A TEMPORAL ORDERING SPECIFICATION OF SOME SESSION SERVICES

Vincenza Carchiolo, Alberto Faro , Giuseppe Scollo

Istituto di Informatica e Telecomunicazioni
Facoltà di Ingegneria - Università di Catania
viale A.Doria 6, 95125 CATANIA (Italy)

ABSTRACT

The achievement of widely accepted standards for Open Systems Interconnection (OSI) is closely tied to the ability of producing unambiguous and implementation independent specifications of related protocols and services. LOTOS, the Language fOr Temporal Ordering Specification, is a Formal Description Technique (FDT) whose definition, though not completed, has already reached such a state as to allow trial specifications of rather sophisticated services and protocols. This paper explores the specification in LOTOS of some of the session services whose discussion is underway within various standardization bodies. Conciseness of specification is tried by adopting a few notational variants which are guessed to be useful at various OSI layers. The session services selected for this trial specification comprise the Basic Combined Subset (BCS) enriched with the Expedited Data service.

1. INTRODUCTION

The challenging goal of producing a set of interrelated standards for Open Systems Interconnection (OSI) has already forced the standardization bodies mainly involved with it - ISO and CCITT - to set up ad-hoc groups for the definition of Formal Description Techniques (FDT's) such as to allow the production of unambiguous and implementation independent specifications of OSI protocols and services.

The narrative form in which those standards are currently being defined will be necessary anyway for helping their understanding "at a first glance"; however, formal definitions must be provided if one wants to give a precise meaning to concepts like "integration of a protocol with an underlying service for offering an higher layer's service" or "OSI conformance assessment of products"; while the former concept comprises the verification of a protocol specification, the latter opens the scope of validation of products implementing the protocols.

It is desirable that both the verification problem and the validation one might employ a common notational framework so as to directly relate,for instance, the definition as well as the results of testing sequences (which constitute the typical validation runs) to a formal conformance statement which should be a major constituent of the (hopefully verified) protocol specification.

Integration of verification and validation in a common notation is one of the major motivations for the development of the temporal ordering specification approach <V 83>; various authors <B 83>, <K 83>, have defined languages showing an algebraic flavour which owes much to Milner's Calculus of Communicating Systems <M 80>.
The Subgroup C of the ISO Ad-Hoc Group on FDT's has produced a Draft Tutorial on the Language fOr Temporal Ordering Specification (LOTOS) <ISO 83a> which, though lacking features like value specification facilities <CFMS 84>, can be considered as having reached a rather stable definition of the kernel concepts as well as of the relevant notation. This language is currently being checked against trial specifications of OSI draft service and protocol standards.

Clear formal specifications can be used in a variety of situations, e. g. by implementors as a guidance both in the design and in the testing phases of their products, by service users for understanding at the desired level of detail what the service can provide them with, by third-party check agencies for industrial or legal certification.
The availability of mechanical tools for checking that an implementation complies with the requirements of its specification will greatly facilitate this process. The development of such tools for LOTOS is planned for the near future.

The aim of this paper is to present a temporal ordering specification of the Basic Combined Subset (BCS) of the Session Service (SS) enriched with the Expedited Data service <ISO 83b>. This choice is motivated by two considerations: first,

among the OSI layers, the Session one is well known for being a rather complex one, with a variety of services and so-called functional units (the top-down taste induced the authors to start with the Service); second, the specification of a richer target (like, e.g., Synchronization or Activity services specification) was inhibited by the available time and space.

A formal description technique is proposed which is based on LOTOS extended by a few notational variants to gain in conciseness and readability.

Thus sect.2 gives a brief review of LOTOS and of some extensions. Sect.3 discusses the service model adopted. Finally sect.4 gives the target formal specification. Appendices A and B provide some definitions which are necessary for a complete understanding of the target specification and in order to obtain a self contained paper.

2. BRIEF REVIEW OF LOTOS AND SOME EXTENSIONS

LOTOS uses the model of event algebra introduced by Milner in <M 80> and allows one to describe the behaviour of an information system by describing the temporal ordering of its interactions with its environment.

In LOTOS a system consists of a set of parts interacting each other and/or with the environment. The interaction is the basic concept of LOTOS. It is defined as a common activity of parts on information. The simplest kind of interaction is called "event". An event consists of the synchronized passing of a value at a gate from a part to another one at a given moment in time. Thus, one can associate with an event: the time of passage, the value passed, the gate of passage and the direction of passage. An event generally has a non-null time duration but it is atomic in the sense that it cannot be disrupted; the time associated with an event is the time at which it is terminated.

The event expressions of LOTOS allow one to specify the behaviour of a system (or part) by describing the temporal ordering of the events observable at its event gates. A process is any behaviour specified in LOTOS. An event is a particular kind of process. LOTOS uses the following simple ordering principles: a process can be
1) enabled by the termination of
2) disabled by the starting of
3) unrelated to
another process; in addition a process can be influenced by the values of events in different processes.

A LOTOS expression defines only one of the processes involved in an event, so it gives only the perspective that a process is prepared to participate in this event. In particular the following principal cases are possible :

- a process can be prepared to accept the value x of type t at gate a. x denotes a free variable which is bound by the occurrence of the event to the value accepted by the process when the event has taken place. This is expressed in LOTOS by using the notation: a?x:t.

- a process can be prepared to produce the output value e at gate a. This is expressed in LOTOS by using the notation: a!e (*).

The concept of event at some internal gate is also used in LOTOS to represent unknown internal conditions which can influence the observable behaviour at the outside, obtaining in this way a certain degree of internal nondeterminism. LOTOS has only one representation of the events at internal gates by means of the notation i.

Event expressions are built by using the operators of sequence, choice, composition, disable and restriction defined in appendix A, where some other features of LOTOS are also shown which make more easy and expressive the representation of a system behaviour. Finally appendix B presents some operators which might be added to the language without affecting its kernel in order to obtain some gain in conciseness and readability of specifications, at least for OSI, as discussed by the authors in <CFMS 84>. These operators are mere notational variants of the kernel language definition (in the sense formalized in <BPW 81>).

LOTOS still lacks value specification facilities for both function and type definitions. Currently a PASCAL-like type specification notation is provisionally adopted with a few extensions like:

string m ... M of <type>

representing a sequence of elements of type <type> whose length can range from m through M; M can be "undefined": in this case, when m=0 the shorter notation string of <type> is adopted. The function definitions needed for the specification tried in this paper have been introduced informally.

3. A MODEL OF THE BASIC COMBINED SUBSET OF THE SESSION SERVICE

In this section a brief informal description is given of the session service model which will be formalized in the next section. Session services are offered to the Presentation Layer at Session Service Access Points (SSAP's) through defined service primitives. The communication at two SSAP's is assumed as a model of the service (fig. 1), which is specified as a single connection between a pair of session service users.

The specification is purely extensional: it comprises the offering of service primitives execution at SSAP's; these events actually take place when the specified service is experimented by its users; each SSAP has been specified as a pair of gates.

In particular the SS provides its users with the means to establish a connection, to exchange data in a synchronized manner, to organize the data exchange into activities and dialogue units and to orderly release the connection. A negotiation is performed in the connection establishment phase in order to let the users agree upon the quality of service (defined by parameters like throughput, error rate, transit delay etc.), the

initial assignment of tokens - which control the exclusive right to request the associated services -, the service elements to be available after the connection is established (users' session requirements).

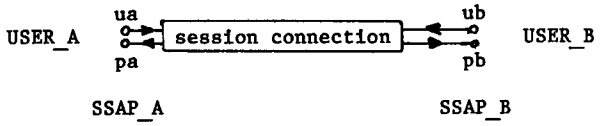

Fig.1

The token concept can be recalled as follows: a token is defined w.r.t. a set of associated services; a token is either available, in which case it can be assigned to only one user (its "owner") who has the exclusive right to request the associated services, or not available, in which case either both or none of the users can request the associated services (hence none of the users has the exclusive right to request them).

Service elements are provided by distinct Functional Units (FU's): the kernel one comprises connection establishment, normal data exchange and orderly release which are essential and not negotiable; either duplex (two way simultaneous) or half duplex (two way alternate) normal data exchange has to be agreed during the negotiation. The other FU's provide the optional service elements; they are selected according to the negotiated session requirements ([2]) . Any combination of the kernel FU with either duplex or half-duplex FU and with optional FU's may be defined as a service subset ([3]). The Draft International Standard <ISO 83b> (hereafter termed "DIS") proposes three different subsets:
1) Basic Combined Subset (BCS)
2) Basic Synchronized Subset (BSS)
3) Basic Activity Subset (BAS).
Only the "data" token can be available in the BCS, which comprises the following FU's:
- kernel
- half duplex
- duplex.
The expedited data FU provides the corresponding additional optional service element. None of the subsets proposed in the DIS comprises it explicitly. The authors argue that it can be optionally added to any of them ([4]). This service element can be provided only when the transport expedited data service is available.

According to the service conventions outlined in <ISO 83c> a service element can be of type confirmed, unconfirmed or provider-initiated; Tab.1 summarizes the service elements and types of BCS+expedited data FU's. The service elements are denoted by the names of the corresponding service primitives, abstracted from their type. The following simplifications have been assumed in the formalization of the next section:
- the interactions denoted by service primitives are modeled as atomic events;
- no quality of service negotiation is modeled.
Short informal descriptions of the specified service elements are given in the form of comments throughout the formal specification.

FUNCTIONAL UNIT	SERVICE ELEMENT	SERVICE TYPE
kernel	S_CONNECT	confirmed
	S_DATA	unconfirmed
	S_RELEASE	confirmed
	S_U_ABORT	unconfirmed
	S_P_ABORT	prov.-init.
half duplex	S_TOKEN_PLEASE	unconfirmed
	S_TOKEN_GIVE	unconfirmed
duplex	no additional service elements	
expedited data	S_EXPEDITED_DATA	unconfirmed

Tab.1

4. FORMAL SPECIFICATION

The specification is organized as follows:

- a PASCAL-like type declaration which defines all the used types but one, the abstract type "nxqueue", which has been formally specified in <CFMS 84>; the latter consists of a sequence of two distinct kinds of objects, "normal" (N) and "expedited" (X): Fig.2 gives a pictorial representation of its access functions; the function "nox" returns the number of expedited objects in the sequence; the symbol * denotes the empty sequence;

- a temporal ordering specification which constitutes the "body" of the specification: here the processes are defined in a top-down fashion, hence the definitions of the atomic ones are placed at the bottom; these may be looked at as a quick glossary of the abbreviations of service primitives.

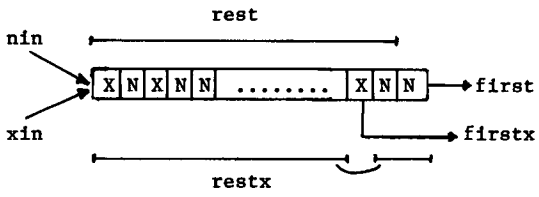

Fig. 2

In order to allow a nice use of the restriction operator, each SSAP has been specified as a pair of gates (a "channel"), one for request and response primitives, the other for indication and confirm ones: if c is a channel, the functions u(c) and p(c) return its respective gates, and the function remote(c) returns the channel to which the remote user is attached.

Comments are inserted in the form (* comment *) as an aid to understanding the formal expressions which follow them.

```
Syst SS_BCS_X

(* type declarations *)

type

    (* Architecture elements *)

    (* SSAP channels *)      SSAP =          (a,b);
    (* SSAP gates *)         gate = (ua,pa,ub,pb);
    (* Functional Units *)   FU   =      (K,H,D,X);

    (* Session Service Primitive *)

SSP = record case code:sspcode of
(*
 S_CONNECT - This service is used to establish a
session connection and to negotiate the session
requirements and initial token assignment; a
result parameter (accept or reject) is given in
the response/confirm primitive: a reject value
can be qualified as due to user or provider (the
latter only in the confirm primitive); a limited
amount of user data can be carried in the service
primitives.
*)
(*S_CONNECTrequest/indication*)  Ci:(parms:crip);
(*S_CONNECTresponse/confirm*)    Cc:(parms:crcp);
(*
 S_DATA - This service is used to carry normal
Session Service Data Units (SSDU's) over an esta
blished session connection. When the data token
is available (half-duplex FU selected) only its
owner can send data by means of a S_DATA request.
*)
(*S_DATArequest/indication*)     D: (parms:SSDU);
(*
 S_RELEASE - This service is used to orderly re
lease an established session connection without
loss of data; all the available tokens are re
quired to be assigned to the user initiating this
service; a limited amount of user data can be
carried in the service primitives.
*)
(*S_RELEASErequest/indication*)  Ri: (parms:rud);
(*S_RELEASEresponse/confirm*)    Rc:(parms:rrcp);
(*
 S_U_ABORT - This service is used to instant-
aneously release the session connection; this may
cause loss of data; a few octects of user data
can be carried in the service primitives.
*)
(*S_U_ABORTrequest/indication*)  AU: (parms:aud);
(*
 S_P_ABORT - This service is used to indicate the
release of a session connection for reasons
internal to the SS-provider; this may cause loss
of data; a reason parameter is provided which may
assume one of the values: transport disconnect,
protocol error, undefined.
*)
(*S_P_ABORTindication*)          AP: (parms:par);
(*
 S_TOKEN_PLEASE - This service is used to request
specific available tokens; this request can be
issued only by the user who does not own the re
quested tokens. An S_TOKEN_PLEASE indication does
not constrain the user to give the requested
tokens.
*)
(*S_TOKEN_PLEASErequest/indic.*) P : (parms:tpp);
(*
 S_TOKEN_GIVE - This service is requested by the
owner of some tokens to surrender one or more of
them to the other user.
*)
(*S_TOKEN_GIVErequest/indic.*)   T : (parms:tk);
(*
 S_EXPEDITED_DATA - This service is used to carry
Expedited Session Service Data Units (XSSDU's)
over an established session connection. The
transfer of an XSSDU is not constrained by the
right to send normal SSDU's. The SS provider
guarantees that an XSSDU will not be delivered
after any subsequently submitted normal SSDU on
that session connection. The size of an XSSDU is
limited to a few octets.
*)
(*S_EXPEDITED_DATAreq./indic.*)  X:(parms:XSSDU)
                                              end

    (* service primitive parameter types *)

    (* S_CONNECT *)

(*conn. user data*)  cud=string 0..512 of octect;

(* the following unspecified types are used for
S_CONNECT primitive parameters which are unspeci-
fied also in the DIS *)

(*user-provided session
 connection identifier*)        sci = unspecified;
(*SSAP identifier*)           ssapid = unspecified;
(*reason for failure in
 user reject of connection*)    urf = unspecified;
(*reason for failure in pro
 vider reject of connection*)   prf = unspecified;

    (* S_CONNECT request/indication *)

(*session requirements: half duplex
 FU,duplex FU,expedited data FU*)   sr = (H,D,X);
(*requestor requirements*)          rsr=set of sr;
(*proposed token assignment (by re
 questor): requestor side, accept
 or side, acceptor choice*)     pta=(rqs,acs,ach);

crip = record conn_id:sci; calling,called:ssapid;
       ud:cud; case sr:rsr of (H),(H,X),(H,D,X):
                  (ita:pta) end;

    (* S_CONNECT response/confirm *)

(*result of connection establishment*)
             ru = record case r:(acc,urj,prj) of
  (*accepted*)                  acc:();
  (*user rejected*)             urj:(rs:urf);
  (*provider rejected*)         prj:(rs:prf) end;
(*half-duplex allowed
 session requirements*)              hsr = (H,X);
(*duplex allowed
 session requirements*)              dsr = (D,X);
(*acceptor session requirements*)   asr=union of
                        (set of hsr,set of dsr);
(*token assignment*)                 ta = (rqs,acs);

crcp=record conn_id:sci; called:ssapid; ru:ru;
     ud:cud; case sr:asr of (H),(H,X):
                (ita:ta) end;

    (* S_DATA *)           SSDU = string of octet;
```

```
    (* S_RELEASE *)

(* release user data *)       rud = cud;
(* release result *)          rru = (affirmative);

    (* S_RELEASE response/confirm *)

rrcp = record ru:(rru); ud:rud end;

    (* S_U_ABORT *)

(* user data *)        aud = string 0..9 of octet;

    (* S_P_ABORT *)

(* reason *)                      par = (td,pe,uf);

    (* S_TOKEN_GIVE *)

(*set of available tokens*)      tk = set of (da);

    (* S_TOKEN_PLEASE *)

(* user data*)         pud = string 0..64 of octet;
                  tpp = record tk:tk; ud:pud  end;

    (* S_EXPEDITED_DATA *)

(*expedited SSDU*) XSSDU = string 1..14 of octet;

    (* data transfer and orderly release
       phases parameters                    *)

(*token availabil./assignment*) taa=(na,rqs,acs);
                                  dtorph = record
(*requestor and acceptor channels*)   a,b:  SSAP;
(*selected functional units*)         sfu:   asr;
(*data token availability/assignment*) dk:   taa;
(*information flows
  [source channel] *)    S: array SSAP of nxqueue;
(*orderly release *)   rel: array SSAP of array
(*phase control:  *)      (req,ind) of boolean end
(*see data transfer phase *)
endtype

    (* temporal ordering *)

spec SS_BCS_X(a,b:SSAP) :=
              (session_connection(a,b)
              []session_connection(b,a)
              );SS_BCS_X(a,b)
(* Hereafter a and b respectively denote
  the requestor's and acceptor's channels. *)

    (* connection establishment phase

it is specified that a connection can be termin-
ated only after the connection establishment has
been requested. Termination processes starting in
this phase are specified after those of the
normal course of action *)

session_connection(a,b:SSAP) :=
      Creq(a|x);(Cind(b,x);pending(a,b,x)
                []early_termination(a,b,x)
                )
(*the connection pending process starts after the
S_CONNECT indication has taken place, and trans-
forms into the confirming one after the S_CONNECT
response is given; the "gates weak disable" ope-
rator is introduced in appendix B. *)

pending(a,b:SSAP,x:crip) :=
 let g = any of (a,b) in
 (abort_termination(g,remote(g)) >g|<
   (Crsp(b|y);
     if   (H in x.sr and H in y.sr)
     -->  confirming(a,b,x,y,y.ita,*,f)
     else confirming(a,b,x,y,na,*,f) fi
 ) )

(* in the confirming process the acceptor is
already either in the data transfer phase or dis-
connected, whereas the requestor is waiting for
the connection confirm; sfu: selected functional
units; r becomes true when the acceptor requests
the orderly release before the connection confirm
has taken place *)

confirming(a,b:SSAP,x:crip,y:crcp,
          dk:taa,Sb:nxqueue,r:boolean) :=
if  y.ru.r=acc
--> let sfu=intersection of (x.sr,y.sr) in
    (Ccnf(a,y);data_transfer
                (a,b,sfu,dk,(*,Sb),((f,f),(r,f)))
    []D in sfu or (H in sfu and dk=acs)-->
          (Dreq(b|z);confirming
                      (a,b,x,y,dk,nin((D,z),Sb),r)
          []Rreq(b|z);(confirming
                (a,b,x,y,dk,nin((Ri,z),Sb),t)\u(b))
          )
    []X in sfu-->Xreq(b|z);confirming
                      (a,b,x,y,dk,xin((X,z),Sb),r)
    []H in sfu-->if   dk=acs
                 -->  TGreq(b,(da));confirming
                   (a,b,x,y,na,nin((T,(da)),Sb),r)
                 else TPreq(b|z);confirming
                   (a,b,x,y,dk,nin((P,z),Sb),r) fi
    )
else Ccnf(a,y) fi

    (* data transfer phase

both requestor and acceptor are in the data
transfer phase which, in this specification,
includes the (orderly) release phase; the latter
is controlled by the values of rel: rel[xxx,g]
becomes true when the primitive S_RELEASE xxx...
is executed at SSAP g. The function onside(g)
returns rqs,acs for g = a,b respectively. *)

data_transfer(ps:dtorph) :=
with ps do for all g in (a,b)
(
   (* S_DATA request or S_RELEASE request *)

 D in sfu or (H in sfu and dk=onside(g)) -->
     (Dreq(g|z);data_transfer
                 (ps where S[g]<-nin((D,z),S[g]))

      (* it can be easily verified that a release
      collision can happen only in duplex mode *)

     []not rel[ind,g]-->
       Rreq(g|z);
        if  rel[req,remote(g)]
        --> release_collision
                 (ps where S[g]<-nin((Ri,z),S[g]))
        else data_transfer
            (ps where S[g]<-nin((Ri,z),S[g]),
                                rel[req,g]<-t)\u(g)
        fi
     )
```

```
   (* S_DATA indication or S_RELEASE indication
                        or S_RELEASE confirm   *)

[]S[g]<> * --> with first(S[g]) do
  code=D -->Dind(remote(g),parms);data_transfer
                        (ps where S[g]<-rest(S[g]))
 []code=Ri-->Rind(remote(g),parms);data_transfer
        (ps where S[g]<- *,rel[ind,remote(g)]<-t)
 []code=Rc-->Rcnf(remote(g),parms)                od

   (* S_RELEASE response *)

[]rel[ind,g] --> Rrsp(g|z);data_transfer
              (ps where S[g]<-nin((Rc,z),S[g]),
                          S[remote(g)]<- * )\g

   (* S_EXPEDITED_DATA service *)

[]X in sfu-->
  (Xreq(g|z);data_transfer
                 (ps where S[g]<-xin((X,z),S[g]))
  []nox(S[g])>0-->
     Xind(g,firstx(S[g]).parms);data_transfer
                    (ps where S[g]<-restx(S[g]))
  )

   (*  S_TOKEN services (data token) *)

[]H in sfu-->
  (if   dk=onside(g)
   -->  TGreq(g,(da));data_transfer
      (ps where dk<-na, S[g]<-nin((T,(da)),S[g]))
   else TPreq(g|z);data_transfer
             (ps where S[g]<-nin((P,z),S[g]))  fi
  []S[g]<> * --> with first(S[g]) do
    code=T-->TGind(remote(g),parms);data_transfer
                 (ps where dk<-onside(remote(g),
                                 S[g]<-rest(S[g]))
   []code=P-->TPind(remote(g),parms);data_transfer
                 (ps where S[g]<-rest(S[g]))  od
  )
) od

   (* release collision

the release collision is specified by the arbitrary interleaving of the provider outputs.

The execution of the S_RELEASE indication at each SSAP enables the involved user to realize the occurrence of the collision.

Two solutions are then conceivable: either this event is considered as terminating the connection (at the SSAP where it occurred) or another termination request primitive is accepted by the provider; no problem is detected by the authors if this is an S_U_ABORT request, since it is clearly stated in the DIS that data conveyed by abort primitives can be lost; on the contrary a not allowable loss of data occurs if an S_RELEASE response is accepted (as the state tables of the DIS indicate) since their user data can never be delivered. *)

   (* release collision as specified
            by the DIS state tables *)

release_collision(ps:dtorph) :=
 with ps do for all g in (a,b) ||
 (data_transfer(ps)\remote(g)\u(g);Rrsp(g|z)) od

   (* release collision without
            S_RELEASE response *)

release_collision(ps:dtorph) :=
 with ps do data_transfer(ps)\u(a)\u(b) od

   (* immediately release processes *)
(*
three kinds of occurrences are represented by the following process:
- a "local rejection" of a connection request,
- a provider-initiated abort before S_CONNECT indication,
- a (requestor) user-initiated abort before S_CONNECT indication.
*)

early_termination(a,b:SSAP,x:crip) :=
     i[y:prf);Ccnf(a,x where ru <- (prj,y))
   []i[y:par);PAind(a,y);
           (skip
           []Cind(b,x);p_abort_pending(b,x,y)
           )
   []UAreq(a|y);
           (skip
           []Cind(b,x);u_abort_pending(b,x,y)
           )

p_abort_pending(a,b:SSAP,x:crip,y:par) :=
(i[y:par);PAind(b,y)[]PAind(b,y))><
 (Crsp(b|z);if  (H in x.sr and H in z.sr)
              --> confirming(a,b,x,z,z.ita,*,f)
          else confirming(a,b,x,z,na,*,f) fi\a
)

u_abort_pending(a,b:SSAP,x:crip,y:par) :=
(i[y:par);PAind(b,y)[]UAind(b,y))><
 (Crsp(b|z);if  (H in x.sr and H in z.sr)
              --> confirming(a,b,x,z,z.ita,*,f)
          else confirming(a,b,x,z,na,*,f) fi\a
)

(*
the following process represents the immediately release, either user- or provider- initiated, at any moment after the S_CONNECT indication has taken place.
*)

abort_termination(a,b:SSAP) :=
     UAreq(a|x);UAind(b,x)
   [] for all x in par []PAind(a,x);PAind(b,x)

   (* service primitives *)

(* S_CONNECT *)
(* request   *) Creq(g:SSAP|x:crip):= u(g)?(Ci,x)
(* indication*) Cind(g:SSAP,x:crip):= p(g)!(Ci,x)
(* response  *) Crsp(g:SSAP|x:crcp):= u(g)?(Cc,x)
(* confirm   *) Ccnf(g:SSAP,x:crcp):= p(g)!(Cc,x)
(* S_DATA    *)
(* request   *) Dreq(g:SSAP|x:SSDU):= u(g)?(D,x)
(* indication*) Dind(g:SSAP,x:SSDU):= p(g)!(D,x)
(* S_RELEASE *)
(* request   *) Rreq(g:SSAP|x:rud) := u(g)?(Ri,x)
(* indication*) Rind(g:SSAP,x:rud) := p(g)!(Ri,x)
(* response  *) Rrsp(g:SSAP|x:rrcp):= u(g)?(Rc,x)
(* confirm   *) Rcnf(g:SSAP,x:rrcp):= p(g)!(Rc,x)
(* S_U_ABORT *)
(* request   *) UAreq(g:SSAP|x:aud):= u(g)?(AU,x)
(* indication*) UAind(g:SSAP,x:aud):= p(g)!(AU,x)
```

```
(* S_P_ABORT *)
(* indication*) PAind(g:SSAP,x:par):= p(g)!(AP,x)
(* S_TOKEN_PLEASE *)
(* request   *) TPreq(g:SSAP|x:tpp):= u(g)?(P,x)
(* indication*) TPind(g:SSAP,x:tpp):= p(g)!(P,x)
(* S_TOKEN_GIVE *)
(* request   *) TGreq(g:SSAP,x:tk) := u(g)?(T,x)
(* indication*) TGind(g:SSAP,x:tk) := p(g)!(T,x)
(* S_EXPEDITED_DATA *)
(* request   *) Xreq(g:SSAP|x:XSSDU):= u(g)?(X,x)
(* indication*) Xind(g:SSAP,x:XSSDU):= p(g)!(X,x)
```

5. CONCLUSIONS

The secret aim of this paper consisted of showing how an event algebra can effectively be employed for the specification of rather complex abstract services.

The goal can be considered reached, in the opinion of the authors, at least for the specified subset. This effectiveness remains to be shown for more complex target applications like the whole SS or the Session Protocol.

Some pitfalls of the current draft definition of LOTOS have been compensated in this example by the definition of some additional operators which do not affect the kernel language semantics, while seeming to be very useful at various OSI layers for the purpose of their concise specification.

Work is underway to complete the LOTOS definition with function and abstract type specification facilities.

6. ACKNOWLEDGEMENT

The authors like to acknowledge the contributions of Chris Vissers, Ed Brinksma, Jan De Meer and Günter Karjoth, with whom they cooperate both in the COST11 bis project on Temporal Ordering Specification and in ISO/TC97/SC16/WG1/FDT/-Subgroup C.

7. REFERENCES

<B 83> E. Brinksma, An Algebraic Language for the Specification of the Temporal Order of Events in Services and Protocols, Proc. of the European Teleinformatics Conference, Varese, Italy, Oct.3-6, 1983, North-Holland (1983) pp.533-542

<BPW 81> M. Broy, P. Pepper, M. Wirsing, On Design Principles for Programming Languages: An Algebraic Approach, in: De Bakker, Van Vliet (eds), Algorithmic Languages, North-Holland (1981)

<CFMS 84> V. Carchiolo, A. Faro, F. Minissale, G. Scollo, Some topics in the design of the specification language LOTOS, to appear.

<ISO 83a> ISO, Information Processing Systems, Open Systems Interconnection, Draft Tutorial Document, Temporal Ordering Specification Language, ISO/TC97/SC16/ /WG1/N 157, August 12, 1983

<ISO 83b> ISO, Information Processing Systems, Open Systems Interconnection, Basic Connection Oriented Session Service Definition,Draft International Standard DIS8326, 1983

<ISO 83c> ISO, Information Processing Systems, Open Systems Interconnection, Proposed DP for Services Conventions, ISO/TC97 /SC16/WG1/N 122 revised, October 1983.

<K 83> G. Karjoth, BDL,a Behaviour Description Language,IFIP WG6.1,Third International Workshop on Protocol Specification, Testing and Verification, Zurich May 1983, (North Holland).

<M 80> R. Milner, A Calculus of Communicating Systems, LNCS 92, Springer - Verlag, Berlin (1980)

<V 83> C. A. Vissers, Architectural Requirements for the Temporal Ordering Specification of Distributed Systems, Proc. of European Teleinformatics Conference, Varese, Italy, Oct. 3-6, 1983, North--Holland (1983), pp.79-95

NOTES

([1]) A type specification for e is not needed since e is a defined value at the time of occurrence of the output event; however, it might be useful, in some cases, to allow for a concise notation of internal nondeterminism in the output value generation: this could be expressed by a typed notation, where the type identifier would denote the range of the values that the output event may transfer.

([2]) A distinguished feature of the FU's definition is that their I/O dictionaries (in terms of parameterized service primitives) are disjoint. Nevertheless the high number of interdependencies among the FU's makes unpractical a model of the service which would describe it as the parallel composition of processes representing the FU's.

([3]) There are some restrictions concerning the selection of related FU's: for instance, the exception reporting FU can be selected only if the half-duplex FU is included.

([4]) Clause 10.12.1 of the DIS, on "extended control parameter", states that " the extended control parameter allows the SS user to make use of the resynchronize, abort and activity (discard and interrupt) services, when normal flow is congested. Note - when the expedited functional unit has been selected the extended control QOS is always provided to the SS user". The "always" adverb in the note should be referred to the services present in the agreed subset.

APPENDIX A: KERNEL LOTOS

This appendix shows the operators and other features of the kernel LOTOS now being developed by Subgroup C of ISO/TC97/SC16/WG1/Ad-Hoc group on FDT's.

LOTOS operators:

- sequence operator denoted by ";": if p and q are processes, also (p;q) is a process which initially behaves like p and, when p terminates, then it behaves like q.

- choice operator denoted by "[]": if p and q are processes, also (p [] q) is a process which behaves either like p or like q. The choice may be extended to an indexed set of processes by means of a construct of the form:

for all <index> in <type> [] p(<index>)

- composition operator denoted by |B| where B is a set of gates; if p and q are processes, then (p |B| q) is a composed process which transforms complementary offers at gates in B into internal events, whereas arbitrary interleaving is allowed of events of p and q at gates not in B.

- disable operator denoted by >< ; if p and q are processes (p >< q) is a process which behaves like q until the occurrence of the first event of p; after this occurrence the process may only continue with the rest of p.

- restriction operator denoted by "\B" where B is a set of gates; if p is a process, p\B is a process which behaves like p without derivations starting with an event at a gate of B.

Other language features are:

- guarded commands in the form:

e --> p

which establishes that process p may be executed only if expression e is true. Mutually exclusive guards can be expressed in the form:

if e --> p else q fi

- behaviour identifiers in the form :

alfa := p

which establishes a way to start the process p simply by using in the language expressions the identifier alfa. The identifier can be also parameterized in such a way as to give rise to a (possibly infinite) number of identifiers simultaneously.

- recursion that means that we may use the behaviour identifier itself in defining the language expressions.

- substitution either in the form:

let <let_substitution_list> in (<LOTOS expression>)

or w.r.t. an identifier of a parameter list:

<identifier> where <where_substitution_list>

The Pascal with clause is adopted.

Finally we note that a high degree of flexibility in modularising our specifications can be obtained by using the notions of export values list and import values list.

The export list of a process p is a set of parameters having a defined value for every termination of p.

The input list of a process p is the set of parameters which must be defined before the execution of p.

These lists are represented in LOTOS by using the following notation :

$$p\ (i_1,\ldots,i_N \mid e_1,\ldots,e_M)$$

where i_i is a generic imported value and e_i is a generic exported value. The symbol "[" is used instead of "(|" when the import list is empty.

Process p may be the internal "unobservable" event i, in which case the internal value generation is represented by the export list.

APPENDIX B: LOTOS EXTENSIONS

This appendix presents some of the new operators proposed in <CFMS 84> to be added to the kernel LOTOS in order to obtain more readable and concise specifications. In the sequel A denotes a set of gates. When A consists of one gate, A={a}, a will be used instead of A.

1. ">|<" :"weak disable"; if p and q are processes, then p>|<q is also a process which behaves like p||q until the last event of either p or q takes place; this event actually terminates the resulting process.

2. ">$_A$|<":"gates weak disable"; if p and q are processes then (p>$_A$|<q) is also a process which behaves like q until the first event of p takes place; after this event it behaves like p'>|<(q'\A) where p' and q' are the processes in which p and q respectively transformed.