not terminate successfully (e.g. it deadlocks), B_2 will never be allowed.

Consider a file transfer procedure that starts with a login procedure like the one in Section 3.2.4. The behaviour expression for this system could have the following outline:

```
LoginProcedure [...] >> FileTransfer [...] >> LogoutProcedure [...]
```

3.2.8 *Disabling*

It is very often possible to identify 'normal' behaviour in systems which can be disrupted at any moment by some exceptional circumstance. Examples of exceptional circumstances are interrupts and errors. In LOTOS, disruption of behaviour is represented by the **disabling construct** '[>'.

The generic behaviour expression B_1 [> B_2 behaves like B_1 until an event of B_2 occurs. B_2 is said to **disable** B_1 in this case. After an event of B_2 occurs, the future behaviour is that of B_2. If B_1 terminates successfully (a δ occurs), B_2 no longer applies.

The following behaviour expression illustrates disabling:

(a; b; **exit**) [> (d; **stop**)

The equivalent behaviour expression in terms of action prefix and choice operators is:

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

3.3 Basic LOTOS Examples

The Basic LOTOS operators will be illustrated with a simplified treatment of the two-key system covered in Chapter 11. This is an access control system that requires insertion of two keys before allowing access. Both keys must be inserted before one access is allowed. The keys can be extracted only after the access has occurred.

The following subsections illustrate the basic operators by providing a LOTOS behaviour expression and the corresponding semantics (behaviour tree) on the right.

The first step is the definition of the observable interface of the system. There are five abstract events for creating a model of the two-key system: *In1*, *In2*, *Access*, *Out1*, *Out2*. The first two of these represent the insertion of the keys, the third represents the access, and the last two represent the extraction of the keys.

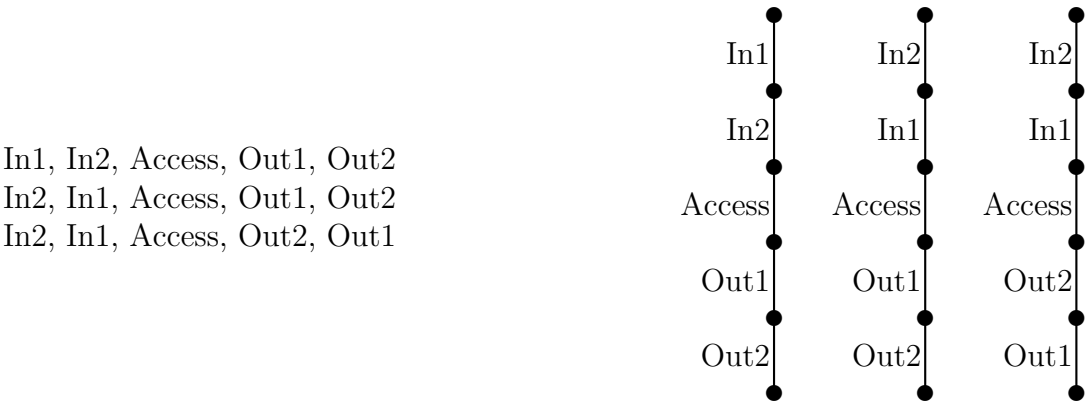

Figure 3.5: Action Prefix Operator for Two-Key System

3.3.1 *Action Prefix*

The action prefix construct allows the specification of sequences of events. This is not enough for representing the complete behaviour of the system. Sequences of events represent just possible execution traces. Some valid sequences are shown in Figure 3.5.

3.3.2 *Choice*

The action prefix and choice constructs allow the representation of trees. Now the complete behaviour of the system can be represented. The representation of the whole behaviour of the system in a tree-like form is only practicable when the number of states is small, as in the example of Figure 3.6.

3.3.3 *Processes*

Processes are used for many purposes. One important use is the creation of infinite behaviours by the use of recursion. Process *TwoKey* in Figure 3.7 allows an unlimited number of access cycles. Another important use of processes is the encapsulation of generic behaviour patterns to avoid duplication, as in process *Acc*. Gate relabelling is also used in this example.

3.3.4 *Enabling*

Enabling and **exit** allow the behaviour of a specification to be structured in phases, three in this example. Phase 1 is the introduction of both keys. Phase 2 concerns access. Phase 3 is the extraction of both keys.

```
  In1;
  In2;
  Access;
  (Out1; Out2; stop [] Out2; Out1; stop)
[]
  In2;
  In1;
  Access;
  (Out1; Out2; stop [] Out2; Out1; stop)
```

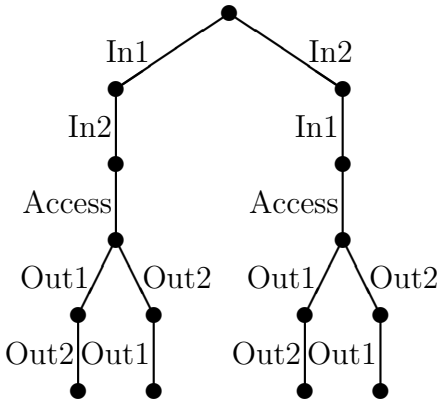

Figure 3.6: Choice Operator for Two-Key System

```
TwoKey [In1, In2, Access, Out1, Out2]

where

process TwoKey [I1, I2, Ac, O1, O2] : noexit :=
  K1; K2; Acc [I1, I2, Ac, O1, O2]
[]
  K2; K1; Acc [I1, I2, Ac, O1, O2]
endproc (* TwoKey *)

process Acc [I1, I2, Ac, O1, O2] : noexit :=
    Ac;
    (
      O1; O2; TwoKey [I1, I2, Ac, O1, O2]
    []
      O2; O1; TwoKey [I1, I2, Ac, O1, O2]
    )
endproc (* Acc *)
```

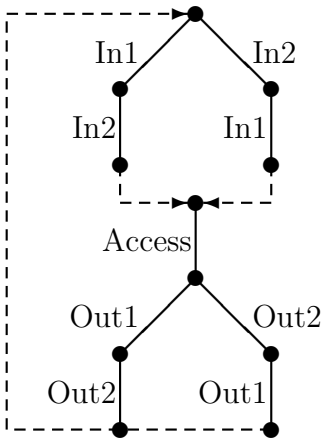

Figure 3.7: Processes for Two-Key System

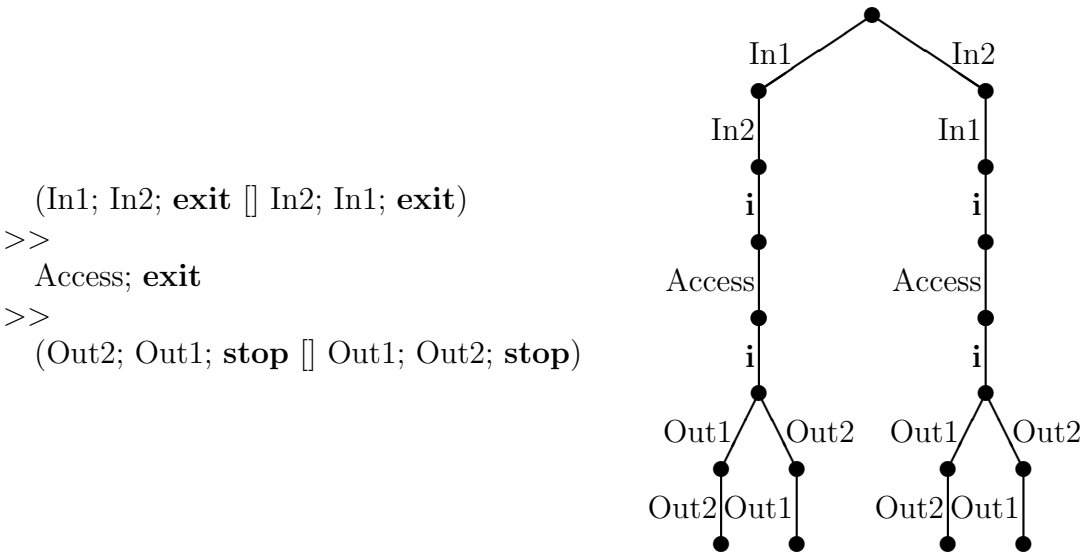

Figure 3.8: Enable Operator for Two-Key System

Notice that internal events in the behaviour tree of Figure 3.8 appear as a consequence of the way the enabling operator works. The appearance of these internal events does not modify the observable behaviour.

3.3.5 *Interleaving*

The first phase is now modelled as interleaved behaviour that deals with the insertion of keys in either order. The upper part of Figure 3.9 shows the trees of the two interleaved behaviours. The tree of the resulting interleaving composition is shown in the lower part of the figure. Both behaviours synchronise on δ despite being composed by interleaving.

3.3.6 *Synchronisation*

An alternative specification for the system uses the synchronisation operator instead of enabling. The system can be specified as a number of independent constraints on valid sequences of behaviour. The independent constraints are then composed with the parallel operator to define the system. Such a specification style is called **constraint-oriented**. A little study will show that the behaviour expression in Figure 3.10 describes the intended behaviour. The behaviour trees of the parenthesised expressions are shown on the right.

Figure 3.9: Interleaving Operator for Two-Key System

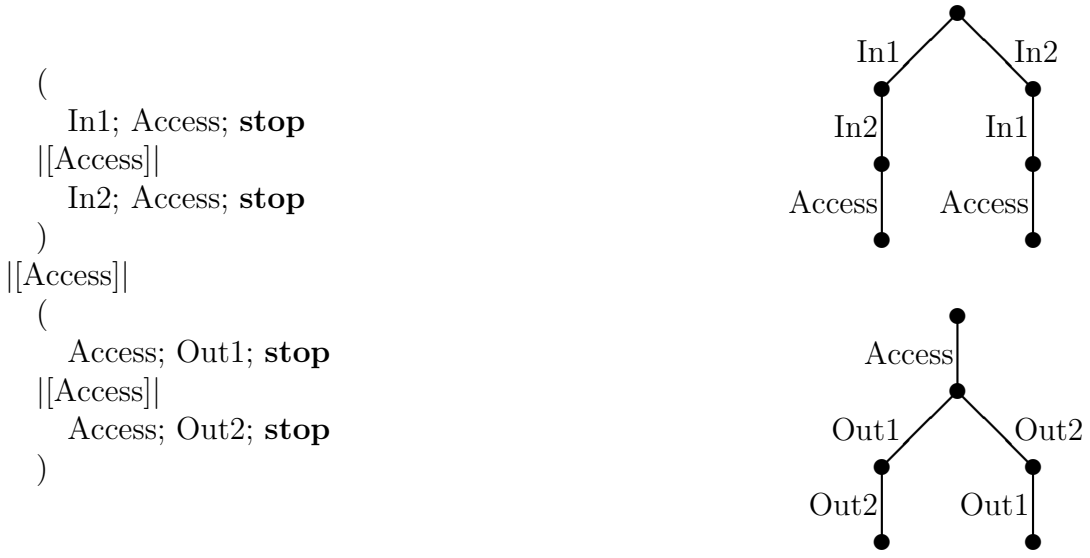

Figure 3.10: Synchronisation Operator for Two-Key System

```
hide In2, Out2 in
   In1;
   In2;
   Access;
   (Out1; Out2; stop [] Out2; Out1; stop)
[]
   In2;
   In1;
   Access;
   (Out1; Out2; stop [] Out2; Out1; stop)
```

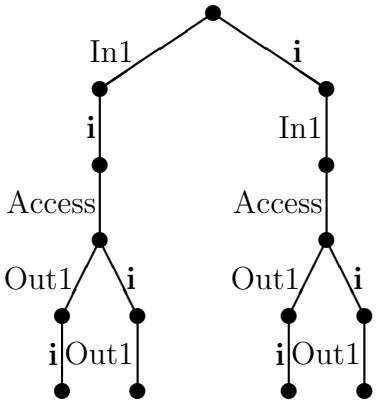

Figure 3.11: Hide Operator for Two-Key System

3.3.7 *Hiding*

Hiding provides a powerful abstraction mechanism. Hiding could be used with this example to obtain a one-key system out of one of the previous specifications of the two-key system. It is only necessary to hide gates *In2* and *Out2* as in Figure 3.11 to achieve this. Although there are internal events composed with the choice operator, the behaviour of the system is deterministic.

3.4 Data Types

Lotos models data as **abstract data types** based on Act One. The underlying theory is that of equational models. These have complex semantics and require complex reasoning about data terms. Since conventional methods of equational reasoning have to be applied, there are difficulties in theorem-proving.

Fortunately, most data types in real specifications may be accurately treated like conventional data structures in a programming language, with functions and procedures to manipulate them. This section presents an informal and intuitive view of data types.

3.4.1 *Basic Data Type Concepts*

There is *no* predefined data type. A Lotos specification starts with no built-in data types. Even Booleans (truth values), explicitly required by language constructs like guards and predicates, must be defined in each Lotos specification.

However, commonly required data types can be included from the **standard library** to save time and space. For example:

library
Boolean, Set, NaturalNumber
endlib

but specifiers are completely free to use whichever data types they wish, even modifying the usual understanding of what Booleans are.

Data specifications are collected into **type** definitions. LOTOS types are not programming language types, but rather a world of things and properties. In a programming language, a Boolean type is typically a set of data values with predefined operations. In LOTOS a Boolean type is a set of data values and operations that require to be defined. Although the LOTOS view is compatible with that of a programming language, a LOTOS type also deals with properties of operations and values.

LOTOS **sorts** are distinct sets of data values. The concept of sort in LOTOS corresponds to the concept of type in many programming languages. Expressions have a definite sort. A variable can hold a value of only a specific sort.

LOTOS **operations** correspond to functions and procedures to manipulate objects. By means of operations it is possible to combine values of the same or different sorts into aggregate values, or establish relations between them.

LOTOS **equations** define which expressions are considered equal, possibly using variables that are universally quantified. For example:

$$\forall x : Nat \;.\; x + 0 = 0$$

Equations state properties that must be satisfied by (any implementation of) the objects of the type. Equational reasoning must therefore be applied to value expressions to find out whether they are equal or not. In theory this may require a complex proof, or may even be unprovable.

In actual practice, however, specifiers usually write equations that can be treated as **rewrite rules**. A rewrite rule handles transformations of expressions. The equation $LHS = RHS$ ('Left-Hand Side equals Right-Hand Side') can be treated as a rewrite rule $LHS \Rightarrow RHS$. The rewrite rule is interpreted 'an expression that matches the LHS pattern can be rewritten according to the RHS scheme'.

The intention of rewriting is to reduce an expression to its **canonical form**. If rewriting is successful, then dealing with (in)equality of two terms becomes trivial: rewrite each term to its canonical form and see whether they are identical or not.

The canonical form is unique, and is sometimes called the **normal form**. The canonical form has the following properties:

- it cannot be rewritten further
- two expressions that can be proved to be equal must yield the same canonical form after applying as many rewrites as possible.

Canonical forms are, loosely speaking, the forms used in computer programming to hold data. Data values are held in arrays, records, sets, etc. All these are (usually) directly stored in canonical form. For example, the expression *3 + 2* would normally be reduced to its canonical form of *5* rather than be preserved as an expression involving '+'.

Canonical forms are convenient, and rewriting is a simple procedure to understand. But, is it possible to avoid rewriting loops? Will the canonical form result after applying every possible rewriting? If there are several possible left-hand sides that match, is it irrelevant what order the rewriting is carried out?

For a large range of practical cases the answers to these questions are positive, so there can be confidence in the informal interpretation of the equations as rewrite rules. This chapter considers equations only as rewrite rules, and it is taken on trust that this interpretation is correct. The limitations of treating equations purely as rewrite rules are not explored here.

3.4.2 *Basic Types*

Here are some basic types to show the relevant syntax:

```
type Boolean is
  sorts Bool
  opns
    true, false :              -> Bool
    not :          Bool        -> Bool
  eqns
    ofsort Bool
      not (true) = false;
      not (false) = true;
endtype (* Boolean *)

type NaturalNumber is
  sorts Nat
  opns
    0 :                        -> Nat
    Succ :         Nat         -> Nat
endtype (* NaturalNumber *)
```

The type *Boolean* has been specified with terms of sort *Bool*. There are the usual constants *true* and *false*, and the *not* operation that complements a Boolean value. A constant in LOTOS is simply an operation with no arguments, as is the case with *true* and *false*. Since an operation always returns the same result for the same arguments, an operation with no arguments always returns the same constant value. Hopefully the equations for Booleans do not require further explanation since they are commonly used in programming.

The *NaturalNumber* type specifies whole numbers (positive or zero), which should again be familiar. The specification is based on the usual mathematical approach rather than the conventional arabic notation for numbers. The idea is to have a single constant *0*, and a *Succ* (successor) operations that yields the next value. The terms of this sort are:

0, Succ (0), Succ (Succ (0)), Succ (Succ (Succ (0))), ...

Once the above types have been specified, variables of sort *Bool* and *Nat* may be defined. Boolean expressions involving *true*, *false* and *not* may be written. There may also be natural number expressions involving *0* and *Succ*.

3.4.3 *Extension*

Types may be extended with new sorts, operations and equations. Specifying new sorts provides more sets of data values. Adding new operations expands the range of expressions that are allowed. Introducing new equations results in new properties of operations, and new ways of rewriting. Here is an extension of the natural numbers:

```
type NaturalExtended is NaturalNumber
  opns
    _ + _ : Nat, Nat -> Nat
  eqns
    forall x, y : Nat
      ofsort Nat
        x + 0 = x;
        x + Succ (y) = Succ (x + y);
endtype (* NaturalExtended *)
```

This enriches the type *NaturalNumber* with a '+' operation for addition. The operation takes two naturals and yields the natural that is their sum. The underscores before and after the operation name indicate that it is an **infix operation** (i.e. is placed between its arguments). The equations allow the new terms that may be written with '+' to be **evaluated**.

Consider now how to apply these equations to add *3* to *2*, something that would usually be written as *3 + 2 = 5* in arabic digits. The specification of '+'

works as follows, rewriting with the second equation twice and then the first equation:

```
    Succ (Succ (Succ (0))) + Succ (Succ (0))
⇒   Succ (Succ (Succ (Succ (0))) + Succ (0))
⇒   Succ (Succ (Succ (Succ (Succ (0))) + 0))
⇒   Succ (Succ (Succ (Succ (Succ (0)))))
```

3.4.4 *Combination*

Types may be combined to build more complex types. The sorts, operations and equations of each component are all included to produce the richer type. For example, a stack of natural numbers is specified with:

```
type NaturalStack is NaturalNumber, Boolean
  sorts Stack
  opns
    empty :                    −> Stack
    push :      Nat, Stack     −> Stack
    top :       Stack          −> Nat
    pop :       Stack          −> Stack
    IsEmpty :   Stack          −> Bool
  eqns
    forall n : Nat, s : Stack
      ofsort Nat
        top (push (n, s)) = n;
      ofsort Stack
        pop (push (n, s)) = s;
      ofsort Bool
        IsEmpty (empty) = true;
        IsEmpty (push (n, s)) = false;
endtype (* NaturalStack *)
```

This defines the new sort *Stack*. The empty stack is the base case for stacks. Note that this is the only operation that yields an object of sort *Stack* without using another stack as argument. The other operations (*push*, *top*, *pop* and *IsEmpty*) work on an existing value of sort *Stack*.

The equations state the expected properties: the top of a stack is the last element pushed onto it; popping a stack yields the stack prior to the last push; and finding whether a stack is empty or not requires checking whether any element has been pushed onto it.

3.4.5 *Conditional Equations*

The applicability of an equation may be made to depend on a Boolean condition — a **premiss**. If the premiss holds for a given set of values, the equation applies for these values; if the premiss does not hold for the values then the the equation does not apply for them. As an example, an operation to return the maximum of two naturals might be specified as[4]:

```
type NaturalMaximum is NaturalNumber
  opns max : Nat, Nat -> Nat
  eqns
    forall x, y : Nat
      ofsort Nat
        x ge y =>
          max (x, y) = x;
        y ge x =>
          max (x, y) = y;
endtype (* NaturalMaximum *)
```

Note that both equations apply when x and y are equal.

For every sort, there is an equality (denoted by the reserved symbol '=') that says whether two terms can be proved equal according to the equations. This equality may be used in conditional expressions. However, for technical reasons there is no direct way to check for inequality (in other words, there is no predefined $\neq$). The equations say which things are *equal*, but do not say which things are *unequal*. Apart from equational equality, specifiers may define their own Boolean equalities. These are operations that yield a Boolean value for two arguments of the same sort:

```
_ eq _ : Nat, Nat -> Bool
```

The specifier will provide enough equations to get a *true* or *false* result in all cases. LOTOS permits the use of these user-defined equalities in conditional expressions. In fact, *(x eq y)* is short for *(x eq y) = true*. A common reason for defining Boolean equality is so that inequality can be tested with expressions like *not (x eq y)*.

LOTOS does not support partial functions — those whose arguments are restricted. There are many cases where partial functions model reality accurately. For example, a specification of a stack would not expect to deal with expressions like *top (empty)* and *pop (empty)*. In the absence of appropriate equations, these are meaningless values in the sorts of natural numbers and stacks respectively. Some specifiers deal explicitly with errors like this, but

[4]The standard data type for natural numbers defines operations such as *ge* (greater than or equal) and *lt* (less than).

there are as many solutions as there are specifiers. Another approach is to 'protect' the use of the stack type in the behavioural part of the language, refusing to allow illegal operations.

3.4.6 *Renaming*

It is often the case that types are very similar. For instance, the example below specifies bit values and parity values. The only differences between these two types is that they normally use different names, and it is undesirable to mix these up.

```
type Bit is Boolean
  sorts Bit
  opns
    0, 1:                          -> Bit
    _ + _:        Bit, Bit         -> Bit
    _ eq _:       Bit, Bit         -> Bool
  eqns
    forall b : Bit
      ofsort Bit
        0 + 0 = 0; 0 + 1 = 1;
        1 + 0 = 1; 1 + 1 = 0;
      ofsort Bool
        b eq b = true;
        0 eq 1 = false; 1 eq 0 = false;
endtype (* Bit *)

type Parity is Bit renamedby
  sortnames
    Parity for Bit
  opnnames
    even for 0
    odd for 1
endtype (* Parity *)
```

The renaming facility of LOTOS allows a new, completely independent type to be specified by changing the names of an existing type. Both sort and operation names may be changed. If a name is not changed, the same name will be used in the new type. The Boolean operations in type *Parity* are the very same ones of *Boolean*. Type *Bit* provides *0*, *1*, '+' and *eq*. The *0* and *1* operations are explicitly renamed as *even* and *odd* respectively. They are therefore new operations, although similar to those in type *Bit*. Operations '+' and *eq* are not explicitly renamed, but since their arguments are renamed (i.e.

Bit is replaced by *Parity*), they become new operations with the same name. This a typical case of **operation overloading**, i.e. operations with the same name but different meanings according to their arguments and results.

Renaming is frequently used before extending a type. Suppose natural numbers have been specified, and that natural numbers modulo N are required. Rather than working on *NaturalNumber* and mixing everything up, it is better to create a fresh copy of *NaturalNumber* and extend it to deal with the new arithmetic. It is a way of protecting the existing type against unintentional changes.

3.4.7 *Parameterisation*

Reusability is a major goal of modern software engineering. In order to achieve this goal, it is necessary that software be decomposed into components that are made as reusable as possible; parameterisation is a technique that can greatly enhance this.

Types may be incompletely specified, leaving a gap for further information. A similar situation arises with functions or procedures in a programming language, where an algorithm may refer to formal arguments that are provided later. When the actual arguments are provided on a call, the algorithm is applied with these definite values. This can save repetitious programming, reducing the opportunities for making mistakes and simplifying maintenance.

Gaps in data types may be left for sorts and operations. These act as an interface to the generic type. Requirements may be imposed on this interface by specifying equations that must hold if actual parameters are to fill the gaps. Understanding this would require a detailed treatment of the semantics and so is omitted here.

The example below specifies a stack parameterised by some element sort:

```
type GenericStack is Boolean
  formalsorts Element
  sorts Stack
  opns
    empty :                      -> Stack
    push :       Element, Stack  -> Stack
    top :        Stack           -> Element
    pop :        Stack           -> Stack
    IsEmpty :    Stack           -> Bool
  eqns
    forall e : Element, s : Stack
      ofsort Element
        top (push (e, s)) = e;
```

```
        ofsort Stack
          pop (push (e, s)) = s;
        ofsort Bool
          IsEmpty (empty) = true;
          IsEmpty (push (e, s)) = false;
endtype (* GenericStack *)
```

3.4.8 *Actualisation*

Parameterised types may be actualised[5] — instantiated with actual parameters to fill the gaps. Two stacks are generated below by instantiating the parameterised stack with different actual arguments:

```
type NatStack is GenericStack actualizedby NaturalNumber using
  sortnames
    Nat for Element
    NatStack for Stack
endtype (* NatStack *)

type BoolStack is GenericStack actualizedby Boolean using
  sortnames
    Bool for Element
    BoolStack for Stack
endtype (* BoolStack *)
```

Actualisation involves a renaming of the components of the parameterised data type. Formal sorts and operations of the parameterised type are renamed as sorts and operations of the actual type. Actualisation differs from renaming in that new names are not invented; they must correspond to names in the actual argument. Usually, this association is explicit, but if both the parameterised type and the instantiating type have sorts or operations with identical names, the renaming may be implicit.

The definition of actualisation in LOTOS allows for additional renaming that is not strictly required for actualisation. This is not theoretically needed but is rather convenient. Sorts and operations that are not formal in the parameterised type may be renamed in the usual way. For instance, the first example above renames sort *GenericStack* as *NatStack* to avoid two definitions of a sort called *GenericStack*. For sorts and operations that are not formal, the rules for renaming apply.

[5] LOTOS syntax requires the spelling 'actualized'.

Beh. Exp. A	Beh. Exp. B	Condition	Interaction Type
$g\ !\ E_1$	$g\ !\ E_2$	*value* (E_1) = *value* (E_2)	value matching
$g\ !\ E_1$	$g\ ?\ x : t$	*sort* $(E_1) = t$	value passing
$g\ ?\ x : t$	$g\ ?\ y : u$	$t = u$	value generation

Figure 3.12: Interaction Types

3.5 Full LOTOS

Section 3.2 showed that Basic LOTOS allows the representation of synchronisation, but it does not allow transfer of data since the concept of data is not defined for it. Merging the process algebra defined by Basic LOTOS and the abstract data type language allows both synchronisation and transfer of data, which is necessary in distributed systems.

3.5.1 *Structured Events*

In Full LOTOS, an event has a gate identifier and a list of interaction parameters called **experiment offers**. There are two kinds of experiment offer:

- a **value offer** has the form *! v*, where *v* is a value expression
- a **variable offer** has the form *? x : s*, where *x* is a variable of sort *s*.

An event can take place at a gate only the experiment offers match in sort, value and order. Interaction possibilities are summarised in Figure 3.12. As an example, the event offer:

g ! Succ (0) ? x : Bool ! false

matches:

g ? x : Nat ! true ! false

but does not match:

g ! Succ (0) ? x : Bool

nor:

g ? x : Nat ! true ! true

Value matching represents synchronisation by matching of expected values. An example of this kind of interaction is password checking, in which an interaction only occurs if the correct password value is conveyed by a user to a resource.

Value passing represents conventional **input-output** interactions. One partner supplies a value (*! E_1* in Figure 3.12) and the other receives a value (*g ? x : t* in Figure 3.12).

Value generation represents the non-deterministic selection of a value for the interaction variable from among the valid ones. It models the concept of negotiation, since both partners in the interaction must agree on a value for the interaction to take place. After the interaction occurs, the variables of both partners have the same value ($x = y$ in Figure 3.12).

3.5.2 *Conditional Behaviour*

The behaviour of a system may depend on certain conditions. The conditions may depend on past events and/or data stored by the system. These conditions are represented in Lotos as Boolean operations involving process variables or interaction parameters. A condition may be applied before an event, or be imposed on its occurrence. The former is represented in Lotos by guards, the latter by selection predicates.

A **guard** contains a Boolean expression, called the **premiss**, and a guarded behaviour expression. If the premiss evaluates to *true*, the guarded behaviour expression is allowed to occur. An example is a system that calculates the maximum of two natural numbers as follows:

```
process MaxCalc [input, max] : exit :=
  input ? x : Nat ? y : Nat;
  (
    [x ge y] ->                    (* x greater than or equal to y? *)
      max ! x; exit                (* output x *)
  []
    [y ge x] ->                    (* y greater than or equal to x? *)
      max ! y; exit                (* output y *)
  )
endproc (* MaxCalc *)
```

In this example, *x ge y* and *x le y* are the premisses, while *max ! x;* **exit** and *max ! y;* **exit** are their respective guarded behaviour expressions. Notice that guards need not necessarily be disjoint; they are not in this example, since when x and y are equal then both guarded behaviour expressions are allowed.

The syntax of an action denotation in Full Lotos contains a gate identifier, an optional experiment offer list, and an optional **selection predicate**. The

selection predicate contains a Boolean expression, again called the **premiss**. An interaction is allowed to occur only for interaction parameters that cause the premiss to evaluate to *true*.

The following example of a public telephone allows international calls only if the user inserts more than 50 cents:

```
process Telephone [phone] : noexit :=
  phone ? dialled : TelephoneNumber ? money : Currency
    [InternationalCall (dialled) implies (money gt Cents50)];
  (Conversation [phone] (money) >> Telephone [phone])
endproc (* Telephone *)
```

3.5.3 *Parameterised Processes*

Processes can be parameterised with data parameters. Process parameters are declared in a process definition as formal parameters. In process instantiations these parameters must be assigned corresponding values, i.e. value expressions of matching sort to be evaluated.

Consider a scheduler that deals with telephone calls, supporting a first-in-first-out queue of calls requiring attention. The data type *Queue* is assumed to be specified elsewhere with operations *Append* (add to end), *Head* (first element), *Tail* (all but first) and *IsEmpty* (at least one element). The scheduler can be specified as follows:

```
process TelephoneQueue
 [phone, transfer] (waiting : Queue) : noexit :=
  phone ? num : Client;                          (* accept a call *)
  TelephoneQueue [phone, transfer] (Append (num, waiting))
                                                 (* insert call in queue *)
[]
  [not (IsEmpty (waiting))] ->                   (* check if queue empty *)
    (
      transfer ! Head (waiting);                 (* transfer first call *)
      TelephoneQueue[phone, transfer] (Tail (waiting))
                                                 (* remove first call *)
    )
endproc (* TelephoneQueue *)
```

The queue of calls is updated on each recursive instantiation of the process. If a call is received, it is inserted in the queue. If there is at least one call in the queue, the system is prepared to deal with the first one; it is then removed from the queue. Process *TelephoneQueue* will be instantiated beforehand in the specification with an initial queue value (probably the empty queue).

Operator	Conditions	Functionality
stop	-	**noexit**
exit	-	<>
exit *($v_1 \ldots v_n$)*	*sort (v_1) = s_1,...,sort (v_n) = s_n*	*<s_1,...,s_n>*
act; B	-	*func (B)*
B_1 [] B_2	*func (B_1) = func (B_2)*	*func (B_1)*
	func (B_1) = **noexit**	*func (B_2)*
	func (B_2) = **noexit**	*func (B_1)*
	otherwise	invalid
B_1 >> B_2	-	*func (B_2)*
B_1 [> B_2	same as *B_1 [] B_2*	same as *B_1 [] B_2*
B_1 parop B_2	*func (B_1) = func (B_2)*	*func (B_1)*
	func (B_1) = **noexit**	**noexit**
	func (B_2) = **noexit**	**noexit**
	otherwise	invalid

act is an action denotation, *parop* is a parallel operator.

Figure 3.13: Functionality Rules

3.5.4 *Parameterised Exit*

In Full Lotos, successful termination of behaviour expressions allows values to be conveyed with **exit**. An **exit** may therefore have a parameter list which can be filled in by value expressions. A special syntax construct **any** is introduced to indicate that any value of a sort is allowed for that parameter. Notice that the parameterless **exit** of Basic Lotos is the special case of an empty exit parameter list. Example behaviour expressions with exit lists are:

```
  a ? x : Nat ? y : Nat; exit (Add (x, y))

  a ? x : Nat; exit (x, any Nat)
|||
  b ? y : Nat; exit (any Nat, y)
```

3.5.5 *Functionality*

Each valid Lotos behaviour expression has assigned a static characteristic called its **functionality**. The functionality of a behaviour expression is an

ordered list of sorts that indicates the exit parameter list associated with this behaviour expression. The rules for the evaluation of functionality are shown in Figure 3.13. Here, **noexit** indicates that the behaviour expression does not terminate successfully at all, while $< \ldots >$ indicates a list of exit parameters.

The functionality of a process is defined as the functionality of its behaviour expression and must be explicitly declared in the process definition. The functionality of a process must match that of its defining behaviour expression.

3.5.6 *Parameterised Sequential Composition*

Section 3.2.7 explained how behaviour expressions are sequentially composed in Basic LOTOS. Full LOTOS extends this concept by also allowing values to be passed between sequentially composed behaviour expressions.

The following specification defines a system that accepts a natural number and a Boolean value as inputs, and delivers the the number or its successor depending on the Boolean input:

```
    (input ? x : Nat ? s : Bool; exit (x, s))
>>
    accept y : Nat, incr : Bool in
      (
        [incr] ->
          output ! Succ (y); exit
      []
        [not (incr)] ->
          output ! y; exit
      )
```

The parameterised sequential composition differs from the parameterless sequential composition in that it contains an **accept** statement. This statement explicitly lists the parameters that are passed to the enabled behaviour expression. The functionality of the behaviour expression preceding the enable operator must match the list of variables in the **accept**.

3.5.7 *Local Value Definition*

A **local value definition** associate values with free variables in behaviour expressions. This operator allow more conciseness and better readability in specifications, since it allows (possibly large) value expressions to be replaced by a single identifier.

Consider part of the specification of a protocol entity whose behaviour depends on the contents of a protocol data unit. The protocol data unit derives from the data parameter of a service primitive. The formal specification might read:

```
g ? sp : ServPrim;
(
  [DataField (PDU (UserData (sp))) eq <>] -> ...
[]
  [DataField (PDU (UserData (sp))) ne <>] -> ...
)
```

Using a local value definition, the equivalent specification would be:

```
g ? sp : ServPrim;
(
  let data : Data = DataField (PDU (UserData (sp))) in
    (
      [data eq <>] -> ...
    []
      [data ne <>] -> ...
    )
)
```

In this case the **let** statement associates a value with the variable *data*. This makes the specification more readable and compact, particularly if *data* is used a number of times.

3.5.8 *Generalised Choice*

There are two types of **generalised choice**: choice over gates, and choice over values. These constructs are basically shorthand notations that allow more compact specifications, and can in most cases be interpreted in terms of choice and action prefix.

Choice over gates allows a choice of identical behaviour expressions with some (formal) gates to be replaced by gates from a gate identifier list. For example:

```
choice g1 in [a1, a2, a3] []
  B [g1, h1]
```

The equivalent behaviour in terms of choice is:

```
B [a1, h1] [] B [a2, h1] [] B [a3, h1]
```

Choice over values allows a choice of identical behaviour expressions with variables instantiated with different values. An example is:

```
choice x1 : s1, x2 : s2 []
  B (x1, x2)
```

where *B (x1, x2)* is a behaviour expression. The effect of the choice over values in terms of choice may depend on the behaviour expression which follows. Consider the case of a behaviour expression consisting of an action prefix expression followed by another generic behaviour expression:

choice x1 : s1, x2 : s2 []
  g ! x1 ! x2; B (x1, x2)

This is equivalent to:

g ? x1 : s1 ? x2 : s1; B (x1, x2)

Consider now a behaviour expression in which the action prefix contains an internal event:

choice x1 : s1, x2 : s2 []
  i; g ! x1 ! x2; B (x1, x2)

In this case the choice over the values is made independently from the environment. The representation of this behaviour in terms of the action prefix and choice operators would be:

  i; g ! v1 ! w1; B (v1, w1)
[]
  i; g ! v2 ! w1; B (v2, w1)
...
[]
  i; g ! vk ! w1; B (vk, w1)
[]
  i; g ! v1 ! w2; B (v1, w2)
...
 []
  i; g ! vk ! wl; B (vk, wl)

where $v_1 \ldots v_k$ is all possible values in sort s_1 and $w_1 \ldots w_l$ is all possible values in sort s_2. If s_1 or s_2 contains infinite different values, the complete representation in terms of the choice operator becomes infinite. It is therefore not possible to say that the generalised choice is always a shorthand, since it sometimes represents behaviours that cannot be interpreted in terms of choice and the action prefix operators.

3.5.9 *Generalised Parallel Construct*

The **generalised parallel construct** is another shorthand notation that allows more compact specifications. It is therefore possible to interpret the generalised parallel operator in terms of parallel compositions of behaviour expressions. For example:

par g1 **in** [a1, a2, a3] *parop*
  B [g1, h1]

where *B [g1, h1]* is a behaviour expression and *parop* is a parallel operator such as '||' or '|||'. The corresponding behaviour expression in terms of the parallel operator looks like:

B [a1, h1] *parop* B [a2, h1] *parop* B [a3, h1]

3.5.10 *Scope*

Processes may have local definitions (after the keyword **where**), in which both data types and other processes can be defined. This determines rules for access to type and process definitions by other types and processes. A **scope** is a certain piece of LOTOS text that controls the accessibility of an element: a data type, a sort, an operation, a variable, a value or a process. LOTOS has scoping rules similar to block-structured programming languages.

Global type definitions (see Section 3.5.11) can be accessed from any part of the specification, but local definitions can be accessed only by the process in which they are defined. Furthermore, some values are accessed only by behaviour expressions.

3.5.11 *Specification*

A **specification** in LOTOS is the process that represents the whole system being specified. However, there are some syntactic differences between a specification and a process, as summarised in Figure 3.14.

A **global type definition** is a type definition that is accessible to the overall behaviour expression, type definitions and processes in the specification. Global type definitions appear before the key-word **behaviour** in a specification and do not exist in process definitions.

Just for illustration, consider the system of Figure 3.1 rewritten as a specification. A global type definition might be included as follows:

Syntax Item	Specification	Process
start of definition	**specification**	**process**
global type definitions	yes	no
start of behaviour	**behaviour**	:=
end of definition	**endspec**	**endproc**

Figure 3.14: Syntactic Differences between Specification and Process

```
specification S [a, b, c, d] : noexit

  type SomeType is                    (* a global type definition *)
    sorts SomeSort
    opns SomeOpn : SomeSort -> SomeSort
  endtype (* SomeType *)

  behaviour
    P [a, b, c] |[b]| Q [b, d]

  where

  process P[a, b, c] ...

  process Q [b, d] ...

endspec (* S *)
```

3.6 Full LOTOS Examples

The capabilities of Full LOTOS will again be illustrated with the two-key system that is further developed in Chapter 11.

3.6.1 *Event Structures*

In Full LOTOS, the definition of system interfaces has to take into account the data values exchanged at gates. The combination of gates and their value parameters is usually called the **event structure**. The first thing to consider

when describing a system with Full LOTOS is the definition of the event structure of the interfaces.

For the two-key system, the event structures are *KI ? x : KeyOps* and *Access*. Gate *KI* multiplexes all the lock-related interactions. The type (i.e. insertion or extraction) and instance (key 1 or 2) of lock operations are defined by the values of sort *KeyOps*. The elements of this sort are four constants — *In1*, *In2*, *Out1* and *Out2*. Gate *Access* is used for access to the system. It does not exchange data values.

The complete specification of the system with these event structures is:

```
specification TwoKeySystem [KI, Access] : noexit

  type KeyOps is
    sorts KeyOps
    opns In1, In2, Out1, Out2 : -> KeyOps
  endtype (* KeyOps *)

  behaviour
      KI ! In1; KI ! In2;
      Access;
      (
        KI ! Out1; KI ! Out2; stop
      []
        KI ! Out2; KI ! Out1; stop
      )
    []
      KI ! In2; KI ! In1;
      Access;
      (
        KI ! Out2; KI ! Out1; stop
      []
        KI ! Out1; KI ! Out2; stop
      )

endspec (* TwoKeySystem *)
```

3.6.2 *Selection Predicates*

The following changes to the specification illustrate the use of selection predicates. Predicates over *KeyOps* values have been defined to select the proper ordering of the lock operations. *IsKeyIn* checks for a key being inserted, while *IsKeyOut* checks for a key being extracted. *IsOtherIn* checks for the other key

of the pair being inserted, while *IsOtherOut* similarly checks for extraction. Notice that in spite of having no parameters, the system has memory: each value identifier following a '?' is like a variable that stores a value for later use.

```
library
  Boolean
endlib

type KeyOps ...

type MoreKeyOps is KeyOps, Boolean
  opns
    IsKeyIn, IsKeyOut :        KeyOps          -> Bool
    IsOtherIn, IsOtherOut :    KeyOps, KeyOps  -> Bool
  eqns
    forall x, y : KeyOps
      ofsort Bool
        IsKeyIn (In1) = true; IsKeyIn (In2) = true;
        IsKeyIn (Out1) = false; IsKeyIn (Out2) = false;
        IsKeyOut (In1) = false; IsKeyOut (In2) = false;
        IsKeyOut (Out1) = true; IsKeyOut (Out2) = true;
        not (IsKeyIn (x) and IsKeyIn (y)) =>
          IsOtherIn (x, y) = false;
        IsOtherIn (In1, In1) = false;
        IsOtherIn (In2, In2) = false;
        IsOtherIn (In1, In2) = true;
        IsOtherIn (In2, In1) = true;
        not (IsKeyOut (x) and IsKeyOut (y)) =>
          IsOtherOut (x, y) = false;
        IsOtherOut (Out1, Out1) = false;
        IsOtherOut (Out2, Out2) = false;
        IsOtherOut (Out1, Out2) = true;
        IsOtherOut (Out2, Out1) = true;
endtype (* MoreKeyOps *)

behaviour
  KI ? op1 : KeyOps [IsKeyIn (op1)];
  KI ? op2 : KeyOps [IsOtherIn (op1, op2)];
  Access;
  KI ? op1: KeyOps [IsKeyOut (op1)];
  KI ? op2: KeyOps [IsOtherOut (op1, op2)];
  stop
```

3.6.3 *Parameters and Guards*

The following changes to the specification express the same behaviour using parameterised processes and guards. Process parameters are used as state variables, while guards place conditions on state variables for transitions to occur. The model here is of an extended finite state machine.

```
library ...

type KeyOps ...

behaviour BeforeAccess [KI, Access] (false, false)

where

process BeforeAccess
 [KI, Access] (InKey1, InKey2 : Bool) : noexit :=
  (
    [not (InKey1)] ->
      KI ! In1;
      BeforeAccess [KI, Access] (true, InKey2)
  []
    [not (InKey2)] ->
      KI ! In2;
      BeforeAccess [KI, Access] (InKey1, true)
  []
    [InKey1 and InKey2] ->
      Access;
      AfterAccess [KI] (InKey1, InKey2)
  )
endproc (* BeforeAccess *)

process AfterAccess [KI] (InKey1, InKey2 : Bool) : noexit :=
  (
    [InKey1] ->
      KI ! Out1;
      AfterAccess [KI] (false, InKey2)
  []
    [InKey2] ->
      KI ! Out2;
      AfterAccess [KI] (InKey1, false)
  )
endproc (* AfterAccess *)
```