Kenneth J. Turner and Marten van Sinderen. LOTOS specification style for OSI. In Ed Brinksma, Tommaso Bolognesi, and Christopher A. Vissers, editors, Proc. 3rd LotoSphere Workshop, pages 5/1-22, Pisa, September 1992. CNUCE.

LOTOS Specification Style for OSI

K. J. Turner
University of Stirling

M. van Sinderen
University of Twente

Abstract

The architecture of OSI is used to derive guidelines for writing LOTOS specifications of distributed systems. In particular, the architectural concepts that underlie service and protocol designs are examined in detail. For each of these concepts a representation in LOTOS is given. Examples are provided of how the LOTOS representations of the concepts are used in the construction of LOTOS specifications of service and protocol designs. The approach described in this paper is motivated by the need to produce distributed system specifications in a more consistent and productive fashion.

1 Introduction

Design and specification are related but distinct notions. A **design** is an abstraction of a technical object of concern. This paper deals with design **specifications** — representations of a design using a specification language.

It is common experience that one of the most difficult and critical aspects of design specification is the choice of **specification structure**. The structure of a specification depends on how architectural concepts that underlie the design are represented, and how these representations are combined to form the specification.

The design goals that determine the choice of architectural concepts and the structure, or architecture, of the design should also be considered when deciding on the specification structure. Ignoring them may lead to specifying unintended design decisions with consequences for derived implementations [VF92]. Examples of design goals are:

- separation of concerns
- modularity, information hiding and encapsulation
- comprehension, enhancement, maintenance and sub-division of work
- mapping onto implementation elements and structures.

This paper presents guidelines for writing LOTOS specifications of service and protocol designs, emphasising the aspect of specification structure. The approach taken is as follows:

- The architectural concepts related to services and protocols are examined, and for each of the concepts a representation in LOTOS is given.
- The generic architectural features of services and protocols are used to derive corresponding specification structures incorporating the LOTOS representations of the relevant architectural concepts.

Services and protocols are particular architectural concepts which are commonly used in the design and implementation of communication systems, and which play an important role in the architecture of OSI (Open Systems Interconnection). Services and protocols arise, however, in any layered architecture, and may also be useful in the design and implementation of systems in other application areas, such as Operating Systems or Computer Integrated Manufacturing.

Basically, a **service** design is a black box model of a distributed system that allows reasoning about user requirements for a system without being concerned about the construction of the system. This abstraction provides the starting point for a corresponding **protocol** design which models the system in terms of distributed functions that cooperatively fulfil the requirements of the service. In a layered approach, protocols are developed in a number of steps. Each step is concerned with the identification of a underlying service and a layer of functions that cooperate through this service to support the required service.

LOTOS is one of the standardised FDTs (Formal Description Techniques). It has been widely used for specifying OSI services and protocols, such as:

Application Layer:

- **DTP (Distributed Transaction Processing):** protocol [WHR90], discussion [SW91]
- **CCR (Commitment, Concurrency and Recovery):** service [Sad90], protocol [JC90]
- **ROSE (Remote Operations Service Element):** service [FA89]

Session Layer: service [ISO89a], protocol [ISO89b], discussion [SA89]

Transport Layer: service [ISO90a], protocol [ISO90b], discussion [LS89]

Network Layer: service [Tur89], protocol [Fer89].

Experience from the application of LOTOS to OSI indicates that it is relatively straightforward to represent architectural concepts and architectures in LOTOS. This is also shown, at a more fundamental level, in [Tur88]. Possible representations of OSI concepts in the standard FDTs have been developed by ISO with the purpose of establishing a precise and unambiguous meaning of the concepts [ISO89c]. In addition, ISO and CCITT jointly developed guidelines for the application of standard FDTs in order to further stimulate and facilitate the production of specifications of OSI standards [ISO91]. The use of a few appropriate styles for service and protocol specification in LOTOS is discussed in [VSS88], motivated by their support of applicable design goals.

There is nothing absolute about the LOTOS representations proposed in this paper. In all cases there are reasonable alternatives. However, the proposed representations have been developed on the basis of wide specification experience, and are consistent with identified specification styles. Following them will encourage consistency in the specification of layered systems such as OSI. By taking advantage of the work that has gone into their definition, a specifier will also be able to work much more productively. Predefined representations of architectural concepts can be extended or specialised to fit the requirements of specific designs, or just the style of specification can be adopted in these cases. Note, however, that LOTOS does not support the reuse (in a formal sense) of generic process definitions.

The stimulus for the work reported in this paper came from the need of the LOTOSPHERE project to specify and develop realistic distributed applications. These included a Distributed Transaction Processing application supported by OSI Application Layer standards, and a Mini-Mail application. The LOTOSPHERE development teams needed guidance on how to specify these applications, particularly their architectural features. The work reported here was integrated into the general design methods evolved by the project.

The remainder of the paper is structured round the discussion of a LOTOS specification style for OSI. Section 2 is devoted to OSI service concepts, and section 3 to OSI protocol concepts.

2 Specification Elements for OSI Services

2.1 General Service Structure

A service design models the interactions between a set of service users and a service provider [VL86]. At this level of abstraction, interactions are considered as shared activities, with no explicit division of responsibility between the service users and the service provider. A service design also abstracts from the (internal) distribution of the service provider. This is depicted in figure 1.

Elementary interactions between a user and the provider are termed **service primitives** which are taken as the building blocks for service definitions [ISO92]. Service primitives occur at abstract interfaces called SAPs (**service access points**), each of which is distinguished by means of a unique **address**. Service users are distinguished by means of a unique **title**.

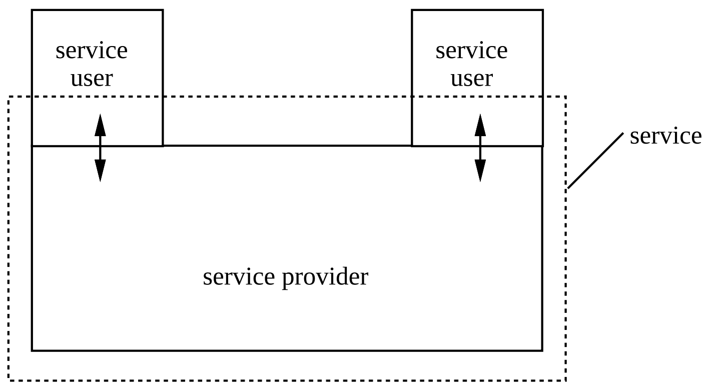

Figure 1: Service Design

Service primitives have one or more associated parameters that model the exchange of information. A parameter of many service primitives is an SDU (**service data unit**) — user data that is transferred transparently by the service provider.

Given the nature of a service, it is appropriate to structure a service design in terms of sets of constraints that apply to (groups of) service primitives. The style of specification that best suits this objective is the **constraint-oriented style** [VSS88]. In the following, this style is used to illustrate the specification of two well-known **service types**: connection-less and connection-oriented.

A service type is a characterisation of the common service requirements of a particular class of users. A CL (**connection-less**) service satisfies the need of users to transfer SDUs independently of each other, without the overhead of agreeing the quality of transfer in advance. A number of variants exist of the connection-less service type. The following sections centre on the simplest connection-less service, often referred to as the unconfirmed or datagram service. Typical of an unconfirmed connection-less service is that calling (sending) users are not informed of the success or failure of the data transfers requested. Although a connection-less service does not require that the sequence of SDUs is preserved between a particular pair of users, some providers may in fact do so.

A CO (**connection-oriented**) service satisfies the requirement of users to transfer SDUs such that for each service invocation the transfer is sequenced and performed under quality conditions which are agreed in advance of data transfer. Three **phases**, or **functional elements**, can be distinguished in a connection-oriented service invocation as a consequence of this requirement. The **connect element** element is concerned with the agreement of quality conditions that will apply to later phases. The **data** element is concerned with the transfer of data. It consists of transferring SDUs in either direction. The **disconnect** element is used to mark the end of the connection-oriented service invocation.

In the case of a connection-oriented service, distinct groups of service primitives may occur at the same SAP. Each of these groups is related to a separate service invocation, or **connection**, and locally distinguished by means of a CEP identifier (**connection endpoint identifier**). In the case of a simple connection-less service, each occurrence of a service primitive is independent so such a concept is not needed. However, intermediate types of service may support (short) groups of service primitive occurrences that may be overlapped with other such groups (e.g. an acknowledged connection-less service). For this reason the concepts of **association** and AEP identifier (**association endpoint identifier**) are introduced as generalisations of connection and CEP identifier respectively. Although these concepts are not recognised in the architecture of OSI, they are used in this paper for the sake of generality. Figure 2 depicts their use.

2.2 Service User, Service Provider, Service Boundary

From a LOTOS viewpoint, the service provider is the system to be specified and the service users form the environment of the system. Service primitives are therefore specified as LOTOS event offers, actually representing the provider view of service primitives.

OSI is indefinite about the nature of service primitives, e.g. whether they occur synchronously, atomically

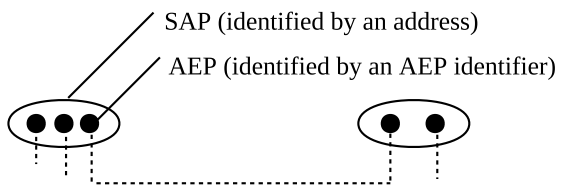
SAP (identified by an address)
AEP (identified by an AEP identifier)
association

Figure 2: Association (Connection) and AEP Identifier (CEP Identifier)

or instantaneously. Treating the occurrence of service primitives as LOTOS events gives them these three properties. However, this is purely a level of abstraction which is appropriate in a service design. In a more refined design (including a protocol design) it is possible to model the occurrence of service primitives as asynchronous, interruptible and spread out in time.

Although service primitive parameters could appear as corresponding parameters of a LOTOS event, this could lead to lengthy events. More seriously, since service primitives may differ in the number of their parameters this approach could lead to a variety of event structures. It is therefore better to collect all service primitive parameters into one composite structure — in fact, a record.

The LOTOS representation of a service primitive occurrence at a AEP has a common format:

```
service_gate ! address ! association_endpoint_identifier ! service_primitive_parameters
```

For services that do not require AEP identifiers, a simpler event structure is used:

```
service_gate ! address ! service_primitive_parameters
```

SAPs and AEPs exist only to distinguish sequences of service primitive occurrences. They therefore have a *behavioural* rather than a *structural* interpretation. In LOTOS, a single gate is used for communication at a **service boundary**, i.e. the collection of SAPs. The occurrence of a service primitive at a particular SAP is distinguished by means of the primary parameter of the corresponding LOTOS event. If needed, the identification of an AEP within the SAP is represented by the secondary parameter.

2.3 Title, Address, Association Endpoint Identifier

Titles, addresses and AEP identifiers are simply sets of distinct labels. Titles and addresses are globally unique within the scope of OSI, whereas AEP identifiers are unique only within the scope of a SAP. Since titles are associated with service users alone, they are not required in a service (although they may be exchanged as parameters of service primitives). All these kinds of identifiers may have structure for convenience in allocation or routing; however this is not relevant at an abstract level. The identifiers are simply specified in LOTOS as distinct values in a sort. Since an identifier set may be infinite, it is constructed inductively from a base value and an operation to yield another value from a given one.

```
type ADDR is Boolean                                                  (* address *)
  sorts Addr opns
    BaseAddr      :                   -> Addr                         (* base address *)
    AnotherAddr   : Addr              -> Addr                         (* yield another address *)
    _eq_, _ne_    : Addr, Addr        -> Bool                         (* (in)equality *)
  eqns
    forall AddrA, AddrB : Addr
      ofsort Bool
        BaseAddr                      eq BaseAddr                     = true;
        AnotherAddr (AddrA)           eq BaseAddr                     = false;
        BaseAddr                      eq AnotherAddr (AddrB)          = false;
        AnotherAddr (AddrA)           eq AnotherAddr (AddrB)          = AddrA eq
        AddrB; AddrA ne AddrB                                         = not (AddrA eq AddrB);
endtype (* ADDR *)

type IDENT is ADDR renamedby                                          (* association endpoint identifier *)
```

```
  sortnames Ident for Addr
  opnnames
    BaseIdent             for BaseAddr
    AnotherIdent          for AnotherAddr
endtype (* IDENT *)
```

2.4 Originator, Reason

Since a service primitive indication or confirm may occur as a result of action by the remote service user or by the service provider, it may be necessary to give the originator of the action as a parameter. In some services, particularly those that are implemented by message store-and-forward, the originator may be a user other than the called or calling user. It may also be appropriate for a service primitive to carry a reason for its occurrence. Typically this is true of a reset or disconnect indication primitive, where the reason is a request at the other service user or is some error code (unknown address, unreachable address, etc.).

```
type ORIG is Boolean, BasicNaturalNumber                      (* originator *)
  sorts Orig opns
    User, Prov, Other     :                 -> Orig           (* possible originators *)
    Ord                   : Orig            -> Nat            (* ordinal number *)
    _eq_, _ne_            : Orig, Orig      -> Bool           (* (in)equality *)
  eqns
    forall OrigA, OrigB : Orig
      ofsort Nat
        Ord (User)        = 0;
        Ord (Prov)        = Succ (Ord (User));
        Ord (Other)       = Succ (Ord (Prov));
      ofsort Bool
        OrigA eq OrigB    = Ord (OrigA) eq Ord (OrigB);
        OrigA ne OrigB    = not (OrigA eq OrigB);
endtype (* ORIG *)

type REAS is Boolean, BasicNaturalNumber                      (* reason *)
  sorts Reas opns
    User, Error, ...      :                 -> Reas           (* possible reasons *)
    Ord                   : Reas            -> Nat            (* ordinal number *)
    _eq_, _ne_            : Reas, Reas      -> Bool           (* (in)equality *)
  eqns
    forall ReasA, ReasB : Reas
      ofsort Nat
        Ord (User)        = 0;
        Ord (Error)       = Succ (Ord (User));
        ...
      ofsort Bool
        ReasA eq ReasB    = Ord (ReasA) eq Ord (ReasB);
        ReasA ne ReasB    = not (ReasA eq ReasB);
endtype (* REAS *)
```

2.5 Option

A service may be characterised by optional functional and qualitative aspects whose use is negotiated when an association is set up. Functional aspects are all-or-nothing functions (e.g. use of receipt confirmation or not), while qualitative aspects have a range of values (e.g. throughput and transit delay). It is usual to group QoS (**quality of service**) options that affect the quality rather than the functionality of the service. In general, all options can be regarded as drawn from an ordered set. The ordering defines what a ‘worse’ value of the parameter means, and depends on the option. Functional aspects such as expedited data selection and receipt confirmation selection would be grouped as functional options. Qualitative aspects such as throughput and transit delay would be grouped as quality options.

```
type OPT is Boolean                                                    (* option *)
  formalsorts Opt
  formalopns
    _eq_, _lt_      : Opt, Opt            -> Bool                      (* equality, 'worse than' *)
  formaleqns
    forall Opt ,OptA, OptB, OptC : Opt
      ofsort Bool
        Opt eq Opt                    = true;
        OptA eq OptB                  = OptB eq OptA;
        OptA eq OptB, OptB eq OptC =>
          OptA eq OptC    = true;
        Opt lt Opt                    = false;
        OptA lt OptB =>
          OptB lt OptA    = false
  opns
    _ne_, _le_, _gt_, _ge_ : Opt, Opt -> Bool                          (* orderings *)
  eqns
    forall Opt, OptA, OptB, OptC : Opt
      ofsort Bool
        OptA ne OptB      = not (OptA eq OptB);
        OptA le OptB      = (OptA lt OptB) or (OptA eq OptB);
        OptA gt OptB      = not (OptA le OptB);
        OptA ge OptB      = not (OptA lt OptB);
endtype (* OPT *)

type FUNPAR is Boolean                                                 (* functional parameters *)
  sorts FunPar
  opns
    ExpSel, ConfSel         : Bool                    -> FunPar
                                                                       (* expedited, confirmation *)
    _eq_, _lt_              : FunPar, FunPar          -> Bool  (* (in)equality, 'worse than' *)
  eqns
    forall bool1, bool2 : Bool
      ofsort Bool
        ExpSel (bool1) eq ExpSel (bool2)          = bool1 eq bool2;
        ConfSel (bool1) eq ConfSel (bool2)        = bool1 eq bool2;
        ExpSel (bool1) lt ExpSel (bool2)          = not (bool1 implies bool2);
        ConfSel (bool1) lt ConfSel (bool2)        = not (bool1 implies bool2);
endtype (* FUNPAR *)

type FUNOPT is OPT actualizedby FUNPAR using                           (* functional options *)
  sortnames
    FunPar for Opt
endtype (* FUNOPT *)

type QOSPAR is NaturalNumber                                           (* quality parameters *)
  sorts QosPar
  opns
    ThrPut, Delay           : Nat              -> QosPar               (* throughput, transit delay *)
    Ord                     : QosPar           -> Nat                  (* ordinal number *)
    _eq_, _lt_              : QosPar, QosPar   -> Bool                 (* (in)equality, 'worse than' *)
  eqns
    forall Nat, NatA, NatB : Nat, QosParA, QosParB : QosPar
      ofsort Nat
        Ord (ThrPut (Nat))                = 0;
        Ord (Delay (Nat))                 = Succ (Ord (ThrPut (Nat)));
      ofsort Bool
        Ord (QosParA) ne Ord (QosParB) =>
```

```
            QosParA eq QosParB                  = false;
          ThrPut (NatA) eq ThrPut (NatB)        = NatA eq NatB;
          Delay (NatA) eq Delay (NatB)          = NatA eq NatB;
          Ord (QosParA) ne Ord (QosParB) =>
            QosParA lt QosParB                  = false;
          ThrPut (NatA) lt ThrPut (NatB)        = NatA lt NatB;
          Delay (NatA) lt Delay (NatB)          = NatA gt NatB;
endtype (* QOSPAR *)

type QOSOPT is OPT actualizedby QOSPAR using            (* quality options *)
  sortnames
    QoSPar for Opt
endtype (* QOSOPT *)
```

The type for a set of QoS parameters, *QOSSET*, may be specified using *QOSOPT* and the library *Set* in the obvious way.

2.6 Service Data Unit

A SDU is a parameter of many types of service primitives. Since the service provider does not operate on SDUs, it is sufficient to have only the constructor operations for a list of data values; the standard library type *OctetString* provides what is needed:

```
type DATA is OctetString renamedby
  sortnames Data for OctetString
endtype (* DATA *)
```

2.7 Service Primitive, Service Primitive Parameter

The LOTOS representations of the types (names) and the parameters of service primitives of a particular service can be collected in a single type definition. For a connection-less service, this might be done as follows.

```
type PRIM is Boolean, BasicNaturalNumber, DATA, ADDR    (* CL service primitive *)
  sorts Prim
  opns
    DatReq, DatInd        : Addr, Data    -> Prim       (* data request/indication *)
    IsDatReq, IsDatInd    : Prim          -> Bool       (* recognisers *)
    IsReq, IsInd          : Prim          -> Bool       (* recognisers *)
    Ord                   : Prim          -> Nat        (* ordinal number *)
    _eq_, _ne_            : Prim, Prim    -> Bool       (* (in)equality *)
  eqns
    forall Prim, PrimA, PrimB : Prim, Addr : Addr, Data, DataA, DataB : Data
      ofsort Nat
        Ord (DatReq (Addr, Data))           = 0;
        Ord (DatInd (Addr, Data))           = Succ (Ord (DatReq (Addr, Data)));
      ofsort Bool
        IsDatReq (Prim)                     = Ord (Prim) eq Ord (DatReq (Addr, Data));
        IsDatInd (Prim)                     = Ord (Prim) eq Ord (DatInd (Addr, Data));
        IsReq (Prim)                        = IsDatReq (Prim);
        IsInd (Prim)                        = IsDatInd (Prim);
        DatReq (AddrA, DataA) eq DatReq (AddrB, DataB) =
          (AddrA eq AddrB) and (DataA eq DataB);
        DatInd (AddrA, DataA) eq DatInd (AddrB, DataB) =
          (AddrA eq AddrB) and (DataA eq DataB);
        PrimA ne PrimB = not (PrimA eq PrimB);
endtype (* PRIM *)
```

A LOTOS representation of the service primitives and parameters of a connection-oriented service is given below. For simplicity, only a few service primitive types are considered and some parameters that would normally be present (e.g. reason and QoS) are omitted.

```
type PRIM is Boolean, BasicNaturalNumber, DATA, ADDR           (* CO service primitive *)
  sorts Prim
  opns
    ConReq                    : Addr, Addr       -> Prim  (* connect request *)
    ...
    DisInd                    :                  -> Prim  (* disconnect indication *)
    IsConReq, ..., IsDisInd   : Prim             -> Bool  (* recognisers *)
    IsReq, IsInd              : Prim             -> Bool  (* recognisers *)
    Ord                       : Prim             -> Nat   (* ordinal number *)
    _eq_, _ne_                : Prim, Prim       -> Bool  (* (in)equality *)
  eqns
    forall
     Prim, PrimA, PrimB : Prim, Data, DataA, DataB : Data,
     Addr1, Addr1A, Addr1B, Addr2, Addr2A, Addr2B : Addr
      ofsort Nat
        Ord (ConReq (Addr1, Addr2))         = 0;
        ...
        Ord (DisInd)                        = Succ (Ord (DisReq));
      ofsort Bool
        IsConReq (Prim)                     = Ord (Prim) eq Ord (ConReq (Addr1, Addr2));
        ...
        IsDisInd (Prim)                     = Ord (Prim) eq Ord (DisInd);
        IsReq (Prim)                        = IsConReq (Prim) or ...;
        IsInd (Prim)                        = ... or IsDisInd (Prim);
        Ord (PrimA) ne Ord (PrimB) =>
          PrimA eq PrimB                    = false;
        ConReq (Addr1A, Addr2A) eq ConReq (Addr1B, Addr2B) =
          (Addr1A eq Addr1B) and (Addr2A eq Addr2B);
        ...
        DisInd eq DisInd                    = true;
        PrimA ne PrimB                      = not (PrimA eq PrimB);
endtype (* PRIM *)
```

Selector operations to access service primitive parameters can be entirely dispensed with if parameters are always accessed constructively rather than destructively, i.e. by assembling the fields required to build the desired record. Suppose, for example, that a connection request is constructed by the operation *ConReq* from source and destination address parameters. If operations to select these fields were introduced, they would have to be defined for *all* primitives. This could lead to many error equations for primitives that would otherwise not have these fields. It is therefore better to dispense with the selector operations and to access the fields constructively. As an example of accessing a record constructively, the fields of a given connection request primitive *CR* might be accessed by:

```
choice Src, Dst : Addr []
  [ConReq (Src, Dst) eq CR] ->
    (* specification referring to Src and Dst *)
```

2.8 Address-Identifier Pair

An AEP acts as a finer structure within a SAP. An AEP identifier is therefore unique only within the scope of a SAP address. In service specifications, it is convenient to deal with the identity of AEPs at a global level. This can be done by means of **pairs**, where each pair consists of a SAP address and an AEP identifier:

```
type PAIR is ADDR, IDENT                                       (* address-identifier pair *)
  sorts Pair
  opns
    Pair         : Addr, Ident     -> Pair                     (* address-identifier *)
    Addr         : Pair            -> Addr                     (* address selector *)
    Ident        : Pair            -> Ident                    (* identifier selector *)
    _eq_, _ne_   : Pair, Pair      -> Bool                     (* (in)equality *)
  eqns
    forall
     Addr, AddrA, AddrB : Addr, Ident, IdentA, IdentB : Ident,
     PairA, PairB : Pair
      ofsort Addr
        Addr (Pair (Addr, Ident))              = Addr;
      ofsort Ident
        Ident (Pair (Addr, Ident))             = Ident;
      ofsort Bool
        PairA eq PairB                         =
          (Addr (PairA) eq Addr (PairB)) and (Ident (PairA) eq Ident (PairB));
        PairA ne PairB                         = not (PairA eq PairB);
endtype (* PAIR *)
```

The type for a set of pairs, *PAIRSET*, may be specified using *PAIR* and the library *Set* in the obvious way.

2.9 Overall Service Provider Constraints

At any time the service provider may support zero, one or more groups of interworking service users. In a service design, each supported group corresponds to an association. (This is a connection in the case of a connection-oriented service, also called a **dialogue** or **session** in some standards.) The overall behaviour of associations can be factored into a number of different concerns. In the case of a service with no need for AEP identifiers, two simple concerns apply: dealing with transfer of isolated SDUs, and refusing to accept new data when the service provider is congested. In the case of a service that needs AEP identifiers, more complex concerns have to be taken into account: dealing with associations, refusing to accept new data on (some) associations when the service provider is congested, refusing to initiate a new association when its endpoints are not uniquely identified (with pairs), and refusing to initiate a new association when there are insufficient resources.

These concerns act as individual but conjoined constraints on the behaviour of the service provider, and so lead to a constraint-oriented style at the top level of the service specification. Such a style is appropriate for giving an abstract, high-level specification as required for a service design. The parallel constraints normally synchronise on each event, but one constraint may (temporarily) forbid an event by not providing a matching event offer.

A LOTOS representation of such constraints for a connection-less service is:

```
behaviour
  Trans [cl] || DataRefusals [cl]

where

process Trans [cl] : noexit :=                              (* CL data transfer *)
  Tran [cl] ||| Trans [cl]
endproc (* Trans *)

process DataRefusals [cl] : noexit :=                       (* CL data congestion *)
  choice Pair : Pair, DataPairs : PairSet []
    cl ! Addr (Pair) ! Ident (Pair) ? Prim : Prim
      [(IsData (Prim) and IsReq (Prim)) Implies (Pair IsIn DataPairs)];
    DataRefusals [cl]
  []
    i;                                                      (* revise acceptable pairs *)
    DataRefusals [cl]
endproc (* DataRefusals *)
```

A LOTOS representation of such constraints for a connection-oriented service uses a similar approach with refusal processes. The *PairRefusals* process in the following is parameterised by the set of pairs already in use.

```
behaviour
```

```
   Conns [co] || DataRefusals [co] || PairRefusals [co] ({}) || ConnRefusals [co]

where

process Conns [co] : noexit :=                              (* CO connections *)
   Conn [co] ||| Conns [co]
endproc (* Conns *)

process DataRefusals [co] : noexit :=                       (* CO data congestion *)
   ...

process PairRefusals [co] (Used : PairSet) : noexit :=      (* CO pair uniqueness *)
   ...

process ConnRefusals [co] : noexit :=                       (* CO connection congestion *)
   ...
```

2.10 Association

For a service that does not need AEP identifiers, the transfer of SDUs (as well as information conveyed in other service primitive parameters) can be dealt with immediately in the service specification. Where AEP identifiers are needed, it is useful to separate different associations in the service specification, and to deal with each of them as independent constraints. Many different ways could be imagined to represent the separation of associations, e.g. pre-allocation, allocation from a central pool of free resources, or allocation from distributed pools.

Abstractly speaking, the resources to support associations are dynamically bound when they are required. Initiation and termination may be implicit (e.g. as in a connection-less service) as well as explicit. In the latter case, it is therefore natural to model the behaviour of an association in a number of phases (see section 2.1).

At one SAP or at one AEP of an association there are **local constraints** on the types of service primitives which may occur and the order in which they may occur. There may also be constraints on the values of service primitive parameters, and there may even be temporal constraints on these (e.g. some disconnection reasons may be valid only when refusing a connection). If an association involves just one user and the provider, the local constraints will fully define it. More normally an association involves the service provider as an intermediary between two users (**point-to-point**). In general, two or more users may be associated (**multi-point**). If a service is symmetrical (**peer-to-peer**) then the local constraints of an association will be identical at each user. In some cases, however, the service is not symmetrical, so the local constraints at some users of an association will be different from those at others (e.g. **primary-secondary**, **master-slave**, **client-server**). Local constraints at different users of an association are independent, and are therefore interleaved in the service specification. If two or more users are involved in an association, service primitive occurrences need to be related on an end-to-end basis. These **remote constraints** may simply relate the request/response by one user to the indication/confirm at the other. In the multi-user case, the provider may be required to broadcast a request by one user to all corresponding users.

The local constraints deal with concerns that can be separated from the concerns dealt with by the remote constraints. This is reflected in a service specification by the synchronised composition of these types of constraints. (Their synchronisation follows from the fact that they apply to occurrences of the same service primitives — LOTOS events.)

A LOTOS representation of association constraints for a connection-less service is:

```
process Tran [cl] : noexit :=                               (* CL data transfer *)
   choice Dst : Addr, Data : Data []
      cl ? Src : Addr ! DatReq (Dst, Data) [Src ne Dst];
      (
         cl ! Dst ! DatInd (Src, Data); stop                (* deliver message *)
      []
         i; stop                                            (* lose message *)
      )
   endproc (* Tran *)
```

A LOTOS representation of association constraints for a connection-oriented service is:

```
process Conn [co] : noexit :=                               (* CO connection *)
  choice PairA, PairB : Pair []
    (ConnLoc [co] (PairA) ||| ConnLoc [co] (PairB))
  ||
    (ConnRem [co] (PairA, PairB, <>) ||| ConnRem [co] (PairB, PairA, <>))

  where

  process ConnLoc [co] (PairX : Pair) : noexit :=           (* CO local constraints *)
    ...
  process ConnRem [co] (PairX, PairY : Pair, Med1 : Med) : noexit :=
    ...                                                     (* CO remote constraints *)

endproc (* Conn *)
```

2.11 Service Object, Service Medium

Modelling the relation between service primitive occurrences on an end-to-end basis should abstract away from how the service provider implements this. In some OSI service standards a queue model is used for this purpose. This model allows the addition to and removal of **service objects** from the queue as well as some additional operations. The addition of a service object corresponds to the exchange of information from user to provider in a **request** or **response** primitive. Similarly, the removal of a service object corresponds to the exchange of information from provider to user in an **indication** or **confirm** primitive. Additional operations are needed to model unreliability (e.g. loss of data), priorities (e.g. expedited data overtaking normal data), and special service facilities (e.g. a reset cancelling requests/responses in transit).

A service specification can include a type definition to represent the queue model. This type is then used in the specification of the remote constraints for associations. The term **medium** is used instead of queue in order to avoid a possible association with normal queue behaviour. Also, the representation does not include operations for explicitly promoting service objects through the medium since this would be too implementation-oriented. As in section 2.7, only some service primitives have been dealt with.

```
type OBJ is PRIM                                            (* medium object *)
  sorts Obj
  opns
    Req                          : Prim          -> Obj     (* primitive constructor *)
    Ind                          : Obj           -> Prim    (* object constructor *)
    IsConMsg, ..., IsDisMsg      : Obj           -> Bool    (* recognisers *)
    _eq_, _ne_                   : Obj, Obj      -> Bool    (* (in)equality *)
  eqns
    forall ObjA, ObjB : Obj, Prim : Prim, Addr, Addr1, Addr2 : Addr, Data : Data
      ofsort Prim
        Ind (Req (ConReq (Addr1, Addr2))) = ConInd (Addr1, Addr2);
        ...
        Ind (Req (DisInd)) = DisInd;
      ofsort Bool
        IsConMsg (Req (Prim)) = IsConReq (Prim) or IsConInd (Prim);
        ...
        IsDisMsg (Req (Prim)) = IsDisReq (Prim) or IsDisInd (Prim);
        ObjA eq ObjB = Ind (ObjA) eq Ind (ObjB);
        ObjA ne ObjB = not (ObjA eq ObjB);
endtype (* OBJ *)

type MED is String actualizedby OBJ using                   (* medium *)
  sortnames
```

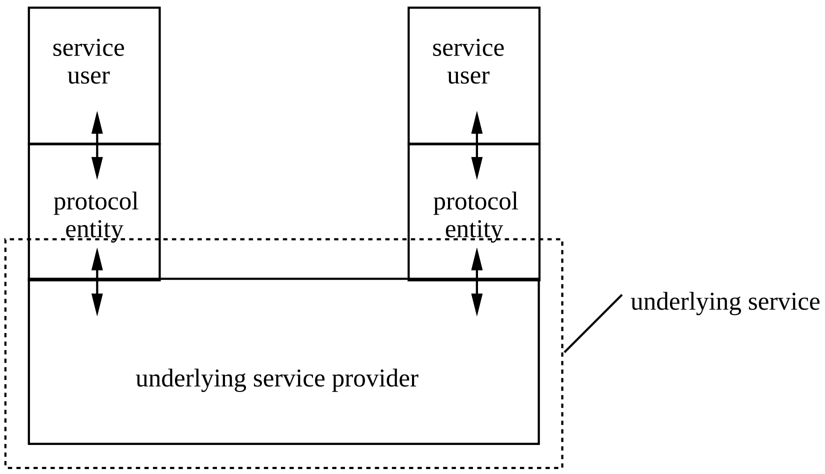

Figure 3: Protocol Design

```
      Med            for String
      Obj            for Element
      Bool           for FBool
endtype (* MED *)

type MEDOPS is MED                                                    (* medium operations *)
   sorts MedSet
   opns
      _overtakes_, _destroys_      : Obj, Obj        -> Bool          (* passes, removes *)
      _cancels_, _ignores_         : Obj, Obj        -> Bool          (* cancels, bypasses *)
      SetOf                        : Med             -> MedSet        (* medium set *)
      _&_                          : MedSet, Med     -> MedSet        (* prefix to medium set *)
      _&_                          : Med, MedSet     -> MedSet        (* append to medium set *)
      Reorders                     : Med             -> MedSet        (* medium reorderings *)
      Reorders                     : MedSet          -> MedSet        (* medium set reorderings *)
   eqns
      ...
endtype (* MEDOPS *)
```

3 Specification Elements for OSI Protocols

3.1 General Protocol Structure

A protocol design models the responsibility of a service provider in interactions with a set of service users, as well as an internal distribution of the service provider. A protocol is therefore a lower level design, as compared to the corresponding service [VL86].

The approach followed in protocol design is that of identifying a layer of distributed functions, or **protocol entities**, that cooperate via an **underlying service provider**. This is depicted in figure 3. The internal structure of the underlying service provider is of no concern to the designer if it is already implemented, or is deferred to the next stage of the design process. In the latter case, it is useful to start with the underlying service design before the explicit responsibility of protocol entities and the underlying service provider are established in the protocol design.

Protocol entities communicate through the exchange of PDUs (**protocol data units**). PDUs convey the information that is exchanged in service primitive parameters of the required service. In addition, they convey information that is internally generated by the protocol entities in order to guarantee that certain end-to-end conditions of the required service are satisfied (e.g. the error-free transfer of SDUs). PDUs in

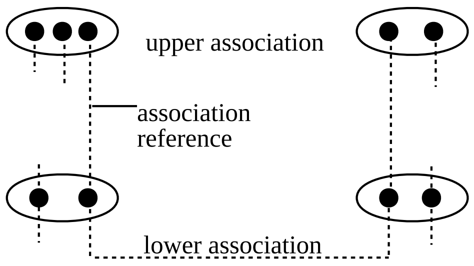

Figure 4: Association Reference

turn need to be mapped onto underlying service SDUs to achieve transparent transfer via the underlying service provider.

Protocol structuring can be done in terms of sets of constraints imposed by the protocol entities. These apply to service primitives of the required service, to service primitives of the underlying service, and to PDU manipulation. One major objective of protocol structuring is to show distribution (locality) and separation (orthogonality) of functions such that their mapping onto implementation resources is facilitated. The style of specification that best suits this objective is the **resource-oriented style** [VSS88].

A protocol design can make use of a number of concepts, and their representation in LOTOS, that the service design also uses. All concepts which are used in the local constraints of the service are also used in the protocol: SAP address and AEP identifier (see section 2.3), service primitive parameter (see section 2.4 through to section 2.6), and service primitive (see section 2.7). In the following, the prefixes 'upper' and 'lower' are used in combination with these concepts to distinguish between their use in relation to the required and underlying service respectively.

3.2 Association Reference

The source and destination of a PDU may be given explicitly, but may be implicit if they can be inferred. Every PDU may contain the source and destination addresses in full. This is usual in the simple connectionless case since each association, with (at most) one service primitive occurrence at each user, can be supported by the exchange of an isolated PDU. If the protocol entities can engage in multiple overlapping groups of service primitives, it is necessary to distinguish these. If a lower association is permanently assigned to a upper association, this can be done on basis of the fixed association between the identifiers of the lower and upper AEPs. If there is no fixed or one-to-one assignment, or when no lower AEP identifiers are used, an **association reference** must be provided by the protocol entities; this is called a **connection reference** in the connection-oriented case. An association reference may be composed of two parts, where each protocol entity provides one of the parts. Since an association reference (or part of it) uniquely identifies a protocol entity as well as the particular association supported by it, both purposes can be combined. Figure 4 depicts the use of association references.

```
type REF is ADDR renamedby                              (* association reference *)
   sortnames Ref for Addr
   opnnames
      BaseRef       for BaseAddr
      AnotherRef    for AnotherAddr
endtype (* REF *)
```

3.3 Sequence Number, Sequencing, Flow Control

PDUs may be numbered to support error and flow control. For practical reasons, sequence numbers must have an upper bound. Provided the underlying service provider can guarantee a limit on the life of a SDU that it is asked to transfer, the protocol can safely re-use sequence numbers used for earlier PDUs. A LOTOS representation of sequence numbers with an upper bound of 8 is:

```
type BASICSEQNO is BasicNaturalNumber renamedby          (* basic sequence number *)
```

```
  sortnames SeqNo for Nat
endtype (* BASICSEQNO *)

type SEQNO is BASICSEQNO                          (* sequence number *)
  formalopns Mod : -> SeqNo                       (* modulus constant *)
  formaleqns
    forall SeqNo : Nat
      ofsort SeqNo
        (SeqNo + Mod) = SeqNo;
endtype (* SEQNO *)
```

This might be instantiated as follows:

```
type MOD8 is BASICSEQNO                           (* modulo-8 number *)
  opns 8 : -> SeqNo
  eqns
    ofsort SeqNo
      8 = Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(0))))))));
endtype (* MOD8 *)

type SEQNO8 is SEQNO actualizedby MOD8 using      (* modulo-8 sequence number *)
  sortnames SeqNo8 for SeqNo
  opnnames 8 for Mod
endtype (* SEQNO8 *)
```

3.4 End of Service Data Unit

A protocol is required to preserve the integrity of SDUs that it transfers between service users. If the protocol carries out segmentation/reassembly (see section 3.7), it must indicate in PDUs whether they convey data corresponding to an intermediate or final part of a SDU. This is achieved by an EoSDU (**end of service data unit**) marker in each PDU:

```
type EOSDU is Boolean renamedby                   (* End of SDU *)
  sortnames EoSDU for Bool
endtype (* EOSDU *)
```

3.5 Protocol Data Unit, Protocol Control Information

During the design of a protocol entity, a separation of concerns calls for the introduction of PDUs that protocol entities use to support the required service. The part of a PDU that conveys a SDU, or a part of a SDU, is called the **user data** field. The remaining information in a PDU is termed PCI (**Protocol Control Information**).

In order to determine the protocol procedures for support of the required service, it is sufficient to consider PDUs in abstract terms. The representation of PDUs is relevant only when their exchange via the underlying service provider needs to be considered. Hence, it is useful to separate the definition of abstract PDUs and concrete PDUs. See section 3.6 for the concrete encoding of PDUs.

A LOTOS representation of abstract PDUs for a connection-less protocol might be:

```
type PDU is ADDR, REF, DATA, SEQNO, EOSDU         (* protocol data unit *)
  sorts Pdu
  opns
    DT  : Addr, Addr, Ref, Data, SeqNo, EoSDU -> Pdu  (* data *)
    AK  : Addr, Addr, SeqNo                   -> Pdu  (* acknowledgement *)
endtype (* PDU *)
```

A LOTOS representation of abstract PDUs for a connection-oriented protocol might be:

```
type PDU is ADDR, REF, DATA, SEQNO, EOSDU, REAS,  (* protocol data unit *)
 QOSSET, FUNOPT
  sorts Pdu
  opns
    CR  : Addr, Addr, Ref, QosSet, FunPar              -> Pdu   (* connect request *)
    CC  : Addr, Ref, Ref, QosSet, FunPar, Reas         -> Pdu   (* connect confirm *)
    DT  : Ref, SeqNo, EoSDU, Data                      -> Pdu   (* data *)
    AK  : Ref, SeqNo                                   -> Pdu   (* acknowledgement *)
    DR  : Ref, Reas                                    -> Pdu   (* disconnect request *)
    DC  : Ref                                          -> Pdu   (* disconnect confirm *)
endtype (* PDU *)
```

3.6 Protocol Data Unit Encoding

In order to ensure that PDUs are uniquely interpreted, a single representation or **encoding** for PDUs must be established. Many protocol functions can be defined to operate on abstract PDUs, independent of any PDU encoding. Some protocol functions, however, must operate on concrete PDUs, i.e. depend on the PDU encoding. Upper protocol functions (see section 3.15) can use abstract PDUs. Lower protocol functions (see section 3.15) require abstract and/or concrete PDUs. Functions that depend on the PDU encoding are, for example, handling incorrectly coded PDUs and PDU delimitation.

The specification of concrete PDUs and the ways in which they are structured are determined by the encoding rules adopted. To avoid showing any particular set of encoding rules, a number of assumptions are made below that are quite general. However, they still illustrate some important aspects of specifying concrete PDUs. As an example, consider a LOTOS representation of concrete PDUs in a connection-less protocol, with abstract PDUs as presented in section 3.5. The detailed encodings would be specified in types such as the following.

```
type PDUTYPECODE is Octet                               (* type code *)
  opns
    DTTypeCode, AKTypeCode: -> Octet
  eqns
    ofsort Octet
      DTTypeCode = Octet (1, 1, 1, 1, 0, 0, 0, 0);      (* for example *)
      AKTypeCode = Octet (0, 0, 0, 0, 1, 1, 1, 1)       (* for example *)
endtype (* PDUTYPECODE *)
```

Such encodings would be used to construct a whole PDU.

```
type CONCRETEPDU is OctetString, PDU,                   (* concrete encoding *)
 PDUTYPECODE, LENGTHCODE, ADDRCODE, REFCODE, DATACODE,
 SEQNOCODE, EOSDUCODE
  opns
    _Encodes_, _EncodesDT_, _EncodesAK_  : OctetString, Pdu      -> Bool
                                        (* recognisers of correct encoding *)
    _Decodes_                            : Pdu, OctetString      -> Bool
                                        (* recogniser of correct decoding *)
    DTEncoding, AKEncoding               : Pdu                   -> OctetString
                                        (* constructors of encoding *)
  eqns
    forall
     Octets : OctetString, Pdu : Pdu, Addr1, Addr2 : Addr, Ref : Ref, Data : Data,
     SeqNo : SeqNo, EoSDU : EoSDU
      ofsort Bool
        Octets Encodes Pdu = (Octets EncodesDT Pdu) or (Octets EncodesAK Pdu);
        IsDT (Pdu) =>
          Octets EncodesDT Pdu = Octets eq DTEncoding(Pdu);
        not (IsDT (Pdu)) =>
          Octets EncodesDT Pdu = false;
        IsAK (Pdu) =>
          Octets EncodesAK Pdu = Octets eq AKEncoding (Pdu);
        not (IsAK (Pdu)) =>
          Octets EncodesAK Pdu = false;
      ofsort OctetString
        DTEncoding (DT (Addr1, Addr2, Ref, Data, SeqNo, EoSDU)) =
          DTTypeCode +
            (
              LengthCode (Length (AddrCode (Addr1)) +
                Length (AddrCode (Addr2)) +
                  Length (RefCode (Ref)) +
                    Length (DataCode (Data)) +
                      Length (SeqNoCode (SeqNo)) +
                        Length (EoSDUCode (EoSDU))) ++
                AddrCode (Addr1) ++
                  AddrCode (Addr2) ++
                    CRefCode (Ref) ++
                    DataCode (Data) ++
                      SeqNoCode (SeqNo) ++
                        EoSDUCode (EoSDU)
            );
        not (IsDT (Pdu)) =>
          DTEncoding (Pdu) = <>;
        AKEncoding (AK (Addr1, Addr2, SeqNo)) =
          AKTypeCode +
            (
              LengthCode (Length (AddrCode (Addr1)) +
                Length (AddrCode (Addr2)) +
                  Length (SeqNoCode (SeqNo))) ++
                AddrCode(Addr1) ++
                  AddrCode(Addr2) ++
                    SeqNoCode(SeqNo)
            )
        not (IsAk (Pdu)) =>
          AKEncoding (Pdu) = <>;
      ofsort Bool
        Pdu Decodes Octets = Octets Encodes Pdu;
endtype (* CONCRETEPDU *)
```

3.7 Segmentation, Reassembly

An SDU may be **segmented** (also called **fragmented**) into a number of PDUs; the inverse operation at the receiver is **reassembly**. The optimum size of PDUs depends on the characteristics of the underlying path, i.e. the lower association. The OSI architecture states that segmentation and reassembly are inverse operations, but does not prescribe the manner in which they are carried out; this is left to individual protocol standards.

```
type SEGMENT is DATA                                        (* segmentation/reassembly *)
  opns
    segment_pdu         : Data          -> Data             (* next PDU from data in SDU *)
    segment_sdu         : Data          -> Data             (* SDU left after removing PDU *)
    reassemble_sdu      : Data, Data    -> Data             (* new SDU after adding PDU *)
  eqns
    forall pdu, pdu1, pdu2 : Data, sdu : Data
      ofsort Data
        segment_pdu (<>) = <>;
        segment_pdu (reassemble_sdu (pdu, <>)) = pdu;
        segment_pdu (reassemble_sdu (pdu2, reassemble_sdu (pdu1, sdu))) =
          segment_pdu (reassemble_sdu (pdu1, sdu));
        segment_sdu (<>) = <>;
        segment_sdu (reassemble_sdu (pdu, <>)) = <>;
        segment_sdu (reassemble_sdu (pdu2, reassemble_sdu (pdu1, sdu))) =
          reassemble_sdu (pdu2, segment_sdu (reassemble_sdu (pdu1, sdu)));
endtype (* SEGMENT *)
```

3.8 Blocking, Deblocking

An SDU may be **blocked** with others into a PDU; the inverse operation at the receiver is **deblocking**. The OSI architecture states that blocking and deblocking are inverse operations, but does not prescribe the manner in which they are carried out; this is left to individual protocol standards.

```
type BLOCK is DATA                                              (* blocking/deblocking *)
  opns
    deblock_pdu   : Data              -> Data                   (* PDU left after removing SDU *)
    deblock_sdu   : Data              -> Data                   (* next SDU from data in PDU *)
    block_pdu     : Data, Data        -> Data                   (* new PDU after adding SDU *)
  eqns
    forall pdu : Data, sdu, sdu1, sdu2 : Data
      ofsort Data
        deblock_pdu (<>) = <>;
        deblock_pdu (block_pdu (<>, sdu)) = <>;
        deblock_pdu (block_pdu (block_pdu (pdu, sdu1), sdu2)) =
          block_pdu (deblock_pdu (block_pdu (pdu, sdu1)), sdu2);
        deblock_sdu (<>) = <>; deblock_sdu (block_pdu (<>, sdu)) = sdu;
        deblock_sdu (block_pdu (block_pdu (pdu, sdu1), sdu2)) =
          deblock_sdu (block_pdu (pdu, sdu1));
endtype (* BLOCK *)
```

3.9 Concatenation, Separation

A PDU may be **concatenated** with others into a SDU; the inverse operation at the receiver is **separation**. The OSI architecture states that concatenation and separation are inverse operations, but does not prescribe the manner in which they are carried out; this is left to individual protocol standards.

```
type CONCAT is DATA                                             (* concatenation/separation *)
  opns
    separate_pdu          : Data          -> Data               (* next PDU from data in SDU *)
    separate_sdu          : Data          -> Data               (* SDU left after removing PDU *)
    concatenate_sdu       : Data, Data    -> Data               (* new SDU after adding PDU *)
  eqns
    forall pdu, pdu1, pdu2 : Data, sdu : Data
      ofsort Data
        separate_pdu (<>) = <>;
        separate_pdu (concatenate_sdu (pdu, <>)) = pdu;
        separate_pdu (concatenate_sdu (pdu2,
        concatenate_sdu (pdu1, sdu))) =
          separate_pdu (concatenate_sdu (pdu1, sdu));
        separate_sdu (<>) = <>;
        separate_sdu (concatenate_sdu (pdu, <>)) = <>;
        separate_sdu (concatenate_sdu (pdu2, concatenate_sdu (pdu1, sdu))) =
          concatenate_sdu (pdu2, separate_sdu (concatenate_sdu (pdu1, sdu)));
endtype (* CONCAT *)
```

3.10 Routing

A protocol must decide which lower association to use for transmission of a PDU; this is a **routing** decision. The OSI architecture does not prescribe the manner in which routing is carried out (though particular standards may). The only logical requirement is that routing gets a PDU to its destination ‘efficiently’. The ability to make a routing decision implies the existence of some network information database.

A LOTOS representation of routing is as follows. In the specification below, the operation *route* takes a PDU, a pair (or address) identifying the remote upper AEP (or SAP), and network information. The operation *route* returns a set of pairs (or addresses) identifying remote lower AEPs (or SAPs) of possible associations to be used for transmission. The operation is normally one-to-one and will return a single lower pair that is unique for the upper pair. However, if multiplexing or splitting are in use then the routing operation is many-to-one or one-to-many respectively (see sections 3.11 and 3.12). Routing is many-to-many if both functions are in use. It is not easy to specify much about routing in general. Broad requirements are that routing chooses a lower pair from the available set, and that this pair is indeed on the route to the remote upper pair. These requirements are not covered below, as they would require a complex specification for the network information type *NETINFO* which has been omitted for simplicity.

```
type ROUTE is DATA, UPRPAIR, LWRPAIRSET, NETINFO                (* route *)
  opns
    route : Data, UprPair, NetInfo -> LwrPairSet
endtype (* ROUTE *)
```

3.11 Multiplexing, Demultiplexing

Multiplexing is the ability of a protocol entity to support more than one association over a single lower association; the inverse operation at the receiver is **demultiplexing**. The associations must, of course, all be with the same remote protocol entity. Multiplexing necessitates the use of association references (see section 3.2). A protocol entity that uses multiplexing will have a mapping function that causes several upper AEP identifiers to be mapped onto one lower AEP identifier. The OSI architecture does not prescribe the manner in which multiplexing and demultiplexing are carried out, only that they be inverses. The LOTOS representation of multiplexing and demultiplexing would be similar to that of a routing operation that is many-to-one or many-to-many in mapping between upper and lower pairs (see section 3.10).

3.12 Splitting, Recombining

Splitting is the ability of a protocol entity to support an association over a number of lower associations; the inverse operation at the receiver is **recombining**. The associations must, of course, all be with the same remote protocol entity. Splitting necessitates the use of association references (see section 3.2). A protocol entity that uses splitting will have a mapping function that causes one upper AEP identifier to be mapped onto several lower AEP identifiers. The OSI architecture does not prescribe the manner in which splitting and recombining are carried out, only that they be inverses. The LOTOS representation of splitting and recombining would be similar to that of a routing operation that is one-to-many or many-to-many in mapping between upper and lower pairs (see section 3.10).

3.13 Protocol Layer, Underlying Service Provider, Protocol Entity

The first level of protocol structuring yields a layer of protocol entities and an underlying service provider. This 'vertical' structuring of the **protocol layer** into protocol entities is a consequence of localising protocol functions. Each protocol entity shares a set of SAPs with a service user and another set of SAPs with the underlying service provider.

The LOTOS specification below shows the combination of the protocol layer and underlying service provider. The gate *upr* represents the boundary between the users of the required service and the protocol layer. The gate *lwr* represents the boundary between the protocol layer and the underlying service provider; *lwr* is hidden. In addition a choice is made between sets of addresses to identify lower SAPs; any choice represents a possible implementation of the provider of the required service. For simplicity, each protocol entity is assumed to have exactly one upper SAP and one lower SAP. The SAP addresses and AEP identifiers at both services are also assumed to be of the same type. This requires that there be as many upper SAPs as lower SAPs, a condition imposed by the guard in the specification of *Prot*.

```
process Prot [upr] (UprAddrs : UprAddrSet) : noexit :=          (* protocol *)
  hide lwr in
    choice LwrAddrs : LwrAddrSet []
```

```
      [Card (LwrAddrs) eq Card (UprAddrs)] ->
        (
          ProtEnts [upr, lwr] (UprAddrs, LwrAddrs)
        |[lwr]|
          UnderServProv [lwr] (LwrAddrs)
        )
endproc (* Prot *)
```

The LOTOS specification below shows the structuring of the protocol layer as a multiplicity of independent protocol entities, each of which is instantiated with an upper and a lower SAP address. Unique use of addresses is accomplished by removing used addresses from the sets of available addresses with each new instantiation of *ProtEnts*:

```
process ProtEnts [upr, lwr] (UprAddrs : UprAddrSet, LwrAddrs : LwrAddrSet) : noexit :=
                                                              (* protocol entities *)
  [UprAddrs eq {}] ->
    stop
  []
    (
      choice UprAddr : UprAddr, LwrAddr : LwrAddr []
        [(UprAddr IsIn UprAddrs) and (LwrAddr IsIn LwrAddrs)] ->
          (
            ProtEnt [upr, lwr] (UprAddr, LwrAddr)
          |||
            ProtEnts [upr, lwr]
              (Remove (UprAddr, UprAddrs), Remove (LwrAddr, LwrAddrs))
          )
    )
endproc (* ProtEnts *)
```

3.14 Protocol Entity Constraints, Protocol Entity Invocation

A protocol entity supports upper associations using lower associations. If there is a one-to-one relation between these associations, for each such relation a **protocol entity invocation** may be described independently of other such invocations. This is not the case when, for example, the service types are different or there are connections that are multiplexed, split or re-used by the protocol entity. Each invocation performs the role of **initiator** (when the local service user is of type 'calling') or **responder** (when the local service user is of type 'called').

A LOTOS representation of a protocol entity is given below as a composition of orthogonal constraint sets, similar to the constraints identified for a connection-oriented service (see section 2.9). The refusal processes can be structured as the independent composition of constraints at the upper and lower SAPs.

```
process ProtEnt [upr, lwr] (UprAddr : UprAddr, LwrAddr : LwrAddr) : noexit :=
                                                              (* protocol entity *)
  ProtEntInvocs [upr, lwr] (UprAddr, LwrAddr)
||
  ProtEntDataRefusals [upr, lwr] (UprAddr, LwrAddr)
||
  ProtEntPairRefusals [upr, lwr] (UprAddr, LwrAddr, {}, {})
||
  ProtEntConnRefusals [upr, lwr] (UprAddr, LwrAddr)
endproc (* ProtEnt *)
```

The following shows the requirements on protocol entity invocations as a multiplicity of independent constraint sets, each one applicable to a single protocol entity invocation. A protocol entity invocation is instantiated with a pair to identify its upper and lower AEPs.

```
process ProtEntInvocs [upr, lwr] (UprAddr : UprAddr, LwrAddr : LwrAddr) : noexit :=
                                                              (* protocol entity invocations *)
  (
    choice UprIdent : UprIdent, LwrIdent : LwrIdent []
      ProtEntInvoc [upr, lwr] (Pair (UprAddr, UprIdent), Pair (LwrAddr, LwrIdent))
  )
|||
  ProtEntInvocs [upr, lwr] (UprAddr, LwrAddr)
endproc (* ProtEntInvocs *)
```

3.15 Upper and Lower Protocol Functions

A protocol entity invocation may be structured in terms of orthogonal constraint sets much as for an association (see section 2.10). There are **local constraints** related to the behaviour at the SAP or AEP of each upper and each lower association, and there are **mapping constraints** on the relation between service primitive occurrences at the upper and lower SAPs or AEPs. The mapping constraints concern the way in which a protocol entity invocation supports an upper association by using a lower association, and therefore implement part of the remote constraints of the upper association.

From a LOTOS point of view, local constraints at an upper SAP or AEP are just the local constraints of the related upper association as they appear in the service specification (unless these include additional constraints on usage of the service). The local constraints at a lower SAP or AEP comply with the local constraints in the underlying service.

The mapping constraints may be structured into **upper protocol functions** and **lower protocol functions**. A typical set of upper protocol functions consists of reliability enhancement functions for use when more reliable data transfer is needed than the provider of underlying service can offer. Such functions could be based on some kind of retransmission mechanism to cater for lost or corrupted PDUs. In addition, PDUs may be constrained as to the amount of user data which can be conveyed. A segmentation/reassembly function would therefore be added to the upper protocol functions (see section 3.7).

The lower protocol functions are concerned with optimal use of the underlying service provider transfer capability, both in time and space. Two independent aspects can be distinguished in this task. First, an unambiguous and efficient coded representation of PDUs must be defined. Second, PDU flow must be adapted to the underlying service provider characteristics, requiring functions such as concatenation/separation, multiplexing/demultiplexing and splitting/recombining (see sections 3.9, 3.11 and 3.12).

This structuring in terms of upper and lower protocol functions obviously depends on the required and underlying service types that together characterise protocol functions. A LOTOS representation of structuring a protocol entity invocation in terms of local and mapping constraints is the following.

```
process ProtEntInvoc [upr, lwr] (UprPair : UprPair, LwrPair : LwrPair) : noexit :=
                                                        (* protocol entity invocation *)
  (
    UprLocal [upr] (UprPair)
  |||
    (LwrLocal [lwr] (LwrPair) [] exit)
  )
||
  (
    ProtEntMap [upr, lwr] (UprPair, LwrPair)
  [>
    exit
  )
endproc (* ProtEntInvoc *)
```

4 Conclusion

The architecture of OSI has been discussed in some detail, focusing on the concepts that underlie services and protocols. The objective of this study has been to derive representations in LOTOS that illustrate the essential architectural features of these concepts. Using these building blocks, a specifier can be guided to produce specifications of layered systems in a more consistent and productive fashion. On the LOTOSPHERE project this approach was used in the development of realistic OSI applications.

Developing a specification component library for the OSI architecture has opened further avenues for exploration. The same approach should be applicable to a number of other problem domains where LOTOS might be required. Preliminary work has been undertaken to incorporate the specification components described in this paper into a library that can be used through a pre-processor. Such a pre-processor would support another level of language via LOTOS, rather than extending LOTOS. It also remains to be investigated how specifications could be developed in a 'macro', architectural fashion based on these ideas.

Acknowledgements

The work reported in this paper was undertaken in the context of LOTOSPHERE Tasks concerned with specification architecture and specification pilot studies. It has therefore benefited from the feedback and comments of co-workers in these Tasks. The work also draws on earlier work conducted by the first author as editor for the *Architectural Semantics for FDTs* [ISO89c] and *Guidelines for the Application of* ESTELLE, LOTOS *and SDL* [ISO91].

References

[Fer89] L. Ferreira Pires: 'On the use of LOTOS to support the design of a connection-oriented internetting protocol', *in ESPRIT Conference 1989*, pp. 957–970, North-Holland, 1989.

[FA89] D. Freestone and S. S. Aujla: 'Specifying ROSE in LOTOS', *in* K. J. Turner (ed.): *Formal Description Techniques I*, pp. 231–245, North-Holland, 1989.

[ISO89a] ISO/IEC: *Information Processing Systems – Open Systems Interconnection – Description in LOTOS of the Connection-Oriented Session Service*, ISO/IEC TR 9571, International Organisation for Standardisation, Geneva, 1989.

[ISO89b] ISO/IEC: *Information Processing Systems – Open Systems Interconnection – Description in LOTOS of the Connection-Oriented Session Protocol*, ISO/IEC TR 9572, International Organisation for Standardisation, Geneva, 1989.

[ISO89c] ISO/IEC: *Architectural Semantics for FDTs*, ISO/IEC JTC 1/SC 21/N4231, International Organisation for Standardisation, Geneva, 1989.

[ISO90a] ISO/IEC: *Information Processing Systems – Open Systems Interconnection – Description in* LOTOS *of the Connection-Oriented Transport Service*, ISO/IEC TR 10023, International Organisation for Standardisation, Geneva, 1990.

[ISO90b] ISO/IEC: *Information Processing Systems – Open Systems Interconnection – Description in LOTOS of the Connection-Oriented Transport Protocol*, ISO/IEC TR 10024, International Organisation for Standardisation, Geneva, 1990.

[ISO91] ISO/IEC: *Information Processing Systems – Open Systems Interconnection – Guidelines for the Application of ESTELLE, LOTOS and SDL*, ISO/IEC TR 10167, International Organisation for Standardisation, Geneva, 1991.

[ISO92] ISO/IEC: *Information Processing Systems – Open Systems Interconnection – Conventions for the Definition of OSI Services*, ISO/IEC 10731, International Organisation for Standardisation, Geneva, 1992.

[JC90] V. M. Jones and R. G Clark: 'LOTOS specification of the OSI CCR protocol', Lo/WP3/T3.1/UST/N0003/V04, ESPRIT Project 2304, Commission of the European Communities, Brussels, 1990.

[LS89] J. van de Lagemaat and G. Scollo: 'On the use of LOTOS for the formal description of a transport protocol', *in* K. J. Turner (ed.): *Formal Description Techniques I*, pp. 247–262, North-Holland, 1989.

[SA89] M. van Sinderen and I. Ajubi: 'The application of LOTOS for the formal description of the ISO session layer', *in* K. J. Turner (ed.): *Formal Description Techniques I*, pp. 263–278, North-Holland, 1989.

[Sad90] F. Sadoun: 'LOTOS specification of the OSI CCR service', Lo/WP3/T3.1/SYS/ N0007/V02, ESPRIT Project 2304, Commission of the European Communities, 1990.

[SW91] M. van Sinderen and I. Widya: 'On the design and formal specification of a transaction processing protocol', *in* J. Quemada, J. Mañas, and E. Vazquez (eds.): *Formal Description Techniques III*, pp. 411–426, North-Holland, 1991.

[Tur88] K. J. Turner: 'An architectural semantics for LOTOS', *in* H. Rudin and C. West (eds.) *Protocol Specification, Testing, and Verification VII*, pp. 15–28, North-Holland, 1988.

[Tur89] K. J. Turner: 'A LOTOS case study: specification of the OSI connection-oriented network service', OTC Workshop on Formal Techniques, Sydney, July 1989.

[VF92] C. A. Vissers and L. Ferreira Pires: 'LOTOSPHERE, an attempt towards a design culture', in this book.

[VL86] C. A. Vissers and L. Logrippo: 'The importance of the service concept in the design of data communications protocols', *in* M. Diaz (ed.): *Protocol Specification, Testing, and Verification V*, pp. 3–18, North-Holland, 1986.

[VSS88] C. A. Vissers, G. Scollo, and M. van Sinderen: 'Architecture and specification style in formal descriptions of distributed systems', *in* S. Aggarwal and K. Sabnani (eds.): *Protocol Specification, Testing, and Verification VIII*, pp. 189–204, North-Holland, 1988.

[WHR90] I. Widya, G. van der Heijden, and F. Riddoch: 'LOTOS description of the TP protocol', Lo/WP3/T3.1/N0020/V02, ESPRIT Project LOTOSPHERE, Commission of the European Communities, 1990.