

# A Challenge to LOTOS as a Formal Description Technique for Open Distributed Processing

Kerry Raymond

Key Centre for Software Technology  
Department of Computer Science  
University of Queensland  
St. Lucia QLD 4067  
AUSTRALIA

Discussion Document No. 23

18th October 1989\*

## 1 Introduction

This document evaluates the merits of LOTOS [1] for the modelling and specification of distributed systems and distributed algorithms. Some of the aspects we examine include the underlying model of LOTOS, its support for the fundamental concepts of *process* and *communication*, its expressiveness in specifying the invocation source and sequence of operations, and its ability to specify liveness properties and other temporal constraints. We also describe a comparative case study; the specification of a distributed termination algorithm in LOTOS. The reader is assumed to have some familiarity with LOTOS.

The following sections highlight significant advantages and disadvantages that we perceive in the use of LOTOS as a formal description technique for modelling and specification in ODP.

## 2 Fundamental Concepts of Process and Communication

Data types and functions cannot be defined directly in LOTOS. Instead LOTOS uses ACT-ONE (another ISO standard) for the definition of abstract data types. ACT-ONE can be used to define arbitrarily complex data (or object) types from primitive types.

---

\*printed February 27, 1990

LOTOS introduces two new concepts, *process* and *gate*. Unfortunately *process* and *gate* are not primitive types in ACT-ONE. Therefore processes and gates *cannot* be used in the definition of any other object, because LOTOS cannot define data types and ACT-ONE does not include them.

By recursive process definition, it is possible to create a group of similar processes, but no mechanism exists to have a process-valued variable or parameter, nor sets or arrays of processes. Gates are declared in static lists, and again there is no facilities to describe sets or arrays of gates. The consequence of these limitations on gates is exemplified by the Topor case study specification (discussed in Section 4).

### 3 Liveness Properties and Traces

LOTOS is capable of specifying so-called "safety" properties (i.e. bad things that must not happen). However, in order to specify "liveness" properties (i.e. good things that should happen, e.g., termination, fairness), one must be able to refer to the possible sequence of events (operations) defined by the specification. For example, in a resource allocator, one might like to specify "Every requested resource must *eventually* be allocated".

A LOTOS specification defines the possible sequence of events of a system (known as the set of *traces*). LOTOS does so by explicit sequencing and by the parallel composition of processes.

Unfortunately a LOTOS specification cannot refer to the traces defined by either itself or any other LOTOS specification.

It is possible to maintain a *history* of events in an individual LOTOS process, by defining this history as one of the process's parameters, and requiring that all "significant" events are recorded in this history. Such a history defines the sequence of significant events that *have* occurred, but cannot describe the possible events that *may* subsequently occur. Therefore a history mechanism (so named because it describes the past) cannot be used to specify the future. Nor is it possible to convert such "future" constraints into "past" constraints. For example, "Every resource allocation must have been previously requested" is not equivalent to "Every resource request must eventually be allocated".

In summary, we know of no way to express traces directly in a LOTOS specification. We can only suggest that liveness properties in LOTOS be specified in an auxiliary notation.

### 4 A Case Study

As a practical exercise, we "road-tested" LOTOS in the specification of an existing distributed algorithm. The algorithm chosen was a termination detection algorithm by Topor [2]. This case study example was chosen because it deals with a classic problem in distributed systems, for which we had not previously attempted a specification. Although reasonably precise and unambiguous, Topor did not attempt to specify this algorithm using any formal method (being more concerned with deriving algorithms through verification).

The algorithm assumes the existence of a connected graph with  $N$  processes (or processors) at its vertices. Only processes which are neighbours in the graph can communicate with one another, i.e. all communication takes place along the edges of the graph.

Initially we envisaged that a LOTOS gate would represent an edge of this graph (and the communication between the two processes at its endpoints). However the number of processes and the graph structure are parameters to the specification, and hence the number of such gates and the connectivity of LOTOS processes with these gates could not be accommodated by LOTOS's mechanism of declaring gates in static lists (both as parameters to processes and in the parallel composition of processes).

Reluctantly we were forced to specify all communication using a single gate. The requirement that communication occur only along the edges of the graph was enforced by a "communication medium" constraint process.

Appendix A is a high-level specification of termination detection which illustrates the solution we adopted. The final alternative in the definition of Proc is of particular significance.

Note that, as all processes share a single gate (*comm*) *each process must participate in each I/O operation despite being neither the sender nor receiver of the communication*. Each process must be specified to participate in (and ignore) any such communication at any time (or else deadlock may arise).

Conceptually, the behaviour of a process is not directly influenced by communication events between other processes, and so this solution is unnatural. Worse still, if the behaviour of the processes becomes very complex, then this method also becomes unwieldy. However given the constraint that the system contains an arbitrary number of processes connected in an arbitrary graph, no better solution is apparent.

As pairwise communication within an arbitrary network of processes is common in many distributed algorithms, LOTOS appears deficient in this area.

## 5 Conclusion

We have identified two major deficiencies in LOTOS which should be addressed before it is suitable for use as a formal description technique for use in ODP.

Firstly the types *process* and *gate* introduced in LOTOS would be useful if available as primitive objects in ACT-ONE, and the LOTOS process definition should include a more flexible definition of gates visible to that process.

Secondly it would be helpful to refer to the traces defined by a LOTOS specification within a LOTOS specification, in order to describe liveness properties.

We suggest asking for comment from WG1 FDT experts through a liaison statement.

## References

- [1] International Organization for Standardization, Geneva. *Information Processing Systems - Open Systems Interconnection - LOTOS, A Formal Description Technique based on the Temporal Ordering of Observation Behaviour*, July 1987.
- [2] R.W. Topor. Termination detection for distributed computations. *Information Processing Letters*, 18:33-36, 1984.

## Appendix A

### LOTOS Specification of Termination Detection

For brevity, we have excluded the ACT-ONE type definitions. Relevant definitions are:

**connections** is a set of pairs. The operation **neighbours** checks whether a given pair is in a set of **connections**.

**status** which is an enumerated type intended to record the status of a process, permitted values are **active** or **idle**.

**message** which is any message sent between processes.

Note that Procs are identified by natural numbers.

```

specification Termination [term, comm] (numproc:nat,
                                     edges:connections ) :=
(* A High level description of termination detection in
a distributed system. The system can be said to have
terminated when all the processes in the system synchronize
on a 'done' event at the 'term' gate. *)
behaviour
  GenProc [term, comm] (numproc)
  ||
  CommsMedium [term, comm] (edges)
where

process CommsMedium [term, comm] (edges:connections) :=
(* Permits communication only between processes connected
by the graph defined by 'edges'. *)
  comm?from:nat?to:nat?m:msgtype[neighbours(edges,from,to)]
  CommsMedium [comms, term] (edges)
  []
  (* terminates along with everything else *)
  term !done; exit
endproc (* CommsMedium *)

process GenProc [term, comm] (numproc : nat) :=
(* Generates numproc processes *)
  (* All processes start 'active'. *)
  [numproc > 1] -> Proc [term, comm] (numproc, active)
  || GenProc [term, comm] (numproc - 1)
  []
  [numproc = 1] -> Proc [term, comm] (numproc, active)
endproc (* GenProc *)

```

```

process Proc [term, comm] (me : nat, status : activity) :=
(* Models processes, 'me' is the process ID, 'status' is
whether the process is active or idle *)
(* active processes may become idle *)
[status = active] ->
  i; Proc [term, comm] ( me, idle)
[]
(* active processes may send a message *)
[status = active] ->
  comm !me ?id:nat !message;
  Proc [term, comm] ( me, active)
[]
(* any process may receive a message, which may make it
active *)
  comm ?id:nat !me !message;
  ( i; Proc [term, comm] ( me, status)
  []
  i; Proc [term, comm] ( me, active)
  )
[]
(* an idle process is always prepared to terminate if
everyone else agrees *)
[status = idle] -> term !done; exit
[]
(* every process is willing to allow pairs of processes
of which it isn't a member, to communicate *)
  comm ?id1:nat ?id2:nat ?m:msgtype [id1 != me and id2 != me];
  Proc [term, comm] (me, status)
endproc (* Proc *)

endspec (* Termination *)

```