*Kenneth J. Turner. Template-based specification in LOTOS. Department of Computing Science and Mathematics, University of Stirling, Scotland, April 1993.*

# Template-Based Specification in LOTOS

**Kenneth J. Turner**

*Department of Computing Science, University of Stirling, Scotland FK9 4LA*
*Electronic Mail: kjt@cs.stir.ac.uk*

The notion of a skeletal specification is introduced as a step in the incremental development of a specification. This leads to the idea of a template specification as a means of capturing specification architecture in a library of reusable specification components. The approach is illustrated by considering the architecture of service specifications in LOTOS. A library of macros written in the *m4* macro language is used to generate LOTOS specifications automatically.

## 1 Introduction

The formal specification language LOTOS (*Language Of Temporal Ordering Specification*, [4]) is one of three FDTs (*Formal Description Techniques*) that have been standardised for the specification of (tele)communications systems. LOTOS has already been widely used for specifying OSI services and protocols such as:

**TP** *(Transaction Processing)* protocol (ISO 10026.3) formalised in [21]

**CCR** *(Commitment, Concurrency and Recovery)* service (ISO 9804) formalised in [14], protocol (ISO 9805) formalised in [10]

**ROSE** *(Remote Operations Service Elements)* service element (ISO 9072/1) formalised in [3]

**Session Layer** service (ISO 8326) formalised in [5], protocol (ISO 8327) formalised in [6]; see also [15] for an explanation of the approach

**Transport Layer** service (ISO 8072) formalised in [7], protocol (ISO 8073) formalised in [8]; see also [11] for an explanation of the approach

**Network Layer** service (ISO 8348) formalised in [19], protocol (ISO 8473) formalised in [12].

Although a similar approach has been taken in these specifications, little attempt has been made to codify the overall approach as part of a 'how to' guide; [16] is one of the few papers which addresses this. The specification of services and protocols in LOTOS and the other FDTs is illustrated in [9]. At a more fundamental level, [17] shows how to represent architectural concepts in LOTOS, and [1] does the same for SDL.

Services and protocols are particular kinds of systems that are found in communications systems, but are not restricted to them; they arise in any layered architecture. Other applications such as Operating Systems or Computer Integrated Manufacturing could also be regarded as layered systems. It would be useful for specifiers to have a pre-defined framework within which LOTOS specifications could be written of services and protocols.

It is common experience that the most difficult and critical aspect of formalisation is the choice of **specification architecture**, i.e. the structure of the specification. The architecture defines the specification components and their relationship. This decomposition of functionality may reflect a number of goals:

- separation of concerns
- modularity, information-hiding, encapsulation
- comprehension, enhancement, maintenance, sub-division of work

- mapping onto implementation components and structures.

Depending on the precise goals, appropriate specification styles can be chosen such as constraint-oriented or resource-oriented (see [20]).

LOTOS specifiers would be assisted in writing new specifications if they had access to a library of specification architectures as well as specification styles. A particular style restricts the ways in which specification components may be composed. However, this is too general to help in the choice of architecture for a particular application. What is needed is an outline framework – a **template** – for specifying each class of application. This is comparable to the concept of **component re-use** in Software Engineering, except that the components in question are parts of specifications rather than re-usable implementation modules.

## 2 Incremental Specification

### 2.1 Skeletal Specifications

Complexity and size are major obstacles in the specification and design of realistic systems. A common solution is a hierarchic approach, which exhibits modularity and information-hiding. Another common approach is **deferred specification**, in which details are omitted at one level of abstraction and are completed in later refinements of the specifications. This differs from the use of **non-determinism**, in which implementation decisions are represented explicitly or implicitly as dynamic choices in the behaviour of a system. Deferred specification can be used effectively with LOTOS as follows:

1. At the first stage of design, only the types, processes and gates are introduced and related to each other. This indicates the basic components of the specification.

2. At an intermediate stage of design, the sorts, signatures of operations, process parameters, and process results are introduced. This indicates the interfaces to the basic components, but without defining the operation of the components formally.

3. At the last stage of design, the equations defining the operations and the behaviour expressions defining the processes are introduced. The specification is now textually complete, but still needs static and dynamic checking.

A **skeletal specification** may thus be progressed through a number of stages of refinement, finally resulting in a complete specification. Different parts of a specification may be elaborated at different times. Typically a specification will be structured into related groups of types and processes. These may be progressively specified and checked in a bottom-up fashion. As with conventional programming languages, purpose-designed 'test harnesses' may be useful in checking parts of a specification. A test harness can be constructed as a process which is composed in parallel with the system at the topmost level of behaviour. If required, a skeletal specification may be partially checked with tools by adding 'null' bodies to type and process definitions. An example of incrementally developing a specification in the above stages is given in appendix A.

### 2.2 Specification Templates

Specification templates support the development of skeletal specifications. They are pre-defined components of specifications (e.g. groups of related type and process definitions), or even whole specifications in their own right. A template is a syntactic concept, unlike a parameterised data type or process. Templates are outside definition of the LOTOS, but lead to specifications in standard LOTOS. The advantage of using templates is that a specifier can already begin well down the route of incrementally developing a specification. In particular, a template which defines the overall architecture of a specification will eliminate much of the difficult work on choosing an appropriate specification structure.

Templates allow a specifier to work at a much higher, architectural level than LOTOS allows. Templates deal with large-scale concepts such as service primitives, data transfer, a connection-less service, or a connection-oriented protocol. This relieves the specifier of much of the basic work in structuring specifications, and imposes uniformity on the way in which specifications are written. This facilitates the integration of specifications (e.g. of adjacent protocol layers) and their validation (e.g. showing that a protocol satisfies the service it supports). Although templates do not have formal semantics in their own right, their meaning is given by the semantics of the LOTOS specifications they generate. Templates can ease verification because the specification components they generate can be analysed generically, once and for all. Templates can also be used to enforce syntactic restrictions such as those which guarantee sufficient completeness or persistency of data types [13].

Templates may be parameterised to make them general-purpose. Such templates can be designed to be filled in automatically using a macro processor or text editor. Although templates can deal with many aspects of a specification, some details can be completed only by hand. For example, it may not be cost-effective to specify templates for features of only a few applications. It is also not feasible to incorporate trickier details which require manual intervention. Although a specifier may still have a lot of work to complete a specification produced by templates, much of the framework will have been laid down. Given a good structure, a specifier is more likely to produce a satisfactory result.

The particular notation used for templates is irrelevant. For the work reported in this paper the widely available *m4* macro language has been used. The template library uses the following features of *m4*:

- definition and 'undefinition' of macros
- parameterised macros (up to 9 parameters) and unparameterised macros
- controlled macro expansion (quoting text '...' delays expansion)
- conditional expansion depending on a value or whether a macro is defined
- string operations such as producing or finding a substring.

With care and ingenuity these features are used to achieve more sophisticated effects in the library:

- text variables
- loops using recursion
- hierarchies of macros
- higher-order macros (which take macros as parameters)
- 'curried' macros (which have been partially applied to some of their parameters).

## 3 An Example Architecture: A Service

The following discussion is based on the concepts of OSI (ISO 7498 and ISO 8509), but is generally applicable to any layered system. The elements of a service are discussed, and a LOTOS representation is discussed. This leads to a template which contains the generic features of a service.

### 3.1 Service Structure

A **service** is a black box which offers a number of **facilities**. The **service provider** interacts with **service users** at a number of abstract interfaces called **service access points**. Service users are distinguished by means of a unique **title**, and service access points are distinguished by means of a unique **address**.

Distinct groups of interactions between a service user and the service provider may occur at the same service access point. In the case of a connection-oriented service these correspond to the behaviour of individual connections; a **connection endpoint identifier** is used to distinguish these. In the case of a simple connection-less service each invocation of a service primitive is independent, so such a concept is not needed. Intermediate kinds of service may support (short) groups of interactions which may be overlapped with other such groups (e.g. an acknowledged connection-less service); an **interaction group identifier** is used to distinguish these. Although the concept of an interaction group is not recognised in OSI, it is used in this paper because of its more general nature.

The interactions between users and the provider are specified abstractly as **service primitives**. Service primitives have one or more parameters, the most important of which indicates the type of primitive. A parameter of many service primitives is a **service data unit** – user data which is transferred transparently by the service.

## 3.2 Title, Address, Interaction Group Identifier

Titles, addresses and interaction group identifiers are simply sets of distinct labels. Titles and addresses are globally unique within the scope of OSI, whereas interaction group identifiers are unique only within the scope of an address. Since titles are associated with service users, they are not required in the specification of a service provider.

All these kinds of identifiers may have structure. For example, an address may be divided into an addressing authority code, a country code, a network code, and a system code. However, such a structure is for convenience in allocation or routing and may be ignored at an abstract level. The identifiers are simply specified in LOTOS as distinct values in a sort. Since the set may be infinite, it is constructed inductively from a base value and an operation to yield another address from a given one.

## 3.3 Service User, Service Provider, Service Primitive

From a LOTOS viewpoint, the service provider is the system to be specified and the service users form the environment of the system. The service provider and service users are therefore specified by behaviour expressions with associated type definitions. It also follows that interactions between service users and the service provider are specified as LOTOS events. OSI is indefinite about the nature of service primitives, e.g. whether they occur synchronously, atomically 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 specification. In a more refined specification (including a protocol specification) it is possible to model the occurrence of service primitives as asynchronous, interruptible and spread out in time.

## 3.4 Service Access Point, Interaction Group

In abstract terms, a service access point exists only to distinguish sequences of interactions between one service user and the service provider[1]. A service access point therefore has a *behavioural* rather than a *structural* interpretation. At first it might seem that each service access point should correspond to an individual LOTOS gate. However, LOTOS requires gates to be listed explicitly, which is not possible in general for service access points. A single gate is therefore used for communication with a service[2]. Interactions at a particular service access point are therefore distinguished by means of an address as the primary parameter of an event. Since an interaction group acts as a finer structure within a service access point, it is identified by a secondary parameter of an event. At a global level, it is convenient to deal with the identity of an interaction group as a **pair** consisting of a service access point address and an interaction group identifier.

## 3.5 Service Primitive Parameter, Service Data Unit

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. All events at a service boundary therefore have a common format:

service_gate ! address ! interaction_group_identifier ! service_primitive

Since the type of a service primitive is its most important parameter, this is indicated as the name of the **constructor** operation which builds the record, rather than including the name as a field in its own right. **Selector** operations are needed to extract fields of the parameter record. For uniformity, all service primitive constructors yield values of the same sort so that primitives can be manipulated without being forced to deal with their type. **Recogniser** operations are therefore needed to distinguish values produced by each constructor.

Because the data typing mechanism in LOTOS is very general, it does not have constructs which allow records to be specified directly. To avoid incompleteness in specifications, LOTOS operations must be defined as *total*. A naïve specification of $m$ different service primitive types with $n$ different types of parameters would lead to $m \times n$ equations for the selector operations and $m^2$ equations for the recogniser operations. The number of equations for the recogniser operations can be made linear in $m$ if each constructor operation is identified with the natural numbers; for example, this allows equality of service primitives to be re-cast as equality of the naturals.

[1] A service user may employ several service access points.

[2] Note that the use of a single gate does not imply a single interface in an implementation.

Selector operations can be entirely dispensed with if records are always used 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 which did not have these fields. It would therefore be better to dispense with the selector operations and to access the fields constructively. For example, the fields of a given connection request primitive *CR* might be accessed by:

```
choice Src, Dst : Addr []
  [ConReq (Src, Dst) eq CR] ->
     (* LOTOS text referring to the values of Src and Dst *)
```

Service data units are often parameters of service primitives. Since the service provider does not operate on service data units, it is sufficient to have only the constructor operations for a list of data values; the standard library type *OctetString* provides what is needed. However, the specification must indicate whether service data units are transferred in one piece at the service boundary or whether they are transferred in smaller units. At the level of abstraction appropriate for a service specification, service data units should be regarded as indivisible.

## 3.6 Service Provider Constraints

At the topmost level, the behaviour of the service provider can be factored into a number of different concerns:

- dealing with interaction groups; in the connection-oriented case this includes support of connections
- refusing to initiate a new interaction group for an existing pair of service access point addresses and interaction group identifiers; in the connection-oriented case this means refusing to accept a new connection at an endpoint that is already in use
- refusing to initiate a new interaction group when there are insufficient resources; in the connection-oriented case this means refusing to accept a new connection
- refusing to accept new data when the service is congested.

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 specification. Such a style is appropriate for giving an abstract, high-level specification as required for a service. In LOTOS terms, the constraints are behaviour expressions that are fully synchronised in parallel (||). These parallel constraints will normally synchronise on each event, but one constraint may (temporarily) forbid an event by not offering it. The three refusal processes operate by permitting actions in certain interaction groups. The pair refusal process keeps a record of those pairs currently in use. The initiation refusal process non-deterministically chooses a subset of the pairs at which initiation is permitted; the subset may change over time. The data transfer refusal process operates similarly.

## 3.7 Interaction Group

### 3.7.1 Phases

By separating out the interference between different interaction groups at the top level of the specification, the behaviour of each interaction group can be specified by independent constraints. Many different mechanisms could be imagined to set up interaction groups, e.g. pre-allocation, allocation from a central pool of free resources, or allocation from distributed pools. Abstractly speaking, the resources to support interaction groups are dynamically bound when they are required. Initiation and termination may be implicit rather than explicit, e.g. as in a connection-less service. Within data transfer there may be priorities, e.g. expedited data taking precedence over normal data, or reset taking priority over all data transfer. It is therefore natural to specify the behaviour of an interaction group in a number of phases. These phases occur within the local and the remote constraints.

Refusal to initiate an interaction group may be given in the informal description as part of the termination function. As careful study of actual services will show, this may be incorrect as the procedures for refusal and termination may be only superficially similar. Refusal should therefore be regarded as part of the initiation function. The relationship between phases may be one of pre-emption (e.g. termination interrupts whatever is happening) or one of priority (e.g. expedited data is processed in preference to normal data). Pre-emption corresponds to non-deterministic disabling in LOTOS (... [> **i**; ...), while priority corresponds to non-deterministic selection of the preferred branch of a choice (... [] **i**; ...). A schematic LOTOS structure for an *InteractionGroup* process is therefore:

```
    Initiation
  >>
    (
      DataTransfer
    [>
      (i; Termination >> InteractionGroup)
    )
```

When normal data transfer is to be blocked, an **i** event causes only expedited data to be offered. Normal data may become unblocked before expedited data is actually transferred, hence the **exit** in the following. In the schematic structure below for a *DataTransfer* process, *NormalData* and *ExpeditedData* handle only one primitive at a time and then **exit**:

```
    (
      NormalData
    []
      i; (ExpeditedData [] exit)
    )
  >>
    DataTransfer
```

#### 3.7.2 Local Constraints

At one service access point or in one interaction group, there are **local constraints** on the order in which service primitives may occur. There are also 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 a service is involves just one user and the provider, the local constraints will fully define it. More normally a service involves the service provider as an intermediary between two users. In general, two or more users may be associated in a group of interactions. If a service is symmetrical (peer-to-peer) then the local constraints will be identical for each party, but in general they may be different (primary-secondary, master-slave, client-server). Local constraints are independent and so are interleaved in LOTOS (|||).

#### 3.7.3 Remote Constraints

For two or more users, the service provider relates interactions on an end-to-end basis. These **remote constraints** may simply involve the indication at one user of a request by the other. In the multi-user case the provider may require to broadcast the request of one user to all corresponding users. The local constraints deal with common concerns such as initiation or termination of an interaction group, and reset or resynchronisation of data transfer within an interaction group. This allows the flow of data from source user to sink user to be specified independently. In LOTOS terms, the concerns above are fully synchronised (||), whereas the data transfer in each direction is interleaved in parallel (|||).

### 3.8 Service Objects, Service medium

Service primitives are conventionally named according to their purpose: **request** (source user to provider), **indication** (provider to destination user), **response** (destination user to provider), and **confirm** (provider to source user). Inside the service provider, however, a request or an indication corresponds to a **message**, and a response or a confirm corresponds to an **acknowledgement**. It is therefore necessary to distinguish between the **service objects** manipulated by the provider and their external manifestation as service primitives. Objects have the same parameters as primitives, so their constructors, selectors and recognisers are defined similarly or in terms of the operations on primitives.

A service may allow the provider to reorder objects. The simplest service behaves as a first-in-first-out queue for each direction of transfer. However, services may define more complex relationships:

- **overtaking**, whereby an object exchanges places with the one ahead of it (e.g. expedited data overtaking normal data)

- **destruction**, whereby an object causes removal of the one ahead of it (e.g. a reset clearing out any data ahead of it)
- **cancellation**, whereby an object mutually annihilates the one ahead of it (e.g. a disconnect catching up with and destroying a connect if a connection attempt is abandoned).

To specify such behaviour as being that of a queue would be misleading, so the term **medium** is used. This uses the LOTOS library data type *String*, with additional operations defined by the specifier for the reordering possibilities above. Since an algorithm for promoting objects through the medium would be implementation-dependent, an abstract approach is followed. After acceptance or delivery of a service primitive, the state of the medium is allowed to change non-deterministically in accordance with the reordering rules. The extreme cases are to promote all priority messages to the head of the medium, and to carry out no reordering at all.

## 4 An Example Template: A Service

### 4.1 The Template Library

All of the architectural decisions justified in the previous section are encapsulated in templates for specifying services. The templates are actually calls on a library of macros which contribute different aspects of a service specification. Many of the macros are general-purpose and apply to different kinds of services, e.g. connection-less as well as connection-oriented. Service and protocol specifications share a substantial amount of text related to the behaviour at a service access point (e.g. local constraints, service primitives, and service primitive parameters). Common templates can therefore support protocol specification as well as service specification. A particularly pleasing feature is that, architecturally, a protocol can be regarded as a kind of service; its users are the adjacent layers. This symmetry makes a common approach even easier.

A library of *m4* macros has been developed to support specification of services. The macros are rather intricate and of interest only to those who wish to enhance or modify the library. They are therefore not reproduced here, although they are available on request from the author[3]. Generally speaking, each data type or process is generated by one macro, but there are auxiliary and general-purpose macros to support these. Most of the complexity stems from having to deal with the data types for service primitives and service objects. The macro library allows these to be defined succinctly as record structures. If the user follows the naming conventions assumed in the library, it is possible to generate substantial amounts of LOTOS automatically. Those parts which cannot be generated automatically are annotated with '*(*** ... ***)*' comments to the specifier as to what must be completed. The macro library currently produces only a few explanatory comments in the generated LOTOS text, but adding more comments would be trivial.

A very simple template generates the type *DATA* for service data units (see appendix B). This fixed text is generated by calling the *data_type* macro which is defined in *m4* as:

```
define(data_type,`
  type DATA is OctetString renamedby
    sortnames Data for OctetString
  endtype (* DATA *)')
```

A slightly more complex template generates the process *ConnRemData* (see appendix B). This parameterised text is generated by the call *conn_rem_data_proc(co)*, where the macro is defined in *m4* as:

```
(conn_rem_data_proc,`
        process ConnRemData [$1] (PairX,PairY : Pair,Med : Med) : exit (Med) :=
            ConnRemDataNrm [$1] (PairX,PairY,Med)
          []
            ConnRemDataExp [$1] (PairX,PairY,Med)
          where
conn_rem_data_nrm_proc($1)
conn_rem_data_exp_proc($1)
        endproc (* ConnRemData *)')
```

[3]The library used to support the example in this paper consists of about 70 macros defined in about 530 lines of *m4*, excluding explanatory comments and blank lines.

The notation '$1' refers to the first parameter of the macro (the service gate name). The macro calls two subsidiary macros (*conn_rem_data_nrm_proc* and *conn_rem_data_exp_proc*) to generate the nested processes (*ConnRemDataNrm* and *ConnRemDataExp*).

A more complex template generates the equations for the operation *Ord* which yields an ordinal number for a service primitive (see type *PRIM* in appendix B). This parameterised text is generated a call such as *prim_ord_eqn(ConReq,[Addr1/Addr2])*, where the macro is defined in *m4* as:

```
define(prim_ord_eqn,
  `                    Ord ($1`'ifelse($2,,,
    ` '(extrn_of($2)))) = ord;define(`ord',
      Succ (Ord ($1`'ifelse($2,,,
    ` '(extrn_of($2))))))
')
```

Here, '$1' is the service primitive name and '$2' is a list of its parameters. The built-in macro *ifelse* checks whether its first parameter is equal to its second; if so then the third parameter is returned, else the fourth parameter. The macro library uses an internal representation of service primitives and their parameters such as *Conreq[Addr1/Addr2]* rather than *Conreq(Addr1,Addr2)* because '(', ',' and ')' are special characters in *m4*. A complication in generating the equation is that the primitive may have no parameters. In there are parameters present, the external representation of the parameters is generated by macro *extrn_of*. Inside *prim_ord_eqn*, the variable *ord* (actually a macro) is repeatedly re-defined with the current service primitive and parameters for use on the next call.

## 4.2 An Example

Suppose it is necessary to specify a connection-oriented service with the following features:

- connection initiation is user-confirmed; a responding address is returned if the attempt is successful
- normal and expedited data transfer is supported, without confirmation of delivery
- disconnection is unconfirmed.

The key elements of the service specification are the service name (taken as *CO*) and the service primitives (*ConReq* for connection request, etc.). If the LOTOS templates are held in the file *lotos.m4*, the following *m4* text will read the definitions and generate the LOTOS specification given in appendix B:

```
include(lotos.m4)

co_serv_spec(
  CO,
  ConReq(Addr1,Addr2) ConInd(Addr1,Addr2)
  ConRsp(Addr)        ConCnf(Addr)
  DatReq(Data)        DatInd(Data)
  ExpReq(Data)        ExpInd(Data)
  DisReq()            DisInd()
)
```

The specifier must respect the conventions of the macro library, but the price is small compared to the saving in specification effort. Service primitives must be listed with initiating primitives first (e.g. *ConReq*, *ConInd*) and terminating primitives last (e.g. *DisReq*, *DisInd*). The data transfer primitives must be defined by the specifier in the generated LOTOS. Service primitive names must have standard suffixes (*Req*, *Ind*, *Rsp*, *Cnf*). Service primitive parameters must be named according to their LOTOS sort, and multiple instances in a primitive must be numbered uniquely.

## 5 Conclusion

The notion of a skeletal specification has been introduced as a stage in the incremental development of a complete specification. Three basic stages in incremental development have been proposed: an outline specification with only types, processes and their relationships; an intermediate specification with signatures and process headings; and a complete specification. Template specifications systematise the use of skeletal specifications by making available a library of pre-defined specification components. Templates may be parameterised, and may generate complete specifications or only specifications with major structure defined, depending on the complexity of the specification. By using a macro language such as *m4*, large amounts of specification text may be generated automatically.

Templates can be used to capture architectural decisions in the style and structure of a specification. This allows the specifier to work at an architectural level and to deal with large-scale concepts. Apart from obvious productivity gains, this encourages a uniform approach to specification, helps in the integration of specifications, and can save verification effort. The advantages of incremental specification and template specification are demonstrated with examples of simple connection-less and connection-oriented services in the appendixes. It is hoped to extend this approach to the specification of protocols, and to other classes of applications.

## Acknowledgements

The architectural decisions embodied in LOTOS service specifications are based on the work of a number of other authors including myself. Individuals who deserve particular mention for their specification work are Juan Quemada, Giuseppe Scollo, Marten van Sinderen and Alastair Tocher.

## References

[1] Belina, F., Hogrefe, D. and Trigila, S.: ‘Modelling OSI in SDL’, *in*: [18], pp. 135–142

[2] van Eijk, P. H. J., Vissers, C. A. and Diaz, M. (*eds.*): ‘The Formal Description Technique LOTOS’, North-Holland

[3] Freestone, D. and Aujla, S. S.: ‘Specifying ROSE in LOTOS’, *in*: [18], pp. 231–245

[4] ISO: ‘Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour’, *ISO 8807*, 1988

[5] ISO: ‘Information Processing Systems – Open Systems Interconnection – Formal Description in LOTOS of the Connection-Oriented Session Session’, *ISO TR 9571*, 1989

[6] ISO: ‘Information Processing Systems – Open Systems Interconnection – Formal Description in LOTOS of the Connection-Oriented Session Protocol’, *ISO TR 9572*, 1989

[7] ISO: ‘Information Processing Systems – Open Systems Interconnection – Formal Description in LOTOS of the Connection-Oriented Transport Service’, *ISO TR 10023*, 1990

[8] ISO: ‘Information Processing Systems – Open Systems Interconnection – Formal Description in LOTOS of the Connection-Oriented Transport Protocol’, *ISO TR 10024*, 1990

[9] ISO: ‘Guidelines for the Application of ESTELLE, LOTOS and SDL’, *ISO DTR 10167*, January 1990

[10] Jones, V. M. and Clark, R. G.: ‘LOTOS Specification of the OSI CCR Protocol’, Lo/WP3/T3.1/UST/N0003/V04, ESPRIT Project 2304, Commission of the European Communities, Brussels, April 1990

[11] van de Lagemaat, J. and Scollo, G.: ‘On the Use of LOTOS for the Formal Description of a Transport Protocol’, *in*: [18], pp. 247–261

[12] PANGLOSS: ‘Specification of the OSI Connection-Less Internet Protocol’, Esprit Project 890, Commission of the European Communities, Brussels, December 1989

[13] Padawitz, P.: ‘Parameter-Preserving Data Type Specifications’, *in*: ‘Mathematical Foundations of Software Development’, *Lecture Notes in Computer Science, 185*, Springer-Verlag, Berlin, 1985

[14] Sadoun, F.: ‘LOTOS Specification of the OSI CCR Service’, Lo/WP3/T3.1/SYS/N0007/V02, ESPRIT Project 2304, Commission of the European Communities, Brussels, February 1990

[15] van Sinderen, M., Ajubi, I. and Caneschi, F.: ‘The Application of LOTOS for the Formal Description of the ISO Session Layer’, *in*: [18], pp. 263–277

[16] van Sinderen, M.: ‘Generic Service and Protocol Structures’, Lo/WP3/T3.3/UT/N0012, ESPRIT Project 2304, Commission of the European Communities, Brussels, April 1990

[17] Turner, K. J.: ‘An Architectural Semantics for LOTOS’, *in*: Rudin, H. and West, C. H. (*eds.*): ‘Protocol Specification, Testing and Verification VII’, pp. 15–28, North-Holland, 1987

[18] Turner, K. J. (*ed.*): ‘Formal Description Techniques 88’, North-Holland, 1988

[19] Turner, K. J.: ‘A LOTOS Case Study: Specification of the OSI Connection-Oriented Network Service’, Proc. of OTC Workshop on Formal Techniques, Sydney, July 1989

[20] Vissers, C. A. *et al.*: ‘Architecture and Specification Style in Formal Descriptions of Distributed Systems’, *in*: Aggarwal, S. and Sabnani, K. H. (*eds.*): ‘Protocol Specification, Testing and Verification VIII’, North-Holland, 1988

[21] Widya, I., van der Heijden, G. and Riddoch, F.: ‘LOTOS Description of the TP Protocol’, Lo/WP3/T3/1/N0020/V02,ESPRIT Project 2304, Commission of the European Communities, Brussels, April 1990

## A Incremental Development of a Connection-Less Service

As an example of incrementally developing a specification, consider a simple connection-less service that supports data transfer between pairs of service access points without guarantee of delivery or sequencing of messages. The following specifications correspond to the stages given in section 2.1. The notation ‘...’ indicates omitted LOTOS text.

1. The basic components of the specification are first defined as:

```
specification CLService [cl] ...

  library Boolean, OctetString ...

  type ADDR is Boolean ...

  type DATA is OctetString renamedby ...

  type PRIM is DATA ...

  behaviour TransferAll [cl]

    process TransferAll [cl] ...

      process TransferOne [cl] ...
```

2. After addition of sorts, operations and process headings, the specification will look like:

```
specification CLService [cl] : noexit

  library
    Boolean, OctetString
  endlib

  type ADDR is Boolean
    sorts Addr
    opns
      BaseAddr          : -> Addr
```

```
        AnotherAddr       : Addr -> Addr
        _ne_              : Addr, Addr -> Bool
    endtype

    type DATA is OctetString renamedby
      sortnames Data for OctetString
    endtype

    type PRIM is DATA
      sorts Prim
      opns
        DatReq, DatInd : Data -> Prim
    endtype

    behaviour TransferAll [cl]

      where

      process TransferAll [cl] : noexit :=
        stop
        where

        process TransferOne [cl] (Src, Dst : Addr, SDU : Data) : noexit :=
          stop
        endproc

      endproc

  endspec
```

3. Finally, adding equations and behaviour expressions will yield:

```
  specification CLService [cl] : noexit

    library
      Boolean, OctetString
    endlib

    type ADDR is Boolean
      sorts Addr
      opns
        BaseAddr          : -> Addr
        AnotherAddr       : Addr -> Addr
        _ne               : Addr, Addr -> Bool
      eqns forall AddrA, AddrB : Addr
        ofsort Bool
          BaseAddr ne BaseAddr                              = false;
          AnotherAddr (AddrA) ne BaseAddr                   = true;
          BaseAddr ne AnotherAddr (AddrB)                   = true;
          AnotherAddr (AddrA) ne AnotherAddr (AddrB)        = AddrA ne AddrB;
    endtype

    type DATA is OctetString renamedby
      sortnames Data for OctetString
    endtype
  type PRIM is DATA
    sorts Prim
    opns
      DatReq, DatInd : Data -> Prim
  endtype

  behaviour TransferAll [cl]

    where

    process TransferAll [cl] : noexit :=
      choice SDU : Data []
        cl ? Src : Addr ? Dst : Addr ! DatReq (SDU) [Src ne Dst];
        (TransferOne [cl] (Src, Dst, SDU) ||| TransferAll [cl])
      where

      process TransferOne [cl] (Src, Dst : Addr, SDU : Data) : noexit :=
        cl ! Src ! Dst ! DatInd (SDU); stop                    (* deliver message *)
      []
        i; stop                                                (* lose message *)

      endproc

    endproc

endspec
```

## B An Example Specification: A Service

The following specification is taken literally from the result of the macro call given in section 4.2. For brevity, similar equations have been omitted, but they are generated in full by the macros. The specification uses hopefully obvious abbreviations such as ‘Loc’ for ‘Local’ and ‘Rem’ for ‘Remote’.

```
specification COService [co] : noexit

  library
    Boolean,BasicNaturalNumber,Set,String,OctetString
      (*** import further types for primitive parameters ***)
  endlib

  behaviour
    Conns [co]
  ||
    PairRefusals [co] ( of PairSet)
  ||
    ConnRefusals [co]
  ||
    DataRefusals [co]

  where

  type DATA is OctetString renamedby

    sortnames Data for OctetString
```

```
endtype (* DATA *)

type ADDR is Boolean

  sorts Addr

  opns
    BaseAddr : -> Addr
    AnotherAddr : Addr -> Addr
    _eq_,_ne_ : Addr,Addr -> Bool

  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

  sortnames Ident for Addr

  opnnames
    BaseIdent for BaseAddr
    AnotherIdent for AnotherAddr

endtype (* IDENT *)

type PAIR is ADDR,IDENT

  sorts Pair

  opns
    Pair : Addr,Ident -> Pair
    Addr : Pair -> Addr
    Ident : Pair -> Ident

    _eq_,_ne_ : Pair, Pair -> Bool

  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 *)

type PAIRSET is Set actualizedby PAIR using

  sortnames
    PairSet for Set
    Pair for Element
    Bool for FBool

endtype (* PAIRSET *)

(*** insert further types for primitive parameters ***)

type PRIM is Boolean, BasicNaturalNumber, DATA, ADDR
  (*** import further types for primitive parameters ***)

  sorts Prim

  opns
    ConReq : Addr,Addr -> Prim
    ConInd : Addr,Addr -> Prim
    ConRsp : Addr -> Prim
    ConCnf : Addr -> Prim
    DatReq : Data -> Prim
    DatInd : Data -> Prim
    ExpReq : Data -> Prim
    ExpInd : Data -> Prim
    DisReq : -> Prim
    DisInd : -> Prim

    IsConReq,IsConInd,IsConRsp,IsConCnf,IsDatReq,IsDatInd,
      IsExpReq,IsExpInd,IsDisReq,IsDisInd : Prim -> Bool
    IsReq,IsInd,IsInit,IsData,IsTerm : Prim -> Bool

    Ord : Prim -> Nat

    _eq_,_ne_ : Prim,Prim -> Bool

  eqns
    forall
      Prim,PrimA,PrimB : Prim,
      Addr,AddrA,AddrB,Addr1,Addr1A,Addr1B,Addr2,Addr2A,Addr2B : Addr,
      Data,DataA,DataB : Data

      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 IsConRsp (Prim)) or IsDatReq (Prim))
                  or IsExpReq (Prim)) or IsDisReq (Prim);
            IsInd (Prim) =
               (((IsConInd (Prim) or IsConCnf (Prim)) or IsDatInd (Prim))
                  or IsExpInd (Prim)) or IsDisInd (Prim);
            IsInit (Prim) = IsConReq (Prim) or IsConInd (Prim);
            (*** insert equation for IsData ***)
            IsTerm (Prim) = IsDisReq (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 *)

type OBJ is PRIM

   sorts Obj

   opns
      Req : Prim -> Obj
      Ind : Obj -> Prim

      IsConMsg,IsConAck,IsDatMsg,IsExpMsg,IsDisMsg : Obj -> Bool

      _eq_,_ne_ : Obj,Obj -> Bool

   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

  sortnames
    Med for String
    Obj for Element
    Bool for FBool

endtype (* MED *)

type MEDSET is Set actualizedby MED using

  sortnames
    MedSet for Set
    Med for Element
    Bool for FBool

endtype (* MEDSET *)

type MEDOPS is MED

  opns
    _overtakes_,_destroys_,_cancels_,_ignores_ : Obj,Obj -> Bool

    SetOf : Med -> MedSet

    _&_ : MedSet,Med -> MedSet (* prefix medium to elements *)
    _&_ : Med,MedSet -> MedSet (* append medium to elements *)

    Reorders : Med -> MedSet (* reorderings of medium *)
    Reorders : MedSet -> MedSet (* reorderings of medium set *)

  eqns
    forall
      Obj,ObjA,ObjB : Obj,
      Med,MedA,MedB : Med,
      MedSet : MedSet

      ofsort Bool
        (*** insert equations for overtakes, destroys, cancels ***)
        ObjA ignores ObjB =
          not (
            ((ObjA overtakes ObjB) or (ObjA destroys ObjB))
              or (ObjA cancels ObjB));

      ofsort MedSet
        SetOf (Med) = Insert (Med, );

         & Med = ;
        Insert (MedA, MedSet) & MedB = Insert (MedA++MedB, MedSet & MedB);

        Med &  = ;
        MedA & Insert (MedB, MedSet) = Insert (MedA++MedB, MedA & MedSet);

        Reorders (<>) = ;
        Reorders (<> + Obj) = ;
        ObjA overtakes ObjB =>
          Reorders ((<> + ObjA) + ObjB) = SetOf ((<> + ObjB) + ObjA);
        ObjA destroys ObjB =>
          Reorders ((<> + ObjA) + ObjB) = SetOf (<> + ObjA);
        ObjA cancels ObjB =>
          Reorders ((<> + ObjA) + ObjB) = SetOf (<>);
        ObjA ignores ObjB =>
          Reorders ((<> + ObjA) + ObjB) = ;

        Reorders ((Med + ObjA) + ObjB) =
          Reorders (
            ((Med & Reorders ((<> + ObjB) + ObjA))
              Union (Reorders (Med + ObjB) & ObjA))
               Union (Reorders (Med) & ((<> + ObjB) + ObjA)));

endtype (* MEDOPS *)
process Conns [co] : noexit :=

  Conn [co] ||| Conns [co]

  where

  process Conn [co] : noexit :=

    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 :=

      ConnLocInit [co] (PairX)
    >>
      (
        ConnLocData [co] (PairX)
      [>
        (
          i;
          ConnLocTerm [co] (PairX)
        >>
          ConnLoc [co] (PairX)
        )
      )

      where

      process ConnLocInit [co] (PairX : Pair): exit :=

        exit (*** insert local initiation constraints ***)

      endproc (* ConnLocInit *)

      process ConnLocData [co] (PairX : Pair): noexit :=

        (
          ConnLocDataNrm [co] (PairX)
        []
          i;
          (ConnLocDataExp [co] (PairX) [] exit)
        )
      >>
        ConnLocData [co] (PairX)

        where

        process ConnLocDataNrm [co] (PairX : Pair) : noexit :=

          stop (*** insert local normal data transfer constraints ***)

        endproc (* ConnLocDataNrm *)

        process ConnLocDataExp [co] (PairX : Pair) : noexit :=

          stop (*** insert local expedited data transfer constraints ***)

        endproc (* ConnLocDataExp *)

      endproc (* ConnLocData *)

      process ConnLocTerm [co] (PairX : Pair) : exit :=

        exit (*** insert local termination constraints ***)

      endproc (* ConnLocTerm *)

    endproc (* ConnLoc *)

    process ConnRem [co] (PairX,PairY : Pair,Med1 : Med) : noexit :=

      (
        ConnRemInit [co] (PairX,PairY,Med1)
      []
        ConnRemData [co] (PairX,PairY,Med1)
      []
```

```
            ConnRemTerm [co] (PairX,PairY,Med1)
         )
      >>
         accept Med2 : Med in
            (
               choice Med3 : Med []
                  [(Med3 eq Med2) or (Med3 IsIn Reorders (Med2))] ->
                     i;
                     ConnRem [co] (PairX,PairY,Med3)
            )

         where

         process ConnRemInit [co] (PairX,PairY : Pair,Med : Med): exit (Med):=

            exit (any Med) (*** insert remote initiation constraints ***)

         endproc (* ConnRemInit *)

         process ConnRemData [co] (PairX,PairY : Pair,Med : Med) : exit (Med) :=

               ConnRemDataNrm [co] (PairX,PairY,Med)
            []
               ConnRemDataExp [co] (PairX,PairY,Med)

            where

            process ConnRemDataNrm [co] (PairX,PairY : Pair,Med : Med) : exit (Med) :=

               exit (any Med) (*** insert remote normal data transfer constraints ***)

            endproc (* ConnRemDataNrm *)

            process ConnRemDataExp [co] (PairX,PairY : Pair,Med : Med) : exit (Med) :=

               exit (any Med) (*** insert remote expedited data transfer constraints ***)

            endproc (* ConnRemDataExp *)

         endproc (* ConnRemData *)

         process ConnRemTerm [co] (PairX,PairY : Pair,Med : Med): exit (Med) :=

            exit (any Med) (*** insert remote termination constraints ***)

         endproc (* ConnRemTerm *)

      endproc (* ConnRem *)

   endproc (* Conn *)

endproc (* Conns *)

process PairRefusals [co] (UsedPairs : PairSet) : noexit :=
        choice Pair : Pair []
          co ! Addr (Pair) ! Ident (Pair) ? Prim : Prim
            [IsInit (Prim) Implies (Pair NotIn UsedPairs)];
          (
            [IsInit (Prim)] ->
              PairRefusals [co] (Insert (Pair, UsedPairs))
          []
            [IsTerm (Prim)] ->
              PairRefusals [co] (Remove (Pair, UsedPairs))
          []
            [not (IsInit (Prim) or IsTerm (Prim))] ->
              PairRefusals [co] (UsedPairs)
          )

      endproc (* PairRefusals *)

      process ConnRefusals [co] : noexit :=

        choice Pair : Pair,ConnPairs : PairSet []
          co ! Addr (Pair) ! Ident (Pair) ? Prim : Prim
            [IsInit (Prim) Implies (Pair IsIn ConnPairs)];
          ConnRefusals [co]
        []
          i; (* revise set of acceptable pairs *)
          ConnRefusals [co]

      endproc (* ConnRefusals *)

      process DataRefusals [co] : noexit :=

        choice Pair : Pair,DataPairs : PairSet []
          co ! Addr (Pair) ! Ident (Pair) ? Prim : Prim
            [(IsData (Prim) and IsReq (Prim)) Implies (Pair IsIn DataPairs)];
          DataRefusals [co]
        []
          i; (* revise set of acceptable pairs *)
          DataRefusals [co]

      endproc (* DataRefusals *)

    endspec (* COService *)
```