An Example of LOTOS Specification: the Matrix Switch Problem

Hubert GARAVEL Carlos RODRIGUEZ

L.G.I.
I.M.A.G. Campus
BP 53X
38041 GRENOBLE cedex
FRANCE

e-mail: {hubert, crodrig}@imag.imag.fr
phone: +(33) 76 51 48 74
telex: 980 134 F
fax: +(33) 76 51 33 79

In this paper we give a possible methodology for specifying in LOTOS. *We illustrate our approach with an example proposed at the FORTE'88 Conference: the matrix switch problem. Starting with an informal specification, we explain how to construct a corresponding* LOTOS *description.*

Introduction

The ISO specification language LOTOS [ISO88] [BB88] is expressive enough to allow a wide range of "*specification styles*" [Tur88]. As a consequence, for a given problem, many different LOTOS descriptions are possible. Faced to this situation, the ISO experts have defined guidelines for the use of LOTOS [ISO87].

From our own experience of LOTOS, we suggest a specification style for LOTOS which can be used systematically and produces easily understandable descriptions. It is illustrated by the matrix switch example, proposed at the session "FDTs on trial" of the FORTE'88 Conference, held in Stirling. By solving this problem, we describe a more general approach, which has been used to specify in LOTOS various examples, such as alternating bit protocol, sliding-window protocol, token-ring protocol, systolic computations of convolution product, and a VME bus arbiter.

This report is organized as follows. Section 1 gives the definition of the matrix switch problem exactly as it was proposed during the conference. Since this informal description is not complete, additional requirements are given in section 2. Sections 3 and 4 explain how to modelize the control part whereas section 5 does the same thing for the data part. An appendix gives the complete description in LOTOS of our solution to the matrix switch problem.

1 Informal description of the matrix switch problem

It is a real problem in telephony to define the combination of a concentrator, switch block, and expander "switch" in which many routes through the switch are simultaneously active. This also offers the problem of describing connections from many ports to many ports and of defining simultaneously active routes. The system is made from two components: a square "matrix switch" and a "concentrator".

An "$n \times n$ switch" can make connections between two sets of n ports, A and B. Any port in A can be connected to any port in B as long as ports are only connected in pairs, i.e., a port may not be connected twice.

An "m into k concentrator" has m inputs and k outputs with $m \geq k$. The concentrator connects an input to a free output when requested.

The system is built from a m into k concentrator, connected to a $k \times k$ matrix switch, connected to a n into k concentrator.

Requests for connections arrive concurrently from all $m + n$ users.

2 Additional requirements

Since the definition given in section 1 is intentionally incomplete and ambiguous, we deliberately added the following constraints:

- ports in A are numbered 1, ... m. Users connected to these ports are called "port A users" and are also numbered 1, ... m
- ports in B are numbered 1, ... n. Users connected to these ports are called "port B users" and are also numbered 1, ... n
- any user can request either for connection or disconnection
- the data transmission phase — which would take place between connection and disconnection — will not be described
- the only purpose of the system is to create a route between port A users and port B users. The users can not specify the number of the user they want to communicate with — such information may appear as data transmission for further processing
- the system is fully symmetric with respect to A and B, save from the fact that A and B don't have the same number of ports

3 Control part: parallel architecture

The first design step is the definition of the overall architecture. The following methodology must be recursively applied to the sub-systems of the system to describe, in order to follow a top-down decomposition strategy.

- *divide the system into components which are executed in parallel. Give names to these concurrent entities by defining one* LOTOS *process for each.*

  For the matrix switch the decomposition is obvious: the system is made from 3 concurrent processes: 2 concentrators and 1 matrix switch. As a design choice we define these 3 sub-systems as different instances of the same process definition, the "generic switch", which is parametrized by the number of input and output users. For the matrix switch these parameters are (k, k). For the concentrators they are respectively (m, k) and (k, n).

- *identify interaction points of each process. Modelize them by formal gate parameters.*

  The generic (p, q) switch interacts with two groups of users: p users on port A and q users on port B. Since p and q are parameters, it would not be possible for the generic switch to have as many gates as users. For this reason we use a LOTOS trick for creating gate arrays: a single gate G can be expanded into p actually different gates by using a discriminating offer $G\texttt{!}1, ...G\texttt{!}p$. Here the two ports of the generic switch are represented by two gates, A and B which are respectively shared by p and q users. Consequently the generic switch has the following header:

  ```
  process GENERIC_SWITCH [A, B] (P, Q:NAT, ...) : noexit :=
     ...
  endproc
  ```

- *identify communications between these processes. Modelize synchronous communications by* LOTOS *rendez-vous and asynchronous communications by defining buffering* LOTOS *processes. Compose all processes with parallel operators as to synchronize the interaction points to be connected.*

  For the matrix switch example we use two internal gates, A_0 which links the (m, k) switch to the (k, k) switch and B_0 which links the (k, k) switch to the (k, n) switch:

  ```
  GENERIC_SWITCH [A, A0] (M, K, ...)
  |[A0]|
  GENERIC_SWITCH [A0, B0] (K, K, ...)
  |[B0]|
  GENERIC_SWITCH [B0, B] (K, N, ...)
  ```

  Note that the "users" of a switch can be either switches or end-users.

- *put suitable abstraction to mask the internal communications of the system, which are irrelevant with respect to its specifications. Use the hiding operator of* LOTOS *to hide internal gates.*

  For the matrix switch example we have to mask gates A_0 and B_0:

```
hide A0, B0 in
   (
   GENERIC_SWITCH [A, A0] (M, K, ...)
   |[A0]|
   GENERIC_SWITCH [A0, B0] (K, K, ...)
   |[B0]|
   GENERIC_SWITCH [B0, B] (K, N, ...)
   )
```

4 Control part: sequential components

The second step consists in describing the behaviour of each sequential component by a finite state machine, and then expressing this machine in terms of LOTOS behaviour expressions.

Before defining the behaviour of the generic switch, we have to agree upon the format of messages. We decide that only two kinds of messages are used:

- A `!CONNECT !`x indicates that a connection request is sent by/to the user $\#x$ attached to port A
- A `!DISCONNECT !`x indicates that a disconnection request is sent by/to the user $\#x$ attached to port A

These rules also apply to port B. Here `CONNECT` and `DISCONNECT` are two values of an sort named `REQUEST`.

The behaviour of the switch can be described as follows:

- when a connection request A `!CONNECT !`x is received, a connection indication B `!CONNECT !`y is sent, provided that an empty channel y exists on port B. The switch also remembers that a $(x \leftrightarrow y)$ connection pair has been set
- when a disconnection request A `!DISCONNECT !`x is received, a disconnection indication B `!DISCONNECT !`y is sent, where y is such that a connection pair $(x \leftrightarrow y)$ was previously set. The switch also forgets the connection pair $(x \leftrightarrow y)$

Requests arriving on port B are symmetrically handled.

So it is necessary for the switch to keep trace of existing connection pairs. The easiest solution is to record these informations into a *state variable* R of sort `ROUTE`, which is internal to the switch. The following data operations on sort `ROUTE` are used:

- `EMPTY`: returns the empty route. The state variable of the switch is initialized to `EMPTY`
- `CREATE` (R, x, y): returns route R augmented by connection pair $(x \leftrightarrow y)$
- `DELETE` (R, x, y): returns route R deprived from connection pair $(x \leftrightarrow y)$
- `USED` (R): returns the number of connection pairs in route R
- `PAIR_B` (R, x): returns y such that connection pair $(x \leftrightarrow y)$ belongs to route R

- FREE_B (R): returns some y such that no connection pair $(x \leftrightarrow y)$ belongs to route R

The translation from such an automaton to an equivalent LOTOS behaviour is straightforward:

- *identify "loop states", i.e., states which can be reached from themselves. For each loop state, create a corresponding* LOTOS *process, called here a "state process".*

  In the switch example, there is only one loop state, the initial state. In fact it is possible to create as many state processes as states in the automaton; this should be avoided since it leads to unstructured specifications (the "goto" style)

- *sequential composition is expressed by the "*`;`*" operator*

- *alternative composition is expressed by the "*`[]`*" operator*

- *for each state variable, add a corresponding value parameter to every state processes.*

  Here we add a third value parameter, R of sort ROUTE, to process GENERIC_SWITCH

Here is the partial specification of process GENERIC_SORT; requests received on port B should be handled symmetrically:

```
process GENERIC_SWITCH [A, B] (P, Q:NAT, R:ROUTE) : noexit :=
   (* connection request from port #X on gate A *)
   [USED (R) lt Q] ->
      A !CONNECT ?X:NAT;
         (
         let Y:NAT = FREE_B (R) in
            B !CONNECT !Y;
               GENERIC_SWITCH [A, B] (P, Q, CREATE (R, X, Y))
         )
   []
   (* disconnection request from user #X on gate A *)
   A !DISCONNECT ?X:NAT;
      (
      let Y:NAT = PAIR_B (R, X) in
         B !DISCONNECT !Y;
            GENERIC_SWITCH [A, B] (P, Q, DELETE (R, X, Y))
      )
   []
   ...
endproc
```

Note that our approach is compatible with the constraint-oriented style. In the matrix switch example, one can see that the generic switch trusts the users: for instance an end-user is allowed to send two connection requests with no disconnection request between. Safety properties could be enforced by synchronizing the matrix switch with a constraint process which would prevent users from doing so.

5 Data part

The last step in the specification is the description of data types. Experience shows that sorts and operations should only be specified when control part is completely determined and stable.

- *list all sorts and operations used in behaviour descriptions. Group them in types to obtain a hierarchical and modular structure. For each type T, fill the "***sorts***" and "***opns***" parts.*

  For the matrix switch, one obtains the following LOTOS type definitions:

  ```
  type REQUEST is
     sorts REQUEST
     opns CONNECT : -> REQUEST
          DISCONNECT : -> REQUEST
  endtype

  type ROUTE is NATURALNUMBER
     sorts ROUTE
     opns EMPTY : -> ROUTE
          CREATE : ROUTE, NAT, NAT -> ROUTE
          DELETE : ROUTE, NAT, NAT -> ROUTE
          USED : ROUTE -> NAT
          PAIR_B : ROUTE, NAT -> NAT
          FREE_B : ROUTE -> NAT
          ...
     eqns
          ...
  entype
  ```

  For type `REQUEST` there are no equations. For type `ROUTE`, operators associated with requests received on port B are not defined here

- *for each type T, fill the "***eqns***" according to the following methodology. For each sort S of T, divide operations which return results of sort S in two classes:*

  - *constructors: they are primitive operations which can not be eliminated; any value of sort S has a normal form term with only constructor operations. The meaning of constructors operations needs not to be defined by equations.*

    For the switch, the constructors are `EMPTY` and `CREATE`. Any set of connection pairs can be expressed only in terms of `EMPTY` and `CREATE`.

  - *non-constructors: they are non-primitive operations, i.e., any term containing non-constructors can be replaced by an equivalent term containing only constructors. Non-constructors have to be defined by induction on the structure of their arguments.*

    For the switch, `DELETE` is a non-constructor operation which is defined as follows:

    ```
    forall R, R0:ROUTE, X, X0, Y, Y0:NAT
      (X eq X0) and (Y eq Y0) =>
    ```

```
      DELETE (CREATE (R0, X0, Y0), X, Y) = R0;
   (X ne X0) and (Y ne Y0) =>
      DELETE (CREATE (R0, X0, Y0), X, Y) =
         CREATE (DELETE (R0, X, Y), X0, Y0);
```

- *for each type T, for each sort S, operations which do not return results of sort S are defined by induction on the structure of their arguments.*

  For instance, the `USED` and `PAIR_B` operations can be defined as follows:

  ```
  forall R, R0:ROUTE, X, X0, Y0:NAT
    USED (EMPTY) = 0;
    USED (CREATE (R0, X0, Y0)) = SUCC (USED (R0));
    X eq X0 =>
      PAIR_B (CREATE (R0, X0, Y0), X) = Y0;
    X ne X0 =>
      PAIR_B (CREATE (R0, X0, Y0), X) = PAIR_B (R0, X);
  ```

By following this methodology based on case analysis (sometimes called "*constructor discipline*"), the specifier can convince himself that no equation is missing and that no equation is redundant (which would otherwise raise the confluency problem).

Moreover it is possible to derive concrete implementations from abstract data type specifications written according to constructor discipline. For instance the CÆSAR.ADT tool [Bar88] [Gar89b] automatically translates such LOTOS sorts and operations into corresponding C types and functions. The translation is fast and the generated code efficient since run-time execution is fully deterministic.

On the opposite hand, this constructive approach may very well lead to over-specification, i.e., loss of generality. We illustrate this point with the `FREE_B` operation; it can be given the following definition:

```
forall R:ROUTE
   FREE_B (R) = FREE_B (R, R, 0);
```

where `FREE_B : ROUTE, ROUTE, NAT -> NAT` is an auxiliary operation defined by:

```
forall R, R0:ROUTE, X, X0, Y0:NAT
   FREE_B (R, EMPTY, X) = X;
   X eq X0 =>
     FREE_B (R, CREATE (R0, X0, Y0), X) = FREE_B (R, R, SUCC (X));
   X ne X0 =>
     FREE_B (R, CREATE (R0, X0, Y0), X) = FREE_B (R, R0, X);
```

In other terms, by providing a constructive definition of `FREE_B`, we lose non-determinism: `FREE_B` R always returns the smallest y such that no connection pair $(x \leftrightarrow y)$ exists in R. A general solution would be to change the definition of process `GENERIC_SWITCH` and replace:

```
let Y:NAT = FREE_B (R) in
   B !CONNECT !Y;
```

by:

```
B !CONNECT ?Y:NAT [FREE_B (R, Y)]
```

where `FREE_B` (R, y) returns true if and if only no connection pair $(x \leftrightarrow y)$ exists in R.

Conclusion

In this paper, we propose a "reasonable" specification style in LOTOS, based on communicating automata for the control part and constructor discipline for the data part. In this approach we only use static control structures and we make plain use of data structures to express dynamism.

A good specification should avoid over-specification, in order not to prevent implementation on various kinds of machines, architectures and environments. On the other hand, it would be a mess to design a specification which is too far from any realistic implementation; otherwise any implementor would have to translate it into a lower level specification: these redundant and error-prone efforts should be avoided.

A good specification should be automatically verified since its complexity is likely to exceed human capabilities. See for instance [PG88] which shows, by exhibiting several incompatibilities between standardized transport protocol T.70 and transport services X.213 and X.214, that even best international experts can very well make mistakes. The specification style proposed here allows the use of tools, such as CÆSAR [Gar89a] [GS90] to compile and verify LOTOS specifications.

Dynamic control features (i.e., for LOTOS, recursion in parallel composition as well as recursion on the left of the enabling and disabling operators) should be avoided, since they are often difficult to understand, to implement efficiently and to verify automatically. Note that the debate on dynamic control is not new and also arose for sequential languages: self-modifying (non-reentrant) programs, dynamic procedure calls, ...

Acknowledgements

The authors are grateful to Susanne Graf, Laurent Mounier and Jacques Voiron for their helpful comments and suggestions.

Appendix: Full specification of the matrix switch

```
specification SWITCH [A, B] (M, K, N:NAT) : noexit

library BOOLEAN endlib

library NATURALNUMBER endlib

type REQUEST is
   sorts REQUEST
   opns CONNECT (*! constructor *) : -> REQUEST
        DISCONNECT (*! constructor *) : -> REQUEST
endtype
```

```
type ROUTE is NATURALNUMBER
   sorts ROUTE
   opns EMPTY (*! constructor *) : -> ROUTE
        CREATE (*! constructor *) : ROUTE, NAT, NAT -> ROUTE
        DELETE : ROUTE, NAT, NAT -> ROUTE
        USED : ROUTE -> NAT
        PAIR_B : ROUTE, NAT -> NAT
        PAIR_A : ROUTE, NAT -> NAT
        FREE_B : ROUTE -> NAT
        FREE_B : ROUTE, ROUTE, NAT -> NAT
        FREE_A : ROUTE -> NAT
        FREE_A : ROUTE, ROUTE, NAT -> NAT
   eqns
   forall R, R0 : ROUTE,
          X, X0, Y, Y0 : NAT
   ofsort ROUTE
     (X eq X0) and (Y eq Y0) =>
       DELETE (CREATE (R0, X0, Y0), X, Y) = R0;
     (X ne X0) and (Y ne Y0) =>
       DELETE (CREATE (R0, X0, Y0), X, Y) =
         CREATE (DELETE (R0, X, Y), X0, Y0);
   ofsort NAT
     USED (EMPTY) = 0;
     USED (CREATE (R0, X0, Y0)) = SUCC (USED (R0));
   ofsort NAT
     X eq X0 =>
       PAIR_B (CREATE (R0, X0, Y0), X) = Y0;
     X ne X0 =>
       PAIR_B (CREATE (R0, X0, Y0), X) = PAIR_B (R0, X);
   ofsort NAT
     Y eq Y0 =>
       PAIR_A (CREATE (R0, X0, Y0), Y) = X0;
     Y ne Y0 =>
       PAIR_A (CREATE (R0, X0, Y0), Y) = PAIR_A (R0, Y);
   ofsort NAT
     FREE_B (R) = FREE_B (R, R, 0);
   ofsort NAT
     FREE_B (R, EMPTY, X) = X;
     X eq X0 =>
       FREE_B (R, CREATE (R0, X0, Y0), X) =
         FREE_B (R, R, SUCC (X));
     X ne X0 =>
       FREE_B (R, CREATE (R0, X0, Y0), X) =
         FREE_B (R, R0, X);
   ofsort NAT
     FREE_A (R) = FREE_A (R, R, 0);
   ofsort NAT
     FREE_A (R, EMPTY, Y) = Y;
     Y eq Y0 =>
```

```
      FREE_A (R, CREATE (R0, X0, Y0), Y) =
        FREE_A (R, R, SUCC (Y));
    Y ne Y0 =>
      FREE_A (R, CREATE (R0, X0, Y0), Y) =
        FREE_A (R, R0, Y);
endtype

behaviour

   hide A0, B0 in
      (
      GENERIC_SWITCH [A, A0] (M, K, EMPTY)
      |[A0]|
      GENERIC_SWITCH [A0, B0] (K, K, EMPTY)
      |[B0]|
      GENERIC_SWITCH [B0, B] (K, N, EMPTY)
      )

where

process GENERIC_SWITCH [A, B] (P, Q:NAT, R:ROUTE) : noexit :=
   (* connection request from port #X on gate A *)
   [USED (R) lt Q] ->
      A !CONNECT ?X:NAT;
         (
         let Y:NAT = FREE_B (R) in
            B !CONNECT !Y;
               GENERIC_SWITCH [A, B] (P, Q, CREATE (R, X, Y))
         )
   []
   (* connection request from user #Y on gate B *)
   [USED (R) lt P] ->
      B !CONNECT ?Y:NAT;
         (
         let X:NAT = FREE_A (R) in
            A !CONNECT !X;
               GENERIC_SWITCH [A, B] (P, Q, CREATE (R, X, Y))
         )
   []
   (* disconnection request from user #X on gate A *)
   A !DISCONNECT ?X:NAT;
      (
      let Y:NAT = PAIR_B (R, X) in
         B !DISCONNECT !Y;
            GENERIC_SWITCH [A, B] (P, Q, DELETE (R, X, Y))
      )
   []
   (* disconnection request from user #Y on gate B *)
   B !DISCONNECT ?Y:NAT;
```

```
      (
      let X:NAT = PAIR_A (R, Y) in
         A !DISCONNECT !X;
            GENERIC_SWITCH [A, B] (P, Q, DELETE (R, X, Y))
      )
endproc

endspec
```

References

[Bar88] Christian Bard. *CÆSAR.ADT Reference Manual*. Laboratoire de Génie Informatique — Institut IMAG, Grenoble, August 1988.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

[Gar89a] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar89b] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162, Amsterdam, December 1989. North-Holland.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394, Amsterdam, June 1990. IFIP, North-Holland.

[ISO87] ISO. Guidelines for the Application of Estelle, LOTOS and SDL. PDRT 10167, International Organization for Standardization, Genève, December 1987.

[ISO88] ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[PG88] Marc Phalippou and Roland Groz. Using ESTELLE for Verification: An Experience with the T.70 Teletex Transport Protocol. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 185–199, Amsterdam, September 1988. North-Holland.

[Tur88] Kenneth J. Turner. A LOTOS-Based Development Strategy. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 157–174, Amsterdam, September 1988. North-Holland.