

# A Comparison of LOTOS and Z for Specifying Distributed Systems \*

Iain Fogg  
Brian Hicks  
Andrew Lister  
Tim Mansfield  
Kerry Raymond

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

Discussion Document No. 18

10 May 1989<sup>†</sup>

## Abstract

Although communications underpin the development of distributed systems, the same formal description techniques may not be equally appropriate for both problem domains. We compare two formal description techniques, the standardized LOTOS and the emerging Z, to evaluate their merits for the specification of distributed systems and distributed algorithms. Some of the aspects we examine include the underlying model of both FDTs, their support for the fundamental concepts of *process* and *communication*, their expressiveness in specifying the invocation source and sequence of operations, and their ability to specify liveness properties and other temporal constraints. We also describe our comparative case study; the specification of a distributed termination algorithm in LOTOS and Z. The reader is assumed to have some familiarity with both LOTOS and Z.

## 1 Introduction

This work is motivated by our need to specify distributed information systems and related algorithms. In particular, we require formal methods to describe a distributed database in order to provide a modelling framework for subsequent research in the areas of concurrency control, weak consistency and data replication.

---

<sup>†</sup>printed February 26, 1990

\*Research funded by Telecom Research Laboratory

To this end, considerable time has been spent in evaluating the strengths and weaknesses of two major specification languages, LOTOS [3] and Z [1]. The initial attraction of LOTOS is its intended use as an ISO standard language for use in OSI networking protocols. However this was also our major concern: that in targeting communication protocols as its problem domain, LOTOS might be too specific for our research needs.

Therefore we chose to compare LOTOS with Z, based on Z's growing reputation as a general purpose specification tool which has been used with much success in other research areas.

General requirements for a specification language are the ability to:

- define precisely *what* is required at a high level,
- avoid prescribing *how* to do it when it is desired to allow the implementer the greatest freedom,
- successively refine the specification, possibly to the level of an implementation,
- simplify the task of verification,

We anticipate that work in the area of distributed information systems will have more specific needs, for example, the ability to:

- express the notion of processes and communication (in particular, message passing),
- specify safety and liveness requirements, e.g. termination, freedom from deadlock and fairness.

The following sections highlight significant advantages and disadvantages that we perceive in the use of LOTOS and Z for distributed information system specifications.

## 2 The Underlying Model

The most important difference between LOTOS and Z is their underlying model.

The LOTOS model is based upon concurrent *processes*, communicating through *gates*. The specification of a LOTOS process describes the behaviour sequence of that process. Processes which are *composed* together (as opposed to *interleaved*) must participate in all events (communications) occurring at their shared gates, providing synchronization points for the processes.

The Z model is based on *schema* definitions. Some schemas define the state of an object or system (i.e. a data type), while others define legal transitions applicable to these states (i.e. the valid operations for that type). A Z specification describes the state of the system, which must be a consequence of the successive application of operations to some defined initial state.

A LOTOS specification is therefore based on a sequence of operations, while a Z specification is based on permissible states. In this sense, LOTOS gives an "active" specification, while Z provides a "passive" specification.

### 3 Fundamental Concepts of Process and Communication

Data types and functions cannot be defined in LOTOS. Instead LOTOS uses ACT-ONE (another ISO standard) for the definition of abstract data types. Although we have reservations about the use of ACT-ONE (due to its verbosity and use of equational logic), it can be used to define arbitrarily complex data (or object) types from primitive types. However, the use of algebraic specifications is counter-intuitive and makes validation difficult.

Unfortunately, LOTOS introduces two new concepts, process and gate, for which there is no corresponding primitive type 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 gravity of these limitations on gates is exemplified by the Topor case study specification (discussed in Section 7).

While LOTOS provides fundamental support for the notion of process and communication, **Z** does not. Such concepts must be specified in **Z** before their use. Hence we were unsure of the suitability of **Z** for distributed information system specifications. However once the groundwork of specifying these concepts has been performed, models of processes and communication will be types that can be used and instantiated in the same way as any other type. In [11], we present a proposal for the specification of processes and message passing in **Z**.

A further advantage of specifying these vital concepts in **Z** is that we can model processes and messages in a way most appropriate for subsequent work in distributed information systems. While LOTOS's gate-and-process model may be well-suited to OSI networking, it does not necessarily follow that it is most appropriate to distributed information systems.

### 4 Invocation of Operations and Visibility to the Environment

When specifying operations in **Z**, it is not clear which operations are internal to the system and which are visible to (or invoked by) the environment. The usual solution in **Z** is to apply a naming or scoping convention to make this distinction.

In LOTOS, only events (communications) that occur at gates declared at the outer level are visible to the environment. However there is nothing to indicate whether these events are initiated by the environment or the system. It is tempting to believe that the process performing output (sending) is the initiator, but numerous examples in the LOTOS literature suggest that this is not always so. For example, the process may be sending data in response to a request from an external process.

### 5 Specifying Sequences of Operations

A disadvantage of **Z**'s state-based model is that it fails to provide the reader with an intuitive feel for simple operation sequences. For example, consider a simple alternating sequence of operations X and Y. The LOTOS specification captures this directly:

```

process P :=
    X ; Y ; P
endproc

```

However the **Z** version is certainly more cryptic with its use of pre- and post-conditions.

$$\textit{WhoseTurn} \hat{=} \textit{Xturn} | \textit{Yturn}$$

<i>ProcessP</i>
$\textit{turn} : \textit{WhoseTurn}$

<i>Initially</i>
<i>ProcessP</i>
$\textit{turn} = \textit{Xturn}$

<i>X</i>
$\Delta \textit{ProcessP}$
$\textit{turn} = \textit{Xturn}$
$\textit{turn}' = \textit{Yturn}$

<i>Y</i>
$\Delta \textit{ProcessP}$
$\textit{turn} = \textit{Yturn}$
$\textit{turn}' = \textit{Xturn}$

*ProcessP* is a schema which defines the state of the process. The state schema *Initially* defines the initial state of *ProcessP*. *X* and *Y* are operation schemas, which define the only legal ways in which the state of *ProcessP* can change. Note that the variable *turn'* refers to value of *turn* after the operation while the undecorated *turn* refers to the value prior to the operation.

To determine that this schema is equivalent to the LOTOS specification, one must check that:

$$\begin{aligned}
 \textit{Initially} &\Rightarrow (\textit{precondition}(X) \wedge \neg \textit{precondition}(Y)) \\
 \textit{postcondition}(X) &\Rightarrow (\textit{precondition}(Y) \wedge \neg \textit{precondition}(X)) \\
 \textit{postcondition}(Y) &\Rightarrow (\textit{precondition}(X) \wedge \neg \textit{precondition}(Y))
 \end{aligned}$$

Some effort was devoted to trying to find some new notation to improve this aspect of **Z**. However it quickly became apparent that the use of comments is a more appropriate way to aid the reader's understanding, provided that the comments are used in addition to (but not as a substitute for) formal specification.

While LOTOS provides a more intuitive specification for simple operation sequencing problems, more complex and/or conditional sequences are not easy to express in the operation-sequence style in LOTOS.

## 6 Liveness Properties and Traces

Both LOTOS and  $Z$  are capable of specifying so-called “safety” properties (i.e. what must not happen) however, in order to specify liveness properties (i.e. what must happen, for example, termination), 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”.

Specifications in both LOTOS and  $Z$  define 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. In  $Z$ , the postcondition of one operation must imply the precondition of the following operation.

Although both LOTOS and  $Z$  *define* traces,  $Z$  specifications alone can refer to the traces within the specification itself. Since a schema is just a data type, a trace in  $Z$  is simply a sequence of “state” schemas. “Operation” schemas define the mappings from each state to the next state in the trace. Hence traces can be expressed directly in  $Z$ , and can be used to express desired liveness properties [2].

$Z$ 's ability to describe traces allows us to define temporal logic operators within  $Z$ . This is of benefit since temporal logic is an excellent notation for describing liveness properties. It has the following desirable features:

- It is a compact notation.
- It is based on a sequence of states (which corresponds exactly to a trace in  $Z$ ).
- It is a formal notation, well-documented in the literature.

A temporal logic specification in  $Z$  for the resource allocation example follows with its English equivalent.

$\square(\forall p : Process \bullet$ $p \in WantResource \Rightarrow$ $\diamond p \in HasResource)$	At all times ( $\square$ ), for all processes $p$ which want a resource, eventually ( $\diamond$ ) each process $p$ will have a resource.
--	---

There are many proposed temporal operators in the literature (a comprehensive set is defined in [4]), but as yet, there does not appear to be any consensus regarding the “best” set of temporal operators. However all of the proposed operators are defined over a sequence of states, and hence are amenable to formal specification in  $Z$ .

As an aside, our experiences suggest cautious use of the “next” operators in temporal logic. These operators refer to the *immediate* next state in which a given condition holds. In a distributed system, the overall trace of operations is an interleaving of the traces of individual processes (subject to certain restrictions). Hence two successive events in the trace of an individual process may be not be successive events in the overall trace. If “next” operators are used to express temporal requirements of the traces of individual processes, then these constraints will not necessarily hold in the overall trace and hence not be useful in proving properties at the system level.

Therefore we recommend that the use of temporal operators in distributed systems specification be restricted to such operators as  $\square$  (“always”),  $\diamond$  (“eventually”) and  $\triangleleft$  (Lamport’s precedence operator [5]) which apply to both the microscopic view of an individual process (or other object) and the macroscopic view of systems and subsystems.

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. 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 *will* 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".

A more detailed discussion of liveness constraints in LOTOS may be found in [6].

## 7 A Comparative Case Study

As a practical exercise, we "road-tested" both LOTOS and Z in the specification of an existing distributed algorithm. The algorithm chosen was a termination detection algorithm by Topor [12]. 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 a network of processes is common in many distributed algorithms, LOTOS appears unsuited for our work.

In contrast to the unanticipated difficulty in developing a satisfactory LOTOS specification of Topor's algorithm, the development of a specification in **Z** progressed without incident.

Our high level specifications of termination detection in LOTOS and **Z** are reported in [8, 7], while the lower level specifications of Topor's algorithm are contained in [10, 9].

## 8 Conclusion

While the problems which arose in the Topor case study remain unresolved, we have grave concerns about the suitability of LOTOS for distributed information systems work. Also LOTOS's apparent inability to specify traces (and hence liveness properties) within a specification further limits its value to us. Doubtless LOTOS *could* be altered/extended/augmented to remove these difficulties, but doing so would negate what is perhaps LOTOS's major attraction, i.e. its status as an ISO standard.

For the work we will be undertaking, **Z** has some disadvantages:

1. no built-in notion of process or communication,
2. simple operation sequences are not immediately obvious,
3. uncertainties regarding invocation of operations and visibility of operations to the environment.

However we believe that these drawbacks can be resolved in the following ways:

1. develop "library" schemas for processes and communication (as in [11]),
2. use comments or explanatory text to increase readability,
3. adopt a naming or scoping convention.

Therefore for specifications of distributed systems and algorithms, **Z** has many advantages and no insurmountable disadvantages. For these reasons, we have chosen **Z** as the more appropriate formal description technique for distributed information systems research.

## Acknowledgements

We thank Dr Roger Duke, Mr Graeme Smith and Dr Ian Hayes for their advice about **Z**, and Dr Bob Pascoe (University of the Northern Territory) and Prof Ken Turner (University of Stirling, Scotland) for their LOTOS comments.

## References

- [1] J.-R. Abrial, S. Schuman, and B. Meyer. Specification language. In R. McKeog and A. Macnaghten, editors, *On the Construction of Programs: An Advanced Course*, pages 343-410. Cambridge University Press, 1980.

- [2] Roger Duke and Graeme Smith. Temporal logic and  $Z$  specifications. In *Proceedings of 12th Australian Computer Science Conference*, Wollongong, Australia, February 1989.
- [3] 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.
- [4] Fred Kroeger. *Temporal Logic of Programs*, volume 8 of *Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1987.
- [5] Leslie Lamport. What good is temporal logic? *Information Processing 83*, pages 657–668, 1983.
- [6] Tim Mansfield and Kerry Raymond. Expressing liveness constraints in LOTOS, March 1989. University of Queensland, Dept of Computer Science, DSL-DD10.
- [7] Kerry Raymond and Tim Mansfield. A high-level specification of distributed termination in  $Z$ , January 1989. University of Queensland, Dept of Computer Science, DSL-DD3.
- [8] Kerry Raymond and Tim Mansfield. High level specification of termination detection in a distributed system in LOTOS, January 1989. University of Queensland, Dept of Computer Science, DSL-DD1.
- [9] Kerry Raymond and Tim Mansfield. A specification in  $Z$  of Topor's distributed termination detection algorithm, January 1989. University of Queensland, Dept of Computer Science, DSL-DD4.
- [10] Kerry Raymond and Tim Mansfield. A specification in LOTOS of Topor's distributed termination detection algorithm, January 1989. University of Queensland, Dept of Computer Science, DSL-DD2.
- [11] Kerry Raymond and Tim Mansfield. Specification of a distributed system, May 1989. University of Queensland, Dept of Computer Science, DSL-DD8.
- [12] 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 *)

```

## Appendix B

### Z Specification of Termination Detection

Let  $N$  be the number of processes, numbered 1 to  $N$ . A Process is considered to be either **active** or **idle**.

$Id \hat{=} 1..N$   
 $activity \hat{=} \{active, idle\}$

*Process*

---

$status : activity$

---

There are  $N$  Processes, arranged in a graph.

*Graph*

---

$node : Id \rightarrow Process$  { the  $N$  processes }  
 $edges : Id \leftrightarrow Id$  { the graph }

---

{ there are no trivial cycles in the graph }  
 $\forall i : Id \bullet (i, i) \notin edges$   
{ the graph is connected }  
 $\forall i : Id \bullet edges^*(i) = Id$   
{ edges are listed one way only }  
 $edges \cap edges^{-1} = \{ \}$

---

We get some arbitrary graph and initialize the Graph to make all nodes active.

*Graph<sub>INIT</sub>*

---

*Graph*  
 $e? : Id \leftrightarrow Id$

---

$edges = e?$   
 $\forall i : Id \bullet node(i).status = active$

---

Active processes send messages

*SendMsg*

---

$\Delta Graph$   
 $to?, from? : Id$

---

{ Is the from node active? }  
 $node(from?).status = active$   
{ are the from and to nodes connected? }  
 $(from?, to?) \in edges \cup edges^{-1}$   
{ send the message }  
 $(node' = node \vee$   
 $node' = node \oplus \{to? \mapsto (active)\})$   
 $edges' = edges$

---

An active process can become idle at any time.

*Passivate*

$\Delta Graph$   
 $victim? : Id$

$node(victim?).status = active$   
 $node' = node \oplus \{victim? \mapsto idle\}$

The system terminates when all the nodes are idle.

*Termination*

$\exists Graph$

$\forall i : Id \bullet node(i).status = idle$

