Kenneth J. Turner. The alternating bit protocol - Constraint-oriented specifications in LOTOS. In Proc. British Computer Society Workshop on Formal Methods in Standards, pages 1-13, London, April 1988. British Computer Society.

Constraint-Oriented Style in LOTOS

Kenneth J. Turner

Department of Computing Science and Mathematics
University of Stirling, Stirling FK9 4LA, Scotland

Telephone: +44-1786-467-420 *Facsimile:* +44-1786-464-551
Email: kjt@cs.stir.ac.uk *Web:* http://www.cs.stir.ac.uk/~kjt/

1st April 1988*

Abstract

The concept of a constraint-oriented specification style is presented in general terms and with respect to the ISO Formal Description Technique LOTOS (*Language Of Temporal Ordering Specification*). The constraint-oriented style has proven very suitable for specifying the abstract, implementation-independent behaviour of systems in a modular fashion. The essential idea behind constraint-oriented specification is separation of concerns, which is facilitated by the behaviour combinators of LOTOS. The constraint-oriented style is illustrated by giving a highly-structured specification in LOTOS of the well-known AB (*Alternating Bit*) Protocol.

1 Introduction

ISO and CCITT have been working for about 8 years on standards for FDTs (*Formal Description Techniques*). The objectives of these FDTs are:

- *unambiguous*, *clear* and *concise* specifications
- a basis for determining *completeness* of specifications
- a foundation for *analysing* specifications for correctness, effectiveness, etc.
- a basis for determining *conformance* of implementations to specifications
- a basis for determining *consistency* of specifications relative to each other
- a basis for *implementation* support.

Work is now reaching completion on standards for the FDTs Estelle (*Extended Finite State Machine Language*, [3]), LOTOS (*Language Of Temporal Ordering Specification*, [4]), and SDL (*Specification and Description Language*, [1]).

The three FDTs share a common underlying behavioural model (*labelled transition systems*). To some extent they also share a common ADT (*Abstract Data Type*) data model, based on *equational axioms* and *initial algebras*. However, they differ in their emphasis on matters such as degree of abstractness, completeness of underlying formal semantics, ease of analysis, convenience for implementation, etc.

*Sic!

Most formal specification languages (including the FDTs) have been applied to the well-worn example of the AB (*Alternating Bit*) Protocol. A number of examples are collected in [12]. Curiously enough, no definitive reference for the AB Protocol seems to exist[1]. Indeed, possible sources such as [12, 13] are contradictory, and in some cases self-contradictory! For the purposes of this paper, the (semi)-formal description given in [12, Figure 3.3] has been taken and formalised in LOTOS. The informal equivalent of this is as follows:

> The AB Protocol supports a reliable, uni-directional, connection-less, data transfer Service between a pair of Users. AB Service Data Units may contain one or more octets.
>
> The AB Protocol is supported by a UM (*Unreliable Medium*) Service. Both the AB Service and the UM Service support two Service Access Points, corresponding to the source and the sink of AB data. To prevent loss or duplication of AB Service Data Units, AB Protocol Data Units carry a sequence number. These sequence numbers are calculated modulo 2, i.e. they alternate between 0 and 1.
>
> The UM Service may lose Service Data Units, but may not corrupt or duplicate them. It is assumed that message corruption will lead to a message being discarded within the medium, so that message corruption appears as message loss. Since the AB Protocol has a sliding window of size 1, there is no point in requiring that the UM buffer more than one message. The issue of misordering in the medium does not, therefore, arise. Since the UM Service supports only two Service Access Points, the issue of misdelivery does not arise either.
>
> The AB Protocol does not support blocking, segmentation, or concatenation. One AB Service Data Unit therefore corresponds to one AB Protocol Data Unit and to one UM Service Data Unit.
>
> An AB source accepts a Service Data Unit via an AB Service Request. The AB source sends the data message with the current sequence number via a UM Service Request. The first sequence number to be sent is 0. If an acknowledgement which has the *next* sequence number is received from the UM Service, the AB source is free to accept another AB Service Request. If an acknowledgement which does not have the next sequence number is received, the data message is sent again. If no acknowledgement is received within some unspecified time-limit, the data message is also sent again.
>
> An AB sink accepts a Protocol Data Unit via a UM Service Indication. The sequence number of this data message is checked, the first expected being 0. If a data message which has the expected sequence number is received, the message is delivered via an AB Service Indication. An acknowledgement with the *next* sequence number is sent. If a data message which does not have the expected sequence number is received, it is discarded but an acknowledgement with the expected sequence number is still sent.

Although the AB Protocol is rather simple, it forms the basis of the *sliding window* mechanism which appears in many standards, including those for OSI (*Open Systems Interconnection*). The AB Service is also prototypical of many OSI Services.

2 LOTOS for the Specification of Standards

Comparative studies of the FDTs such as [10, 16] are now beginning to emerge. Large-scale application of the FDTs to OSI standards have also been undertaken (e.g. see the references in [9, 14]). The results of these studies show that LOTOS is very suitable for producing implementation-independent specifications of OSI standards. The reasons for this success include:

- LOTOS allows *formal analysis* of the syntactic and semantic correctness of specifications
- LOTOS allows *abstract* specifications to be written, focussing on the externally-observable ordering of events

[1] There would be some justification in proposing an ISO standard for it! To confuse the issue, the Alternating Bit Protocol or a variant of it is often given another name (e.g. *Idle RQ* [6], or *Stop-and-Wait ARQ* [11]).

- LOTOS allows specifications to be written in a *modular* fashion
- LOTOS allows *separation of concerns* in writing specifications.

These are all important issues in producing specifications of international standards. For other purposes, such as giving a reference implementation, Estelle or SDL could be more appropriate than LOTOS.

The ability of LOTOS to allow separation of concerns stems largely from the nature of parallel and alternate composition in LOTOS (the ||, |[]|, |||, and [] operators). These behave much like the logical connectives $\wedge$ and $\vee$ respectively[2]. These properties have led to the widespread use of a *constraint-oriented* style in LOTOS. However, because of its abstractness and unfamiliarity to newcomers, the constraint-oriented style deserves some explanation.

3 Constraint-Oriented Style in LOTOS

The constraint-oriented style is similar to specifying a system by giving logical assertions, constraints, invariants, or properties which the behaviour of the system satisfies. As an example, consider a normal person's day as far as eating and being awake are concerned:

- the constraints on eating are:
  - *breakfast* is followed by *lunch*
  - *lunch* is followed by *dinner* or *tea*
- the constraints on eating in relation to being awake are:
  - *waking* is followed by *breakfast*
  - *dinner* or *tea* is followed by *sleeping*

A less obvious aspect, implicit in the above natural language description, is that the terms *breakfast*, *lunch*, *dinner*, and *tea* refer to the same event in all cases. This would not necessarily be so in a LOTOS description, where it may be useful *not* to synchronise on events with the same name. The above constraints translate directly into LOTOS:

```
    (
      (breakfast; lunch; exit)
    |[lunch]|
      (lunch; (dinner; exit  []  tea; exit))
    )
  |[breakfast, dinner, tea]|
    (
      (waking; breakfast; exit)
    |||
      (dinner; sleeping; exit  []  tea; sleeping; exit)
    )
```

Solving a cross-word puzzle or doing linear programming are useful analogies in helping to understand the constraint-oriented style. In a cross-word puzzle, the constraints are that the solutions must fit the clues, and where the solutions intersect in the grid they must define the same letters. In linear programming, the constraints define a permissible set of values (a volume in the solution space). In both analogies, the constraints may lead to:

[2] Note that LOTOS deals only with *potential* behaviour and does not have the equivalent of logical negation, $\neg$.

- a set of solutions, corresponding to alternative event offers in LOTOS
- a unique solution, corresponding to *the* event which is offered in LOTOS
- no solution, corresponding to *deadlock* in LOTOS.

The deserved popularity of the constraint-oriented style stems from its ability to express different aspects (or *projections*) of a complex system in separate and manageable specification modules. The arguments against the constraint-oriented style are that:

- it diffuses system behaviour throughout the specification
- it is hard to implement.

These objections are discussed in the following sections.

4 Constraint-Oriented Style and Component Engineering

The argument that the constraint-oriented style diffuses information about global system behaviour is fair. It *can* be hard to grasp the overall behaviour of a LOTOS specification from a knowledge of its parts and their relationships. However, this is the price to be paid for having complex systems. Although LOTOS specifications are generally written top-down, it may be easier to analyse a constraint-oriented specification from the bottom up. In this respect, the constraint-oriented style reflects a component-engineering approach.

An electronic engineer may decompose a complex function into its component parts: multi-vibrators, Schmidt triggers, operational amplifiers, etc. Each of these may then be decomposed into its basic components: resistors, capacitors, transistors, etc. (The component fabricator will carry this decomposition further: thin films, dielectrics, doped silicon, etc.) To understand the workings of the complete circuit decomposition would be very difficult, and probably impracticable. Instead, it is better to analyse the properties of each part in terms of the properties and relationships of its components. In this way, confidence in the circuit design can be obtained in a bottom-up fashion. At each level of the design, the properties of the components can be assumed and used to derive properties of the part.

Communications engineers are unfortunately not as well off as electronic (or indeed most) engineers when it comes to a component-engineering approach. What *are* the basic components of communications services and protocols? The OSI Basic Reference Model ([2]) defines some static components (e.g. service access point, layer, and title) and some dynamic components (e.g. multiplexing, segmentation, and flow control). However, these concepts are informally (and sometimes fuzzily) defined, and are still quite high-level. Some recent work [15] has focussed on the formalisation in LOTOS of some of these basic architectural concepts. Comparable formalisations in Estelle and SDL [5, Chapter 7] have also been produced. The work of the Alvey project FORMAP (*Formal Methods Applied to Protocols*) is also relevant to component-engineering of communications systems.

5 Constraint-Oriented Style and Implementation

The argument that constraint-oriented style leads to specifications which are hard to implement is also fair. It is easy to dismiss this objection if the goal of using LOTOS is *specification* rather than *implementation*. For international standards, the main objective is to write specifications which are clear, precise, compact, and implementation-independent. However, a useful standard must lead to conforming implementations. Unfortunately, the constraint-oriented style is rather abstract and non-constructive. It relies on the powerful, but hard to implement, LOTOS feature for multi-way synchronisation on multi-valued event offers.

Suppose, for example, that a sorting algorithm were specified in LOTOS as:

```
    Permutations [output] (UnsortedList)
  ||
    Orderings [output]
```

where **Permutations** offered to output all possible permutations of an unsorted list of numbers, and **Orderings** offered to output all possible lists of ordered numbers. The effect of their composition is clear as a specification: the only event(s) offered will contain the sorted list. However, as a prescription for a direct implementation the specification above would be useless. This constraint-oriented specification would need to be refined to a more constructive form before implementation could be considered. The Esprit project PANGLOSS (*Parallel Architecture for Networking Gateways Linking OSI Systems*) is studying such correctness-preserving transformations as part of its work. Other Esprit work in this area is also anticipated.

6 Other Specification Styles in LOTOS

Unlike the behavioural part of LOTOS, the data typing part virtually forces a bottom-up, component-based view. This is supported by the *renaming*, *enrichment*, and *parameterisation* features which are usual in ADTs. If anything, the data typing part of LOTOS encourages an object-oriented rather than a constraint-oriented approach. Nonetheless, the data typing can be used to support a constraint-oriented style directly. For example, the previous specification of sorting could be recast as:

```
choice SortedList : NatList []
  [IsPermuted (SortedList, UnsortedList) and
    IsOrdered (SortedList)] ->
      output ! SortedList
```

LOTOS specifications can often be written in a data-oriented or a behaviour-oriented style. Other aspects which may be emphasised in LOTOS specifications include:

- guidance in implementation (e.g. algorithms, data structures, and inter-process communication)
- management of resources (e.g. connections, protocol handlers, and gateways)
- ease of verification (e.g. [7, 8])
- explicit representation of system state and state variables.

7 Application to the Alternating Bit Protocol

```
(*-------------------------------------------------------------------
```

The specification of the AB Protocol is parameterised by the gate **ab** for communication with AB Service Users.

```
-------------------------------------------------------------------*)

specification AlternatingBitProtocol [ab] : noexit

(*-------------------------------------------------------------------
```

The specification uses the standard library data types **Bit** and **OctetString**.

For simplicity, AB and UM Service Access Points are specified as having addresses **0** or **1**, corresponding to the AB source or sink respectively. These are conveniently expressed as values of sort **Bit**. The alternating sequence numbers are also expressed as values of sort **Bit**. Standard operations on bits include **eq** (boolean equality) and **ne** (boolean inequality).

AB Service Data Units are directly expressed as values of sort **OctetString**. Standard operations on octet strings include <> (empty string), **eq** (boolean equality), and **ne** (boolean inequality).

```
------------------------------------------------------------------*)

  library Bit, OctetString endlib

(*------------------------------------------------------------------
```

The standard library sort **Bit** is enriched with a **next** operation (for sequence numbers) which is equivalent to a complement operation.

```
------------------------------------------------------------------*)

  type RicherBitType is Bit

    opns next : Bit -> Bit

    eqns

      ofsort Bit
        next (0) = 1;
        next (1) = 0

  endtype (* RicherBitType *)

(*------------------------------------------------------------------
```

AB Service Primitives are of sort **ABSp**. They are constructed by **ABSreq** (for transmitting) and **ABSind** (for receiving), which both take a Service Data Unit as a parameter. Because AB Service Primitives are only *constructed* in the specification, no selectors are defined.

```
------------------------------------------------------------------*)

  type ABSpType is OctetString

   sorts ABSp

   opns ABSreq, ABSind : OctetString -> ABSp

  endtype (* ABSpType *)

(*------------------------------------------------------------------
```

AB Protocol Data Units are of sort **ABPdu**. They are constructed by **ABPmess** (for a data message) and **ABPack** (for an acknowledgement), which both take a one-bit sequence number as a parameter. **ABPmess** additionally takes a Service Data Unit as a parameter. Because AB Protocol Data Units are only *constructed* in the specification, no selectors are defined.

```
-------------------------------------------------------------*)

  type ABPduType is Bit, OctetString

    sorts ABPdu

    opns ABPmess : Bit, OctetString -> ABPdu
         ABPack  : Bit -> ABPdu

  endtype (* ABPduType *)

(*-------------------------------------------------------------
```

UM Service Primitives are of sort **UMSp**. They are constructed by **UMSreq** (for transmitting) and **UMSind** (for receiving), which both take a Protocol Data Unit as a parameter. Because UM Service Primitives are only *constructed* in the specification, no selectors are defined.

```
-------------------------------------------------------------*)

  type UMSpType is ABPduType

   sorts UMSp

   opns UMSreq, UMSind : ABPdu -> UMSp

 endtype (* UMSpType *)

(*-------------------------------------------------------------
```

Communication between the AB Protocol and the UM Service is at gate **um**. This gate is, however, hidden from the view of AB Service Users. The constraints on overall behaviour, synchronised on UM communication, are therefore:

- the constraints on the behaviour of the AB Protocol (**ABProtocol**); and
- the constraints on the behaviour of the UM Service (**UMService**).

```
-------------------------------------------------------------*)

 behaviour
   hide um in
     ABProtocol [ab, um]
   |[um]|
     UMService [um]

 where

(*-------------------------------------------------------------
```

The constraints on AB Protocol behaviour, synchronised on AB communication, are:

- the constraints on behaviour at AB Service Access Points (**ABSaps**); and

- the constraints on behaviour relating communication at AB Service Access Points to that at UM Service Access Points (**ABProtocolEntities**).

```
-----------------------------------------------------------------*)

  process ABProtocol [ab, um] : noexit :=

    ABSaps [ab]
  |[ab]|
    ABProtocolEntities [ab, um]

    where

(*-----------------------------------------------------------------
```

The constraints on AB Service Access Point behaviour are:

- an AB Service Request with a non-empty Service Data Unit may occur at the source address, **0**; and
- an AB Service Indication with any Service Data Unit may occur at the sink address, **1**.

For pedagogic purposes, and as a realistic condition, an empty Service Data Unit is forbidden on a Request. Note that no constraint is put on the Service Data Unit of an Indication. It is a *property* of the specification that this will be non-empty (due to processes **ABSaps**, **ABSource**, **UMService**, and **ABSink**).

```
-----------------------------------------------------------------*)

    process ABSaps [ab] : noexit :=

      choice sdu : OctetString []
        ab ! 0 ! ABSreq (sdu) [sdu ne <>];
        ABSaps [ab]
      []
        ab ! 1 ! ABSind (sdu);
        ABSaps [ab]

    endproc (* ABSaps *)

(*-----------------------------------------------------------------
```

The constraints on AB Protocol Entity behaviour, unsynchronised, are:

- the constraints on behaviour of the data source (**ABSource**); and
- the constraints on behaviour of the data sink (**ABSink**).

In both cases, the sequence number is initialised to **0**.

```
-----------------------------------------------------------------*)

    process ABProtocolEntities [ab, um] : noexit :=

      ABSource [ab, um] (0)
```

```
    |||
      ABSink [ab, um] (0)

      where

(*-------------------------------------------------------------
```

The constraints on AB source behaviour are:

- to accept an AB Service Request with any Service Data Unit at any AB Service Access Point, causing a data message with the current sequence number and the same data to be sent via the corresponding UM Service Access Point using process **ABSend**
- to repeat AB source behaviour with the next sequence number once **ABSend** has finished.

Note that no check is made on the validity of the Service Access Point address or the Service Data Unit size. These constraints are covered in process **ABSaps**.

```
-------------------------------------------------------------*)

      process ABSource [ab, um] (seq : Bit) : noexit :=

        choice sdu : OctetString []

          ab ? addr : Bit ! ABSreq (sdu);
          ABSend [ab, um] (addr, seq, sdu)
        >>
          ABSource [ab, um] (next (seq))

        where

(*-------------------------------------------------------------
```

The constraints on AB sending behaviour are:

- to send a data message with the specified sequence number and data via the specified UM Service Access Point address; and then
- to accept a UM Service Indication with an acknowledgement at that address, such that if the sequence number in this acknowledgement is:
  - the next one, **ABSend** exits; or
  - is not the next one, **ABSend** repeats its behaviour; and
- to non-deterministically repeat the behaviour of **ABSend** for some unspecified internal reason (intended to be time-out).

```
-------------------------------------------------------------*)

        process ABSend
          [ab, um] (addr, seq : Bit, sdu : OctetString) : exit :=

          um ! addr ! ABPmess (seq, sdu);
```

```
          (
            choice newseq : Bit []

              um ! addr ! ABPack (newseq);
              (
                [newseq ne seq] ->
                  exit
              []
                [newseq eq seq] ->
                  ABSend [ab, um] (addr, seq, sdu)
              )
            []
              i; (* time-out *)
              ABSend [ab, um] (addr, seq, sdu)
          )

        endproc (* ABSend *)

      endproc (* ABSource *)

(*-------------------------------------------------------------
```

The constraints on AB sink behaviour are:

- to accept a UM Service Indication with any data message at any UM Service Access Point, such that if the sequence number in this data message is:
  - the expected one, then in *either order*:
    - the data in the message is delivered to an AB Service User at the corresponding AB Service Access Point; and
    - an acknowledgement is sent to the AB source
  - not the expected one then the data message is discarded, but an acknowledgement with the expected sequence number is sent to the AB source
- the behaviour of **ABSink** repeats with the next sequence number if the expected sequence number was received, otherwise with the same sequence number.

Note that no check is made on the validity of the UM Service Access Point address or Service Data Unit size. Their validity is a *property* of the specification (due to processes **ABSaps**, **ABSource**, and **UMService**).

```
-------------------------------------------------------------*)

      process ABSink [ab, um] (seq : Bit) : noexit :=

        choice newseq : Bit, sdu : OctetString []

          um ? addr : Bit ! ABPmess (newseq, sdu);
          (
            [newseq eq seq] ->
              (
                (
                  (
                    ab ! addr ! ABSind (sdu);
```

```
                    exit
                  )
                |||
                  (
                    um ! addr ! ABPack (next (seq));
                    exit
                  )
                )
              >>
                ABSink [ab, um] (next (seq))
              )
          []
            [newseq ne seq] ->
              um ! addr ! ABPack (seq);
              ABSink [ab, um] (seq)
          )

      endproc (* ABSink *)

    endproc (* ABProtocolEntities *)

  endproc (* ABProtocol *)

(*-------------------------------------------------------------------
```

The constraints on UM Service behaviour, unsynchronised, are:

- the constraints on behaviour for uni-directional transfer from the AB source to the sink (process **UMOneWay**, from address **0** to **1**); and
- the constraints on behaviour for uni-directional transfer from the AB sink to the source (process **UMOneWay**, from address **1** to **0**).

These are both the same since the UM Service is symmetrical.

```
-------------------------------------------------------------------*)

  process UMService [um] : noexit :=

    UMOneWay [um] (0, 1)
  |||
    UMOneWay [um] (1, 0)

    where

(*-------------------------------------------------------------------
```

The constraints on UM Service one-way behaviour are:

- to accept a UM Service Request with any AB Protocol Data Unit at the specified **from** address; and then either:
  - to deliver a UM Service Indication with the same Protocol Data Unit at the specified **to** address; or

- to lose the UM Service Request as a result of some internal decision (intended to be message loss or corruption).

```
------------------------------------------------------------*)

    process UMOneWay [um] (from, to : Bit) : noexit :=

      choice pdu : ABPdu []

        um ! from ! UMSreq (pdu);
        (
          um ! to ! UMSind (pdu);
          UMOneWay [um] (from, to)
        []
          i; (* message loss or corruption *)
          UMOneWay [um] (from, to)
        )

    endproc (* UMOneWay *)

 endproc (* UMService *)

endspec (* AlternatingBitProtocol *)
```

8 References

[1] CCITT: *Specification and Description Language*, International Consultative Committee for Telephony and Telegraphy, Z.100, Geneva, January 1988

[2] ISO: *Information Processing Systems - Open Systems Interconnection - Basic Reference Model*, International Organisation for Standardisation, ISO 7498, Geneva, October 1984

[3] ISO: *Information Processing Systems - Open Systems Interconnection - Estelle, A Formal Description Technique based on an Extended State Transition Model*, International Organisation for Standardisation, DIS 9074, Geneva, July 1987

[4] ISO: *Information Processing Systems - Open Systems Interconnection - LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, DIS 8807, International Organisation for Standardisation, Geneva, July 1987

[5] ISO: *Guidelines for the Application of Estelle, LOTOS, and SDL*, ISO/IEC JTC1/SC21/WG1 McL 65, International Organisation for Standardisation, Geneva, March 1988

[6] Halsall, F.: *Data Communications, Computer Networks, and OSI*, Addison-Wesley, 1988

[7] Leduc, G. J.: *The Intertwining of Data Types and Processes in LOTOS*, Proc. 7th. International Conf. on *Protocol Specification, Testing, and Verification*, pp. 123-136, Zurich, May 1987

[8] Najm, E.: *A Verification-Oriented Specification in LOTOS of the Transport Protocol*, Proc. 7th. International Conf. on *Protocol Specification, Testing, and Verification*, pp. 181-203, Zurich, May 1987

[9] SEDOS: *Public Report - Task C1*, Deliverable SEDOS/119, ESPRIT Project 410, Commission of the European Communities, Brussels, October 1987

[10] SPECS: *IBC View of Existing Specification Methods and Tools - Critical Evaluation*, Deliverable D4 Part II, RACE Project 2039, Commission of the European Communities, Brussels, 1986

[11] Stallings, W.: *Data and Computer Communications*, Macmillan, 1985

[12] Sunshine, C. A.: *Formal Modelling of Communication Protocols*, Proc. NPL Workshop on *Protocol Testing - Towards Proof?*, National Physical Laboratory, Teddington, March 1981

[13] Tanenbaum, A. S.: *Computer Networks*, Prentice-Hall, 1981

[14] Turner, K. J.: *LOTOS - A Practical Formal Description Technique for OSI*, Proc. International Conf. on *Open Systems*, pp. 265-280, London, March 1987

[15] Turner, K. J.: *An Architectural Semantics for LOTOS*, Proc. 7th. International Conf. on *Protocol Specification, Testing, and Verification*, pp. 15-28, Zurich, May 1987

[16] Vissers, C. A. and Scollo, G.: *Formal Specification in OSI*, University of Twente, Enschede, August 1986