ソフトウェア工学 64－12
（1 9 8 9. 2. 2）

図書館の問題とエレベータの問題のＬＯＴＯＳによる仕様記述

大 蒔 和 仁、 二 木 厚 吉
電子技術総合研究所

LOTOSはISOにおいてプロトコル記述用に設計された仕様記述言語である。現在、幾つかのプロトコルに対して記述例が示されている。我々は、プロトコル以外の一般的な問題として図書館の問題とエレベータの問題とを選び、それらをLOTOSにより仕様記述を行なってみた。

その動機は次の二つである。
(1)一般に問題は静的な部分と動的な部分に分かれている。LOTOSは抽象データ型記述と並行プロセス記述の部分の枠組を用意しているので、問題の仕様記述に向いていると思われる。
(2)LOTOSは現在OSIの下位層に関する記述が行なわれている。今後応用層の記述へと進んでいくと思われる。その際にも通信に関わること以外のより一般的な問題についてもLOTOSの仕様記述言語としての性質を調べておくことが必要である。

この論文ではLOTOSによる図書館の問題とエレベータの問題とに記述を示し、LOTOSの利点と欠点とを述べる。さらにLOTOSのプログラミング環境についても考察する。

Specifications of Library and Lift Problems in LOTOS

Kazuhito OHMAKI and Kokichi FUTATSUGI
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, Ibaraki 305, JAPAN

LOTOS was designed to specify protocols. We have applied LOTOS to specify more general problems, Library and Lift problems, other than protocols.

Our motivations are :
(1)Generally, any problem has two features of static and dynamic behaviors. LOTOS provides the frameworks for both abstract data types and process communications. We think that LOTOS is suitable to specify by using these two frameworks to spececify static part and dynamic part of the given problem.
(2)There are already lower layer protocols written in LOTOS. The application layers will be written in LOTOS in the future. To make a feasibility study in the application area, we should try to specify more general problems in LOTOS other than "standard" protocols.

In this paper, we show the specification of Library and Lift problems in LOTOS, and dicuss the feasibilities of LOTOS from the viewpoint of a general specification language. We also observe the requirements of LOTOS programming environment.

1. Introduction

LOTOS (Language of Temporal Ordering Specification) has been designed to specify OSI protocols by FDT (Formal Description Technique) experts[3]. LOTOS has two features of abstract data type definitions and process definitions. The data type definitions are based on algebraic specification of data types. The process behavior definitions are bases on CCS (Calculus of Communicating Systems)[7]. There are several protocol specifications written in LOTOS[4,5,6].

We have applied LOTOS to specify more general problems, i.e. Library and Lift problems[1]. Our motivations are :
(1)Generally, any problem has two features of static and dynamic behaviors. LOTOS provides the frameworks for both abstract data types and process communications. We think that LOTOS is suitable to specify by using these two frameworks to spececify static part and dynamic part of the given problem.
(2)There are already lower layer protocols written in LOTOS. The application layers will be written in LOTOS in the future. To make a feasibility study in the application area, we should try to specify more general problems in LOTOS rather than "standard" protocols.

Our experiment show us advantages and disadvantages of LOTOS as a specification language.

In the specification of the Library problem, we will conclude that this type of problems is composed mainly of data type definitions. And most operators are defined in the data type definitions. In the process definitions, we only define two concurrent processes where staffs of the library and borrowers are acting independently. In the case of the Lift problem, most part of the specification consist of process definitions. In this specification, we used the same logical structure as in [2]. LOTOS can explicitly specify the communication between different processes, but CSP can not. Moreover, data types are exactly specified within the LOTOS specification. For example, a queue structure is employed as a process in [2], but in LOTOS we can specify it as a data type.

We will also consider the requirements for programming environment for LOTOS.

From the point of the programming environment of LOTOS, there are few LOTOS processors other than [8]. A sophisticated programming environment will be needed to browse type definitions and process definitions. Without one, we could scarcely write down any LOTOS programs.

We assume that the readers are familiar with LOTOS in the paper. Readers can refer the exact definitions of LOTOS in the documents [3].

2. Specification of the Libray problem

2.1 Definition of Library Problem

The library problem is defined in [1]. Here is its informal specification.

A library data base accepts the following transactions:
1. Check out a copy of a book/ Return a copy of a book;
2. Add a copy of a book to/ Remove a copy of a book from the library;
3. Get the list of books by a particular author or in a particular subject area;
4. Find out the list of books currently checked out by a particular borrower;
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users of the data base: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff user, except that ordinary borrowers can perform transactions 4 to find out the list of books currently borrowed by themselves.

2.2 Specification steps

We have developed the LOTOS documents according to the following steps.

(1) Interface gates

Whatever specification language we use, we have to decide the interface between the observer (outer environment) and the specified system. In the case of LOTOS, this interface should be specified in terms of gates.

There are two types of users of the library system; staffs and borrowers. Therefore, we have decided to use two different gates for the communication between the outer environment and the library system. We call them `StaffGate` and `BorrowerGate`.

```
type QueryByStaffOnlyType is Boolean
  sort QueryByStaffOnly
  opns doBorrow                 : -> QueryByStaffOnly
         doReturn               : -> QueryByStaffOnly
         doAdd                  : -> QueryByStaffOnly
         doDelete               : -> QueryByStaffOnly
         doShowBorrowedBooks    : -> QueryByStaffOnly
         doShowLastBorrower     : -> QueryByStaffOnly
         _eq_ : QueryStaffOnly, QueryStaffOnly -> Bool
  eqns ofsort Bool
    doBorrow eq doBorrow = true;
    doBorrow eq doReturn = false;
    ...
endtype

type QueryByBothStaffAndBorrowerType is Boolean
  sort QueryByBothStaffAndBorrower
  opns doShowListByAuthor  : -> QueryByBothStaffAndBorrower
         doShowListBySubject : -> QueryByBothStaffAndBorrower
         _eq_ : QueryByBothStaffAndBorrower,
                QueryByBothStaffAndBorrower -> Bool
  eqns ofsort Bool
    doShowListByAuthor  eq doShowListByAuthor  = true;
    doShowListByAuthor  eq doShowListBySubject = false;
    doShowListBySubject eq doShowListBySubject = true;
    doShowListBySubject eq doShowListByAuthor  = false;
endtype
```

Fig.1 Data types for query on a data base.

queries	attributes
doBorrow	StaffGate?query:QueryByStaffOnly?param1:BorrowerName?param2:Book
doReturn	StaffGate?query:QueryByStaffOnly?param1:BorrowerName?param2:Book
doAdd	StaffGate?query:QueryByStaffOnly?param:Book
doDelete	StaffGate?query:QueryByStaffOnly?param:Book
doShowBorrowedBooks	StaffGate?query:QueryByStaffOnly?param:BorrowerName!bookList(param,BorrowedQueue)
doShowLastBorrower	StaffGate?query:QueryByStaffOnly?param:Book!borrowerName(param,BorrowedQueue)
doShowListByAuthor	StaffGate?query:QueryByBothStaffAndBorrower?param:AuthorName!searchByAuthor(param,CollectedQueue)
doShowListBySubject	StaffGate?query:QueryByBothStaffAndBorrower?param:SubjectName!searchBySubject(param,CollectedQueue)
doShowListByAuthor	BorrowerGate?query:QueryByBothStaffAndBorrower?param:AuthorName!searchByAuthor(param,CollectedQueue)
doShowListBySubject	BorrowerGate?query:QueryByBothStaffAndBorrower?param:SubjectName!searchBySubject(param,CollectedQueue)

Table1 Queries and gate attributes

(2) Define query types

In Fig.1, we define two types for queries to the Library system;
`QueryByStaffOnlyType` and `QueryByBothStaffAndBorrowerOnlyType`.
These types define constants (nullary operators) for queries. And the former type is for queries only allowed to staffs, and latter is for both staffs and borrowers.

(3) Define gate attributes

Then we attach several attributes on two gates. These attributes are corresponding to inquiries and responses on the data base. The Table 1 shows the queries and the corresponding attributes. For example, the attributed event

```
StaffGate?query:QueryByStaffOnly
          ?param1:BorrowerName
          ?param2:Book
```

is only valid when the variable `query` is `doBorrow`. This validity is checked using an event with a selection predicate, that is,

```
StaffGate?query:QueryByStaffOnly
          ?param1:BorrowerName
          ?param2:Book
          [query eq doBorrow]
```

There are several other data types and process parameters in Table 1. We will explain them soon.

(4) Define data types

To complete the specification, we have to declare all necessary data types appeared in Table 1.

The sorts `BorrowerName` and `Book` are shown in Fig.2(a) and (b). These sorts represent the names of borrowers and books.

`CollectedQueue` and `BorrowedQueue` in Table 1 are process parameters of the main process of the Library system. The data types of these parameters are
`CollectedBookQueueType` and
`BorrowedBookQueueType`
which are shown in Fig.3. These are renamed types derived from
`QueueOfCollectedBookType` and
`QueueOfBorrowedBookType`
of Fig.4. Moreover, we are using the formal list type to keep books in the library and borrowed books as shown in Fig.5. All operators are renamed to be appropriate names for the Library system.

`CollectedBookQueueType` is for the data base for collected books in the library. `BorrowedBookQueueType` is the data base for borrowed books. In Table 1, the parameters `CollectedQueue` of `CollectedBookQueueType` and `BorrowedQueue` of `BorrowedBookQueueType` are initialized to be empty when the process is invoked. For the parameter `CollectedQueue`, we can perform the operators `add`, `delete`, `searchByAuthor`, and `searchBySubject`. For `BorrowedQueue`, we can use the operators `borrow`, `return`, `bookList`, and `borrowerName`. These operators are corresponding to the transactions in 2.1.

(5) Main process of the Library system

We show the main process of the Library system in Fig.6. The expressions which we will not elaborate on are commented out for brevity.

```
type BookType is BookNameType, AuthorNameType, SubjectNameType
  sorts Book
  opns triple    : BookName, AuthorName, SubjectName -> Book
       nameIs    : Book -> BookName
       authorIs  : Book -> AuthorName
       subjectIs : Book -> SubjectName
  eqns forall bn:BookName, an:AuthorName, sn:SubjectName
   ofsorts BookName    nameIs(triple(bn, an, sn)) = bn;
   ofsorts AuthorName  authorIs(triple(bn, an, sn)) = an;
   ofsorts SubjectName subjectIs(triple(bn, an, sn)) = sn;
endtype
```

(a) Definition of BookType

```
type BorrowerBookPairType is BorrowerNameType, BookNameType
  sorts BorrowerBookPair
  opns tuple      : BorrowerName, Book -> BorrowerBookPair
       borrowerIs : BorrowerBookPair   -> BorrowerName
       bookIs     : BorrowerBookPair   -> Book
  eqns forall p: BorrowerBookPair, borrower:BorrowerName, bk:Book
    ofsort BorrowerName borrowerIs(tuple(borrower, bk)) = borrower;
    ofsort Book         bookIs(tuple(borrower, bk)) = bk;
endtype
```

(b) Definition of BorrowerBookPairType

Fig.2 Data types related to Books

```
type CollectedBookQueueType is QueueOfCollectedBookType, BorrowedBookListType
  renamedby sortnames CollectedBookQueue for Queue
  opns add             : Book, CollectedBookQueue -> CollectedBookQueue
       delete          : Book, CollectedBookQueue -> CollectedBookQueue
       searchByAuthor  : AuthorName, CollectedBookQueue  -> BookList
       searchBySubject : SubjectName, CollectedBookQueue -> BookList
  eqns forall b:Book,  cbq:CollectedBookQueue,
              bn:BookName, an:AuthorName, sn:SubjectName
  ofsorts CollectedBookQueue
    add(b, cbq) = b +-- cbq;
    delete(b, cbq) = dequeueIf(cbq, b);
  ofsorts BookList
    isInQueue(cbq, triple(bn, an, sn)) =>
     searchByAuthor(an, cbq) =
      triple(bn, an, sn).searchByAuthor(an, dequeueIf(cbq, triple(bn,an,sn)));
    not(isInQueue(cbq, triple(bn, an, sn)) => searchByAuthor(an, cbq) = nil;
    isInQueue(cbq, triple(bn, an, sn)) =>
     searchBySubject(sn, cbq) =
      triple(bn, an, sn).searchBySubject(sn, dequeueIf(cbq, triple(bn,an,sn)));
    not(isInQueue(cbq, triple(bn, an, sn)) => searchBySubject(sn, cbq) = nil;
endtype

type BorrowedBookQueueType is QueueOfBorrowedBookType, BorrowedBookListType
  renamedby sortnames BorrowedBookQueue for Queue
  opns borrow     : BorrowerName, Book, BorrowedBookQueue -> BorrowedBookQueue
       return     : BorrowerName, Book, BorrowedBookQueue -> BorrowedBookQueue
       bookList     : BorrowerName, BorrowedBookQueue     -> BorrowedBookList
       borrowerName : Book, BorrowedBookQueue             -> BorrowerName
  eqns forall person:BorrowerName, book:Book, bbq:BorrowedBookQueue
  ofsort BorrowedBookQueue
    borrow(person, book, bbq) = tuple(person, book) +-- bbq;
    return(person, book, bbq) = dequeueIf(bbq, tuple(person, book));
  ofsort BorrowedBookList
   isInQueue(bbq, tuple(person, book)) =>
    bookList(person, bbq) =
      book.bookList(person, dequeueIf(bbq, tuple(person, book)));
  not(isInQueue(bbq, tuple(person, book)) => bookList(person, bbq) = nil;
  ofsort BorrowerName
  isInQueue(bbq, tuple(person, book)) => borrowerName(book, bbq) = person;
endtype
```

Fig.3 Definitions of CollectedBookQueueType and BorrowedBookQueueType

```
type FormalQueueType is Boolean, Element
  sorts Queue
  opns empty     :                     -> Queue
       _+--_   : Element, Queue  -> Queue
       _--+_     : Queue, Element  -> Queue
       _eq_      : Queue, Queue    -> Bool
       dequeueIf : Queue, Element  -> Queue
       isEmpty   : Queue           -> Bool
       isInQueue : Queue, Element  -> Bool
  eqns forall q, q1, q2 : Queue, e, e1, e2 : Element
    ofsort Queue
      e +-- empty = empty --+ e;
      e1 +-- (q --+ e2) = (e1 +-- q) --+ e2;
      isInQueue(q --+ e1, e) and (e1 eq e) =>
        dequeueIf(q --+  e1,e) = q;
      isInQueue(q --+ e1, e) and (e1 ne e) =>
        dequeueIf(q --+ e1,e) =dequeueIf(q,e);
      not (isInQueue(q, e)) => dequeueIf(q, e) = q;
    ofsort Bool
      isEmpty(empty) = true;
      isEmpty(e +-- q) = false;
      empty eq empty = true;
      (e +-- q) eq empty = false;
      empty eq (e +-- q) = false;
      (e1 +-- q1) eq (e2 +-- q2) = (e1 eq e2) and (q1 eq q2);
      isInQueue(empty, e) = false;
      isInQueue(q --+ e1, e) = (e1 eq e) or isInQueue(q, e);
endtype

type QueueOfCollectedBooksType is FormalQueueType
  actualizedby BookType using sortnames Book for Element
endtype

type QueueOfBorrowedBookType is FormalQueueType
  actualizedby BorrowerBookPairType using
  sortnames BorrwerBookPair for Element
endtype
```

Fig.4 Queues for collected books and borrowed books

This process is composed from two behavior expressions. The first one is for `StaffGate` and the other one is for `BorrowerGate`. These two expressions are composed by a choice operator []. Each behavior expression is composed by choice operator [] with guarded expressions.

For example, the part (A) and (B) in Fig.6 have the following meanings:
If the query `doBorrow` command is entered at the gate `StaffGate` with attributes `param1` and `param2` whose values are `BorrowerName` and `Book`, then the operation `borrow` will be performed on `BorrowedBookQueue`.

In the case of (C), when `doShowBorrowedBooks` is entered at the gate `StaffGate` with `param` of `BorrowerName`, then the third argument `bookList(param, BorrowedQueue)` is returned to the outer environment.

The part (D) in Fig.6 shows `doShowListByAuthor` on the gate `BorrowerGate`. This returns the value of `searchByAuthor` operation on `CollectedQueue`.

We omit the explanations on other parts of behavior expressions in Fig.6.

(6) Whole specification of the Library system

Fig.7 shows the whole structure of the specification `LibrarySystem`. The parameter `MaxBorrowable` is of sort `Nat` and is used to check whether a borrower wishes to borrow the number of books exceeding this number or not. This is used in the part (A) of Fig.6. The part (A) of Fig.7 gives three initial values for three parameters for the process `MainProc`. This part initializes the parameters `CollectedQueue` and `BorrowedQueue` to be empty.

3. Specification of Lift Problem

3.1 Definition of Lift problem

An n lift system is to be installed in a building with m floors. The problem concerns the logic to move lifts between floors according to the following constraints:

1.Each lift has a set of buttons, one for each floor. These illuminate when pressed and cause the lift to visit the corresponding floors.
2.Each floor has two buttons (except ground and top floor), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed. The illumination is cancelled when a lift visits the floor and is either moving in the desired direction, or has no outstanding requests. In the latter case, if both floor request buttons are pressed, only one should be cancelled. The algorithm to decide which to service first should minimize the waiting time for both requests.
3.When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests.
4.All requests for lifts from floors must be serviced eventually, with floors given equal priority.
5.All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel.
6.Each lift has an emergency button which, when pressed, causes a warning signal to be sent to the site manager. The lift is then deemed 'out of service'. Each lift has a mechanism to cancel its 'out of service' status.

3.2 Specification

Instead of describing the development steps precisely as in Library case, we show here the final result and its structure. The basic structure of our specification is basically the same as that of the CSP trial[2].

```
type FormalListType is Element, NaturalNumber
  sort List
  opns nil : -> List
       _._ : Element, List -> List
       length : List -> Nat
  eqns forall e:Element, l:List
  ofsort Nat
    length(nil) = 0;
    length(e.l) = Succ(length(l));
endtype

type ListOfBookType is FormalListType actualizedby BookType using
  sortnames Book for Element
endtype

type BookListType is ListOfBookType renamedby
  sortnames BookList for List
endtype

type BorrowedBookListType is BookListType renamedby
  sortnames BorrowedBookList for BookList
  opnnames  bookListLength for length
endtype
```

Fig.5 Data types related to borrowed book list

```
process MainProc [StaffGate, BorrowerGate]
      (CollectedQueue:CollectedBookQueue, BorrowedQueue:BorrowedBookQueue,
       MaxBorrowable:Nat) : noexit :=
( StaffGate ?query:QueryByStaffOnly ?param1:BorrowerName ?param2:Book
            [(query eq doBorrow)
             and
             (bookListLength(bookList(param1, BorrowedQueue)) le
              MaxBorrowable)];                                        (* <--(A) *)
  MainProc [StaffGate, BorrowGate] (CollectedQueue,
                                    borrow(param1, param2, BorrowedQueue),
                                    MaxBorrowable)                    (* <--(B) *)
 [] (* behavior expression of doReturn *)
 [] (* behavior expression of doAdd *)
 [] (* behavior expression of doDelete *)
 [] StaffGate ?query:QueryByStaffOnly ?param:BorrowerName
            !bookList(param, BorrowedQueue)
            [query eq doShowBorrowedBooks];
    MainProc [StaffGate, BorrowGate]
             (CollectedQueue, BorrowedQueue, MaxBorrowable)           (* <--(C) *)
 [] (* behavior expression of doShowLastBorrower *)
 [] (* behavior expression of doShowListByAuthor *)
 [] (* behavior expression of doShowListBySubject *)
)
[]
( BorrowerGate ?query:QueryByBothStaffAndBorrower ?param:AuthorName
               !searchByAuthor(param, CollectedQueue)
               [query eq doShowListByAuthor];
  MainProc[StaffGate, BorrowerGate]
         (CollectedQueue, BorrowedQueue, MaxBorrowable)              (* <--(D) *)
 [] (* behavior expression of doShowListBySubject *)
)
endproc (* MainProc *)
```

Fig.6 Definition of MainProc

```
specification LibrarySystem[StaffGate,BorrowerGate](MaxBorrowable:Nat):
                                                                    noexit
(* standard types are from Appendix A of DIS 8807 *)
library
  NaturalNumber, Boolean, NonEmptyString
endlib
(* type definitions *)
behaviour
  MainProc [StaffGate, BorrowerGate] (empty, empty, MaxBorrowable) (* <--(A) *)
where
  process MainProc [StaffGate, BorrowerGate]
         (CollectedQueue:CollectedBookQueue, BorrowedQueue:BorrowedBookQueue,
          MaxBorrowable:Nat) : noexit :=
   (* behavior expression for StaffGate *)
  []
   (* behavior expression for BorrowerGate *)
   endproc
endspec
```

Fig.7 The whole specification of Library System

3.2.1 Logical structure of process

We graphically show the outermost structure of `LiftSystem` in Fig.8. There are three gates to interface between the `LiftSystem` and the outer environment. The gate `LiftButton` manages the buttons at each lift. The gate `FloorButton` is for up or down buttons at each floor. And `LiftMovement` is for the lift movement such as arriving from or leaving for up or down stairs. From the specification of 3.1, we know the system should be parameterized with the number of lifts (n) and the number of floors (m). Therefore, there are two parameters `e` and `f` in Fig.8. These parameters will be initialized to be n and m when the `LiftSystem` is activated.

3.2.2 Gate attributes

Each gate has attributes as shown in Fig.8. These attributes should be interpreted as follows.

For the gate `LiftButton`, the event
`LiftButton?lift_num:Nat?floor_num:Nat`

is of pressing the (floor_num)-th floor request button at the (lift_num)-th lift. The event

```
  LiftButton?lift num:Nat
            ?emergency_or_BackInService:
                         EmergencyNotification
```

is of pressing the emergency or `BackInService` button at the (lift_num)-th lift. And the event

```
  LiftButton!lift num!floor_num
```

is of cancelling the illumination of the (floor_num)-th floor request button at the (lift_num)-th lift.

For the gate `FloorButton`,

```
  FloorButton?floor_num:Nat
            ?up_or_down:Direction
```

is the event of pressing the up_or_down floor button at the (floor_num)-th floor. The event

```
  FloorButton!floor_num!up_or_down
```

is of cancelling the illumination of the up_or_down floor button at the (floor_num)-th floor.

For the gate `LiftMovement`,

```
  LiftMovement!lift_num!floor_num
             !up_or_down!leave_or_arrive
```

is to indicate (lift_num)-th lift arrives/leaves from/for up/down stairs at the (floor_num)-th floor, depending on values of up_or_down and leave_or_arrive. The types of up_or_down and leave_or_arrive contain constants to indicate the directions, but we omit their definitions for brevity.

This design of gate attributes is one of the most important things when programming in LOTOS. Given a specification in some natural language, we have to add "meanings" or "interpretations" to gate attributes. There is no explicit way to describe these "meanings" of gate attributes in LOTOS specification. This point should be improved in a later version of LOTOS.

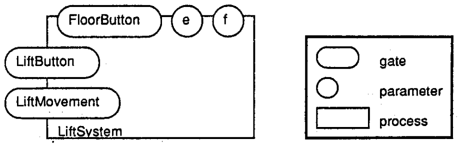
3.2.3 Data types to be used

As in CSP approach[2], we use two queues to decide the direction of lift movement: one queue is for getting off requests from lift buttons at each lift and the other one is for picking up request from up or down buttons at each floor. Each queue has the sort `Nat` to indicate a floor number to be serviced as its element. Therefore, as shown in Fig.9, we define a queue `FloorQueueType` whose elements are floor numbers, and define `LiftButtonQueueType` and `FloorButtonQueueType` by renaming sort names for `FloorQueue`.

We create the same process corresponding to each lift. Each lift process contains a queue of `LiftButtonQueueType`. This queue keeps all getting off requests caused by the lift buttons. One queue of `FloorButtonQueue` exists in the whole system and is shared by all lift processes. To decide the floor number to be serviced, each lift process looks around these two queues. To minimize the waiting time for both requests, we need operators `atMost` and `atLeast` which return boolean values according to:

`atMost(queue, f) = true` if `f` is the largest number in `queue`, and
`atLeast(queue, f) = true` if `f` is the smallest number in `queue`.

Roughly speaking, if a lift arrives at the floor `f` from downstairs and `atLeast(queue, f)` is `true`, then this lift has to go upward. These queue types are prepared just to make this decision.



```
LiftButton ?lift_num:Nat ?floor_num:Nat
LiftButton ?lift_num:Nat ?emergency_or_BackInService:EmergencyNotification
LiftButton !lift_num !floor_num
FloorButton ?floor_num:Nat ?up_or_down:Direction
FloorButton !floor_num !up_or_down
LiftMovement !lift_num !floor_num !up_or_down !leave_or_arrive

e:Nat (* number of lifts *)
f:Nat (* number of floors *)
```

Fig.8 Interface of LiftSystem and gate attributes

```
 type FloorQueueType is NaturalNumber, Boolean
  sorts FloorQueue
  opns empty   :                             -> FloorQueue
       _+--_   : Nat, FloorQueue             -> FloorQueue
       _--+_   : FloorQueue, Nat             -> FloorQueue
       dequeue : FloorQueue, Nat             -> FloorQueue
       _eq_    : FloorQueue, FloorQueue      -> Bool
       isEmpty : FloorQueue                  -> Bool
       atMost, atLeast, isInQueue : FloorQueue, Nat -> Bool
  eqns forall fq, fq1, fq2 : FloorQueue, f, f1, f2 : Nat
   ofsort FloorQueue
    f +-- empty = empty --+ f;
    f1 +-- (fq --+ f2) = (f1 +-- fq) --+ f2;
    isInQueue(fq --+ f1, f) and (f1 eq f) =>
      dequeue(fq --+ f1, f) = fq;
    isInQueue(fq --+ f1, f) and (f1 ne f) =>
      dequeue(fq --+ f1, f) = dequeue(fq, f);
    not (isInQueue(fq, f)) => dequeue(fq, f) = fq;
   ofsort Bool
    isEmpty(empty) = true;
    isEmpty(f +-- fq) = false;
    empty eq empty = true;
    fq eq fq = true;
    (f +-- fq) eq empty = false;
    empty eq (f +-- fq) = false;
    (f1 +-- fq1) eq (f2 +-- fq2) = (f1 eq f2) and (fq1 eq fq2);
    atMost(empty, f) = true;
    atMost(fq --+ f1, f) = (f1 le f) and atMost(fq, f);
    atLeast(empty, f) = true;
    atLeast(fq --+ f1, f) = (f1 ge f) and atLeast(fq, f);
    isInQueue(empty, f) = false;
    isInQueue(fq --+ f1, f) = (f1 eq f) or isInQueue(fq, f);
 endtype (* FloorQueueType *)

 type LiftButtonQueueType is FloorQueueType
  renamedby sortnames LiftButtonQueue for FloorQueue
 endtype (* LiftButtonQueueType *)

 type FloorButtonQueueType is FloorQueueType
  renamedby sortnames FloorButtonQueue for FloorQueue
 endtype (* FloorButtonQueueType *)
```

Fig.9 Definition of queues for Lift

```
specification LiftSystem [LiftButton, FloorButton, LiftMovement]
                                               (e:Nat, f:Nat) : noexit
library
  NaturalNumber, Boolean
endlib
(* type definitions *)
behavior (* of LiftSystem *)
  [(e gt 0) and (f gt 0)] ->
     MainProcess[LiftButton, FloorButton, LiftMovement](e, f)
where (* subprocs of LiftSystem *)
  process MainProcess [LiftButton, FloorButton, LiftMovement]
                                          (e:Nat, f:Nat) : noexit :=
   hide PickupQueueAtMost, PickupQueueAtLeast, PickupQueueDequeueIf in
    (
    LiftControll[LiftButton, LiftMovement, PickupQueueAtMost,
                 PickupQueueAtLeast, PickupQueueDequeueIf] (e)
    |[PickupQueueAtMost, PickupQueueAtLeast, PickupQueueDequeueIf]|
    PickupQueueControll[FloorButton, PickupQueueAtMost, PickupQueueAtLeast,
                        PickupQueueDequeueIf] (f)
    )
     where
     (* subprocesses for MainProcess *)
   endproc (* MainProcess *)
endspec (* LiftSystem *)
```

Fig.10 The header part of the Lift specification

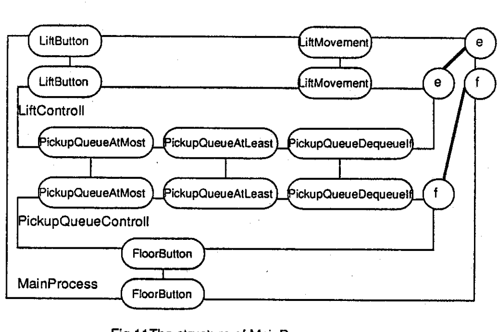


Fig.11 The structure of MainProcess

3.2.4 Subprocesses

Fig.10 shows the the header part of the Lift specification. At the entrance of `MainProcess`, parameters `e` and `f` are checked whether they are positive numbers. In `MainProcess` there are two parallel behavior expressions. One is for the lift control which manages each lift movement. The other one is for the pick up queue control which is collecting the pick up requests from up or down buttons at each floor. In Fig.11, we illustrate the relations between these subprocesses. In Fig.10, three gates are hidden to avoid unnecessary interference between inner processes and the outer environment.

The process `LiftControll` keeps each lift process. The process `PickupQueueControll` keeps queues from floor buttons. Each lift process communicate with pickup queues via three gates; `PickupQueueAtMost`, `PickupQueueAtLeast`, and `PickupQueueDequeueIf`. Note that, in this example, we prepared these three gates for three queries to the pickup queues. In the case of Library problem, we use one gate for several operations. This is a controversial issue of tradeoffs between the number of gates and the number of attributes on a gate.

3.2.5 Lift process creation

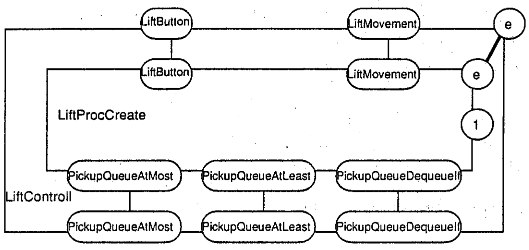
Fig.12(a) shows a subprocess of `LiftControll`. `LiftControll` includes the process `LiftProcCreate`. Fig.13 shows the process `LiftProcCreate` and its subprocesses in LOTOS text. As a standard way of creating similar processes from a template process[6], we have to use a recursive form like the definition of `LiftProcCreate` where `MaxLifts` limits the number of parallel processes `EachLiftProc`. The initial value `1` of `LiftId` is given by the process `LiftControll`, and then incremented to `MaxLifts` during recursive calls of `LiftProcCreate`. At each call, `EachLiftProc` is created with `LiftId`(see Fig.12(b)). `EachLiftProc` contains the process `LiftIdentification` which has a process parameter `LiftId`. When some event occurs at `LiftButton`, three subprocesses of `EachLiftProc`, i.e. `LiftButtonInput`, `Lift`, and `LiftQueue`, are synchronized by the `LiftIdentification` process. Since `LiftIdentification` only accepts the event with the parameter `LiftId`, every event on `LiftButton` gate is only accepted by other three processes which have the same `LiftId`.

We can decompose these three processes, but we omit further explanations.

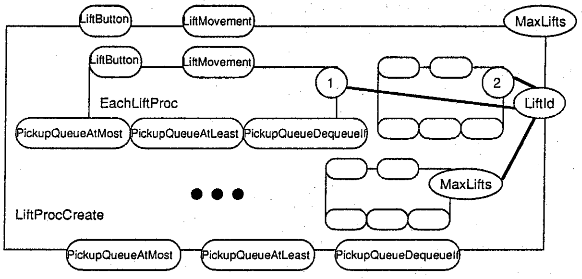
4. Discussions

4.1 From the viewpoint of the specification language

(1) The notion of sort and subsort is needed. For example, the operations of staffs and borrowers are defined in Fig.9. If there is some construct for "subset" or "subsort", we will be able to describe these two data types much easily. In OBJ2[9], we have introduced these constructs.
(2) The meaning of gate attributes should be explicitly specified somewhere in the specification. From the specification, we could hardly understand the meaning of gate attributes unless there is specification on the meaning of gate attributes.
(3) Some notion of "theory" will be needed. For example, we often use `eq` operators when using guarded



(a) process PickupQueueControll



(b) process LiftProcCreate

Fig.12 Lift process creation

```
process LiftControll
              [LiftButton, LiftMovement, PickupQueueAtMost,
               PickupQueueAtLeast, PickupQueueDequeueIf] (e:Nat) : noexit :=
  LiftProcCreate
       [LiftButton, LiftMovement, PickupQueueAtMost, PickupQueueAtLeast,
        PickupQueueDequeueIf] (e, 1)
where (* subprocs for LiftControll *)
  process LiftProcCreate[LiftButton, LiftMovement,  PickupQueueAtMost,
                         PickupQueueAtLeast, PickupQueueDequeueIf]
                         (MaxLifts:Nat, LiftId:Nat) : noexit :=
    [LiftId lt MaxLifts] ->
     (LiftProcCreate
         [LiftButton, LiftMovement, PickupQueueAtMost, PickupQueueAtLeast;
          PickupQueueDequeueIf] (MaxLifts, Succ(LiftId))
     |||
      EachLiftProc
         [LiftButton, LiftMovement, PickupQueueAtMost, PickupQueueAtLeast,
          PickupQueueDequeueIf] (LiftId))
  where (* subprocesses for LiftProcCreate *)
    process EachLiftProc
                [LiftButton, LiftMovement, PickupQueueAtMost,
                 PickupQueueAtLeast, PickupQueueDequeueIf]
                 (LiftId:Nat) : noexit :=
     (hide LiftQueueEnqueue, LiftQueueAtMost, LiftQueueAtLeast,
           LiftQueueDequeueIf, EmergencyNotify  in
      (
        LiftButtonInput[LiftButton, LiftQueueEnqueue, EmergencyNotify]
      |[LiftQueueEnqueue]|
        LiftQueue[LiftButton, LiftQueueEnqueue, LiftQueueAtMost,
                  LiftQueueAtLeast, LiftQueueDequeueIf] (empty, LiftId)
      |[LiftQueueAtMost, LiftQueueAtLeast, LiftQueueDequeueIf,
        EmergencyNotify]|
        Lift[LiftMovement, PickupQueueAtMost, PickupQueueAtLeast,
             PickupQueueDequeueIf, LiftQueueAtMost, LiftQueueAtLeast,
             LiftQueueDequeueIf, EmergencyNotify]  (1, nil, LiftId)
      )
     )
     |[LiftButton]|
      LiftIdentification[LiftButton](LiftId)
    where (* subprocs for EachLiftPros *)

      process LiftIdentification[LiftButton](LiftId) : noexit :=
        LiftButton?Identifier:Nat ?AnyArg:LiftButtonIO[Identifier eq LiftId];
        LiftIdentification[LiftButton](LiftId)
      endproc (* LiftIdentification *)

      process LiftButtonInput [...]
      endproc (* ButtonInput*)

      process Lift [...](floor_position:Nat, dir:Direction, LiftId:Nat)
      endproc (* Lift *)

      process LiftQueue [...](q:LiftButtonQueue, LiftId:Nat)
      endproc (* LiftQueue *)

endproc (* LiftControll *)
```

Fig.13 process LiftControll and LiftProcCreate

expressions. In these cases, we need to define the general features. In OBJ2, we have introduced these constructs as well.
(4) Data type specification should be more hierarchical. Processes can be hierarchically defined by using the key word **where**. But data types are written in a flat text.
(5) Error handling or exception handling mechanism should be implemented in data type definitions. For

example, when trying to delete unentried books from the data base in Library system, we are using `dequeueIf` operator, but if some standard exception handling mechanism is implemented, it will be defined much easily.
(6) When generating processes from a process template, a "standard" way of creating is to use `Identification` process like `Lift` processes. This construction is hard to understand. We prefer to use process or event labelings like CSP approach.

4.2 From the viewpoint of programming environment

There are several programming environments for developing concurrent systems[10]. With these in mind, we list here the necessary conditions of the programming environment for LOTOS.

(1) Every data type or process should be entered interactively like Lisp programming environment, because the order of defining data types or processes are not always top down or bottom up. Moreover, when we define some concrete data type, e.g., `List`, and wish to change it into a formal type, we should be allowed to do this kind of restructuring.

(2) Whenever some objects are entered, symbolic execution should be preformed as much as possible in order to confirm the definitions.
(3) Hierarchical structure of data types and processes should be displayed in a proper way. When writing specifications, we often refer the names of sorts, equations of data types, or the order of gates. To do this, we need some assistance of the browsing definitions around.
(4) For example, a formal type, e.g., `List`, should contain usual operators like `head` or `tail`. But we do not use these operators in our specification Library. In these cases, there should be some tools to indicate these unused operators.
(5) When using a parameterized type, we need to know which parameters should be actualized and which should not.
(6) Especially in case of process definitions, the support by graphic interface is quite useful, because it is very hard to trace the meaning of processes from the LOTOS text. As an activity in ISO, there are several proposals for G-LOTOS which intend to incorporate this[11]. Most proposals are similar to some sophisticated "flow chart" of LOTOS. This might not resolve some substantial difficulties concerning parallelism. Because the programmer wishes to know the event transitions by the expressions in stead of the program text itself. In the previous section, we use several figures to explain the Lift problem. LOTOS environment should accept these graphical data as part of the specification.
(7) Generating processes from some template process is hard to comprehend. As we stated, the "standard" way is to prepare identification processes to distinguish each process. This is a bit complicated. So, we need some graphic representations for generating processes from a process template.

4.3 Miscellaneous comments

(1) Generally, a given problem includes both features of static part and dynamic part. LOTOS has these two features. So it is quite natural to use LOTOS to specify other problems than OSI protocols.
(2) We can write some part of a given problem by both data type definitions and process ones. We need some criteria to use data types and processes. This is a tradeoff between data types and processes.
(3) In our case, we use the Library problem to exemplify the data type definition facilities in LOTOS, and the Lift problem to the process definitions. We think these two problems are typical ones to test the two distinct abilities of such languages as LOTOS.
(4) LOTOS explicitly specifies the communication channels between processes. On the other hand, CSP does not. So, we can trace the structure of process communication in LOTOS easier than CSP, although LOTOS text becomes longer than CSP ones.
(5) The distinction of using between gates and attributes is also a tradeoff problem. We can communicate via only one gate if we prepare a lot of attributes on that gate. But if we prepare many gates, then we can reduce attribute patterns. We need some standard discipline of using these two means.

5. Conclusions

We can say that the parallelism should be explicitly expressed by parallel operators. But this parallel definition part should be in a specification as little as possible, because it is very hard to comprehend the meaning of the parallelism if there exist unnecessary process definitions.

This means the writer of the specification should localize the parallelism in a given problem. This is a tradeoff between abstract data type definitions and parallel process behavior expressions.

In case of the Library problem, most parts of the specification are data type definitions. The main data type is a queue for the library book data base. Most operations like add a book to the data base or search by author are realized as operators defined in a data type. The only parallel behavior is for "Staff" process and "Borrower" process. These two processed should be independent.

In case of the Lift problem, there should be several processes which are essentially in parallel. The main part of data type definition are request queues from floors of lifts. Each process dedicated to each lift looks around these queues and decides its direction to move. In LOTOS, there are "standard" techniques to create actual processes from some process template. When communicating to processes by different commands, there are two alternatives: first one is to prepare many gates corresponding to each command; the other one is to prepare many attributes on one gate. This is also tradeoffs between number of gates and attributes.

Although we did not state in the Lift problem, we used >> operators for sequencing. We are able to rewrite these processes by a data type if necessary. These >> constructs should not be used frequently. We also used [> operator and internal event symbol **i** to specify the emergency behavior in the Lift system. We think these constructs are much useful. In the case of CSP, there are no such "exception handling operators".

We wish to point out several requirements for LOTOS which are found during our experiments. Strictly speaking, our data type definitions presented in the previous sections are not legal, because we suppose that LOTOS satisfies the following requirements.

(1)We wish to define a renamed type even when it inherits two or more other data types. For example, the following specification is illegal in the current LOTOS definition[3]:

```
type A is B,C renamedby
  sortnames X for Y
  opnnames W for Z
endtype
```

(2)We wish to add sorts or opns to a renamed type. For example, the following specification is illegal in the current LOTOS.

```
type A is B renamedby
  sortnames X for Y
  opnames W for Z
  sort S
  opn O: X -> S
endtype
```

(3)We wish to use both actualize and rename in the same data type definition.

Acknowledgement
The authors thank Dr. A. Tojo, the director of the computer science division of Electrotechnical Laboratory, for his encouraging us to carry out the present study. They are also indebted to Mr. K.Okada for his helpful discussions. Mr.A.Nakagawa carefully read this manuscript and corrected errors.

References

[1]*Problem Set for the Fourth International Workshop on Software Specification and Design*, IEEE, Proc. of Fourth International Workshop on Software Specification and Design, pp.ix-x, April 1987.
[2]M.D. Schwartz and N.M. Delisle, *Specifying a lift control system with CSP*, ibid, pp.21-27.
[3]ISO/DIS 8807, *Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal behaviour.*
[4]ISO/IEC DTR 9571, *Information processing systems - Open systems interconnection - LOTOS Description of the Session Service - Type 2.*
[5]ISO/IEC DTR 9572, *Information processing systems - Open systems interconnection - LOTOS Description of the Session Protocol - Type 2.*
[6]ISO/IEC JTC1/SC21/WG1 N556, *Formal description of a Transport protocol in LOTOS for the ISO/CCITT Guidelines on the application of Estelle, LOTOS and SDL.*
[7]Robin Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, 92, Springer Verlag, 1980.
[8]*Software Environment for Design of Open Distributed Systems, HIPPO-LOTOS simulator*, the University of Twente, The Netherlands.
[9]K.Futatsugi, J.Goguen, J.-P.Jouannaud, and J.Meseguer, *Principles of OBJ2*, Proc. of 1985 Symposium of Principles of Programming Languages, ACM, pp.52-66, 1985.
[10]*Special Issue on Tools for Computer Communication Systems*, IEEE Trans. Software Engineering, Vol.14, No.3, March 1988.
[11]ISO/IEC JTC1/SC21/WG1 N549, *Graphics syntax for LOTOS.*
[12]J.M. Wing, *A study of 12 specifications of the library problem*, IEEE Software, pp.66-76, July 1988.