Specifying Distributed Algorithms in LOTOS

Mazen Haj-Hussein and Luigi Logrippo
Protocols and Software Engineering Research Group
University of Ottawa, Computer Science Department
Ottawa, Ont., Canada K1N 9B4
E-mail: lmlsl@acadvm1.uottawa.ca

ABSTRACT. LOTOS is a formal description technique that was conceived for the specification of the services and protocols of the Open Systems Interconnection (OSI). It has, however, a much more general scope of application. In this paper, we present a framework for the specification of a wide class of distributed algorithms in LOTOS. In particular, we provide a general method for specifying network topologies. We illustrate our method with two examples: an election algorithm in a distributed network, and a single-initiator spanning tree construction algorithm. Finally, we show how such specifications can be validated by using a "random walk" technique, with the help of an interpreter.

1. INTRODUCTION

A quick scan of papers written in the area of distributed algorithms will show that no uniform presentation of such algorithms has been adopted in the literature. A number of formalisms, quasi-formalisms, and non-formalisms are used by different authors. Among the most common are: common English, mathematical notations, programming language notations, and variants of CSP. In some cases these notations are adequate. For complex algorithms, however, it might not be completely clear exactly how the algorithm functions. Lack of precision and formalism can be supplemented to some extent by redundancy, at the expense of somewhat wordy descriptions. In spite of this redundancy, some ambiguities may persist. In any case, if it is desired to have a running model of the algorithm, it is necessary to program it in some programming language.

LOTOS is a formal description technique that was conceived for the specification of the services and protocols of the Open Systems Interconnection (OSI)[ISO]. It is an ISO standard, IS 8807 [ISO1]. Soon after its introduction, other uses were found for LOTOS: among others, the specification of telephone systems [FLS]. In this paper, we show how LOTOS can be used for the specification of a large class of distributed algorithms. Because LOTOS is executable (at least under certain conditions), execution and testing of the algorithm are possible. Because LOTOS has a formal basis, a framework for verification also exists.

In order to prove our point, in this paper we show how to specify in LOTOS two classical distributed algorithms: electing a leader in a unidirectional ring where any number of nodes may independently start the algorithm, and constructing a single initiator spanning tree in a strongly connected network with bidirectional (full duplex) links. As we illustrate these examples, we also develop some principles showing how other algorithms of the same type can be specified in LOTOS. We show how to specify the network (i.e. the nodes and the links) and the algorithm, i.e. the whole environment. We conclude by presenting a technique by which an algorithm specified

in LOTOS can be validated by using an interpreter.

This paper intends to have a tutorial component. Researchers in the area of distributed algorithms should be able to specify their algorithms in LOTOS after studying a general LOTOS tutorial, and this paper.

2. LOTOS AND LOTOS TOOLS.

We do not intend to provide a tutorial on the language LOTOS in this paper. At least two tutorials have been or are being published in journals [BBr][LFH], and several other tutorials have enjoyed some degree of distribution. A tutorial is also included in [ISO1]. Strictly speaking, this paper is directed towards people who already have acquired some knowledge of the language. However, since the concepts of the language are in part well-known, we provide a brief introduction below to allow readers who have previously studied similar languages to understand the main points of this paper.

LOTOS, the Language of Temporal Ordering Specifications, is one of the most precisely defined languages in use today. Its static semantics are defined by an attributed grammar, while its dynamic semantics are based on algebraic concepts. LOTOS is made up of two components: a *data type* component, which is based on the algebraic specification language ACT ONE [EM], and a *control* component, which is based on a mixture of Milner's CCS [Mil] and Hoare's CSP [Hoa]. Most of the theoretical framework of the control component, and especially the concept of *internal action* are based on Milner's work. In particular, non-determinism is modelled by internal actions (denoted by the letter **i**) as in [Mil], rather than by adding special operators as in [Hoa]. The rendezvous semantics follow Hoare's "multi-way synchronization" concept, by which all processes that share a gate which is in a common synchronization set must participate in synchronization on that gate. Actions, however, can be transformed into internal actions by *hiding* them. In this way, further participation in the action of processes outside the **hide** is prevented.

LOTOS dynamic semantics for the *control* component are expressed in operational terms by inference rules as in [Mil], and the operators were chosen in such a way that they enjoy a rich set of algebraic properties, similar to those of [Mil]. Therefore, the language is at the same time "executable" (at least to a certain extent, by virtue of the operational semantics), and amenable to proof techniques (by virtue of the algebraic properties).

The language is purely recursive in nature. Side effects are only those that are produced by process synchronization. It supports process parameterization, for both value and gate parameters. The basic element of the *control* part of a LOTOS specification is the *action offer,* where a process declares itself ready to synchronize with other processes and establish one or more values. For example, *g!3* states that the process is offering to synchronize on the specific value *3* with other processes, on gate *g*. *g?x:integer* states instead that the process is ready to synchronize on any integer value with other processes, on gate *g*. Thus, two processes containing these two complementary action offers may be able to synchronize, and if they do, the second process gets the value *3* for the variable *x*. "Multiple" and "bidirectional" action offers are also possible, such as *g !3 ?x:integer*, where the process declares itself ready to simultaneously synchronize on *3* and any integer, still on gate *g*. Such a process would be able to synchronize with another process offering at the same time

actions such as *g !3 !4* or *g ?x:integer !4* or even *g ?y:integer ?z:integer* (in the latter case, the value for *x* and *z* respectively in the first and second process would have to be determined by another process participating in the same synchronization; this process could be the environment). *Selection predicates* can establish conditions for the executability of an action. For example, *g ?x:integer [x gt 3]* means that the process is ready to accept only an *x* greater than *3*. Similarly, an action can be subject to a guard, e.g. *[y gt 4] -> g!y* means that the guarded action is offered only if *y* is greater than *4*. Action offers can be combined by the use of several operators. The most important are: **;** (prefixing a process or an action by another action), [] (choice), |[A]| (parallel execution with synchronization on gates in set A), || (parallel execution with synchronization on all gates, or *dependent parallel composition*), ||| (parallel execution in *interleave*, or *independent parallel composition*), **hide** (hiding of gates), >> (sequential composition of processes), and [> (*disable*, modelling an interruption).

The *data* part (ACT ONE) supports parameterized types, type renaming, and conditional rules. The reader will note that this part of the language does not enjoy an elegant notation, nor does it have a sufficient number of useful shorthands or built-in types. This fact is currently being addressed, and enhancements to the standard are being planned.

A "graphic" version of the language is in advanced stage of standardization within ISO and CCITT.

Because of the fact that LOTOS is (partially) executable, a specification is effectively a "fast prototype" of the entity specified, thus it is possible to exercise a specification of a complex system at the design stage. The two LOTOS interpreters in existence today are described in [LOBF] [GHL][VVD]. Other tools existing in various prototypical form are: symbolic expanders [QP] and translators into C [MM]. In this paper, we insist on an executable LOTOS style, to take full advantage of our interpreter, which is the tool discussed in [GHL].

It is possible to carry out in LOTOS proofs of correctness such as the ones found in [Mil] [Hoa], and the proof methods are similar to those found in these references. For the data part, the proof techniques developed in the area of abstract data types [EM] are applicable. Of course, the challenging aspect is to be able to prove properties of systems of realistic size. To this end, computer-assisted verification tools are being envisaged. Some such tools appropriate for small specifications are documented in [BC] [GS].

3. SPECIFICATION OF NETWORK TOPOLOGY

3.1 Conceptual Model and Assumptions

The network model that we have used in our examples is a *point-to-point model* where the underlying communication network is composed solely of directed communication links.

A distributed system is constructed from a set of named *nodes* that are interconnected by a set of unidirectional *links*. Formally, the network is viewed as a *strongly-connected* digraph **G=(N,L)** where

N is the set of *nodes*, each representing an entity in the distributed system, and

L is the set of directed edges or *links*, each representing a directed communication channel between two nodes.

Nodes are the active elements in our distributed system model. A node is a concurrent entity that communicates with other nodes by message passing. The set of nodes in the system implements the distributed algorithm. Note that although the term *process* is often used as a synonym for *node* in the literature on distributed algorithms, we reserve it to designate LOTOS processes. As we shall soon see, there are processes associated with nodes, there are others associated with links, etc. The links are represented by pairs (n_i,n_j) where n_i is the source node and n_j is the destination node (Fig. 1). If $(n_i,n_j) \in \boldsymbol{L}$, n_j is said to be an *out-neighbour* of n_i, and n_i is said to be an *in-neighbour* of n_j. If one of (n_i,n_j) or (n_j,n_i) is a link, n_i and n_j are said to be *neighbours*.

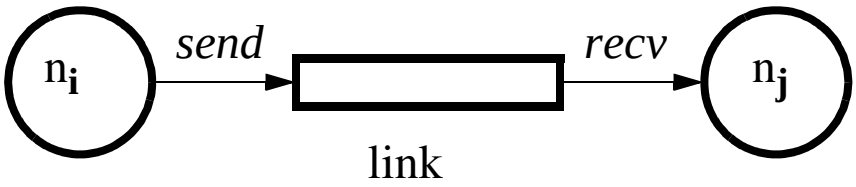

Figure 1

The following assumptions form the basis of the model that we have used in our examples.

• **A1 [Local Orientation]** : A node can distinguish between its (in- and out-) neighbours, and can detect from which in-neighbours a received message was sent. But a node has no knowledge of the other nodes' links (i.e. no global knowledge about the network).

• **A2 [Lossless Communication]** : A message sent is eventually received. Note that this assumption does not imply the existence of any bound on the transmission delay; it only states that a message will arrive after finite delay without corruption.

• **A3 [One-Place Links]** : All links between nodes are one-slot queues; that is to say, a node cannot send a message through a link where there is still a message to be received.

• **A4 [Local non-shared memory]** : Each node has a local non-shared memory, which is assumed to be of unbounded capacity.

• **A5 [Unique Identifiers]** : Each node $n_i \in \boldsymbol{N}$ has a unique key $key(n_i)$ and a unique node identifier $name(n_i)$.

For simplicity, we do not include link or node failure in our examples. Also we consider only asynchronous systems, i.e. where nodes communicate by links and there is no central clock.

3.2 Specification of Network Configuration in the Data Part

The network configuration is defined in the data part of the LOTOS specification as a sort called *Network*. In this section we describe all the sorts and operations that are needed to specify arbitrary networks.

Name: Is the sort that distinguishes node identifiers (e.g. *n1*, *n2*, etc.).

Key: Is the sort for node keys.These are integers, represented in a notation defined by us in the data part in order to simplify the standard LOTOS notation for integers. For example: *pos(3,2,4,2)* represents +3242, while *neg(0,1,5,4)* represents -154.

LinkList: The sort of a list of elements of sort *Name*. It is used to identify the out-neighbours of a node. Values of this sort have the form:

NIL2	for an empty list
Name ++ LinkList	for a nonempty list

Example: *n1 ++ (n2 ++ (n3 ++ NIL2))*

Node: The sort for the local knowledge of the nodes. It contains the node name, node key, and out-neighbours. Values of this sort have the form:

node(Name, LinkList, Key)

Example: *node(n1, n2 ++ (n3 ++ (n4 ++ NIL2)), pos(3,2,4,2))*

means that *n1* has *n2*, *n3*, and *n4* as out-neighbours and has the key 3242.

Network: Is the sort for a network configuration, that is a list of the nodes in the network with their local knowledge (i.e. list of elements of sort *Node*). Values of this sort have the form:

NIL3	for an empty network
Node ++ Network	for a nonempty network

For example the network configuration of figure 2 can be specified as:

```
 node(n1, n2 ++ NIL2, pos(2,0,5)) ++
(node(n2, n3 ++ NIL2, pos(1,0,0,5)) ++
(node(n3, n1 ++ NIL2, neg(5)) ++
NIL3))
```

and the network configuration of figure 3 can be specified as:

```
 node(n1, n2 ++ (n3 ++ NIL2),          pos(6) ) ++
```

```
(node(n2, n1 ++ (n3 ++ (n4 ++ NIL2)),   pos(5) ) ++
(node(n3, n1 ++ (n2 ++ (n4 ++ NIL2)),   pos(8) ) ++
(node(n4, n2 ++ (n3 ++ NIL2),           pos(2) ) ++
 NIL3 ) ) )
```

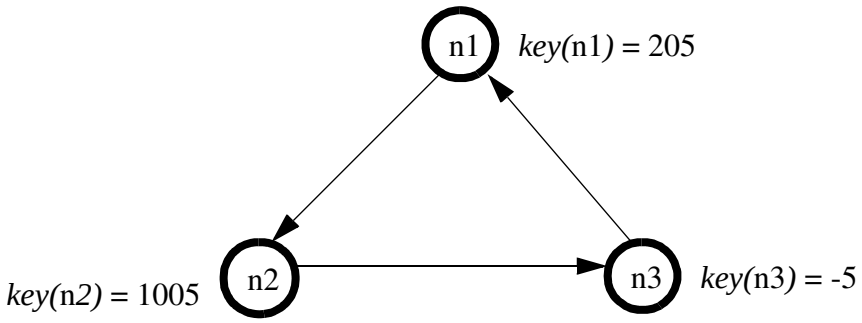

Figure 2

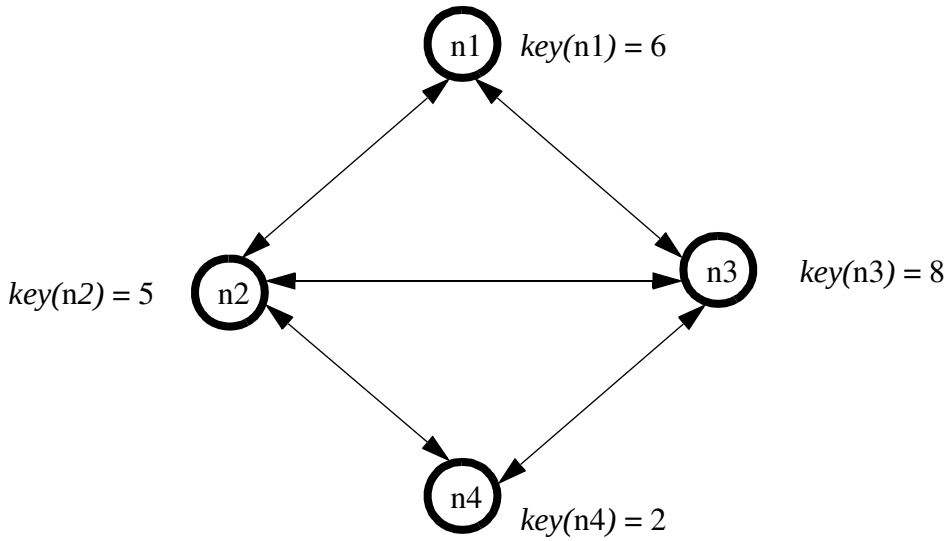

Figure 3

Several common list (set) operations have also been defined, to be used on *LinkList* and *Network* sorts, such as:

head(List)	returns the first element in List
tail(List)	returns the tail of List (excluding the first element)
is_null(List)	returns *true* if List is empty (i.e. if List equal NIL)
List \ Elt	removes the first occurrence of Elt from List

List v List gives the union of two lists
List ^ List gives the intersection of two lists
List + List gives the concatenation of two lists

Other important operators are:

name(Node): returns the name identifier of Node
key(Node): returns the key of Node
out(Node): returns the out-neighbours of Node

Examples:

name(node(n2, n1 ++ (n3 ++ (n4 ++ NIL2)), pos(5))) = n2
key(node(n2, n1 ++ (n3 ++ (n4 ++ NIL2)), pos(5))) = pos(5)
out(node(n2, n1 ++ (n3 ++ (n4 ++ NIL2)), pos(5)))) = n1 ++ (n3 ++ (n4 ++ NIL2))

3.3 Network Composition in LOTOS

A node can send a message to an out-neighbour node through the link between them by synchronizing on gate *send*. Similarly a node can receive a message from an in-neighbour through the link between them by synchronizing on gate *recv*.

We assume that a node identifier n_i of sort *Name* has been defined in the data part of the specification for each node in the distributed system. The local knowledge of n_i will be passed as a parameter when a process is instantiated to represent the associated node. As an example, in the network of Figure 2, n_i can send a message *M* to an out-neighbour node n_j using the following LOTOS action:

$$send\ !n_i\ !n_j\ !M$$

and n_j can receive message *M* from the in-neighbour n_i using the action:

$$recv\ !n_i\ !n_j\ ?Msg: Message$$

in other words, both sending and receiving nodes must agree on the link on which they communicate, which is identified by the nodes' names.

Here are some interesting examples of use of these actions:

$$send\ !n_i\ ?n:Name\ !M\ [is\_member(n, out(Ninfo_i)\backslash n_j)]$$

where $Ninfo_i$ is a variable of sort *Node* containing the local knowledge of node n_i. Node n_i sends the message *M* to any out-neighbour node, except n_j (the selection predicate in square brackets establishes a condition on the *n* that can be accepted).

recv ?n:Name !n_j ?Msg:Message

Node n_j is ready to receive any message from any in-neighbour node

recv ?n:Name !n_j !M

Node n_j is ready to synchronize on a specific message *M* with any in-neighbour node.

If a message needs to be broadcast to a set of out-neighbours, the following LOTOS process can be used:

```
process broadcast[send](n:Name, Msg:Message, Neighbours:LinkList): exit:=
        [not(is_null(Neighbours))] ->
         (
            send !n !head(Neighbours) !Msg; exit
           |||
            broadcast[send](n, Msg, tail(Neighbours))
         )
        []
        [is_null(Neighbours)] -> exit
endproc
```

The following example enables Node 3 to broadcast the message *M* to all its out-neighbours, then behave as *B*:

broadcast[*send*](n3, M, out(Ninfo3)) >> B

Figure1 can be described in LOTOS as:

```
      ((Nd[send, recv](Ninfo_i)
   |||
     Nd[send, recv](Ninfo_j))
   ||
     link[send,recv](name(Ninfo_i), name(Ninfo_j))
```

Where Nd[*send,recv*](Ninfo$_k$) is the behaviour of n_k, and process *link* has the following form:

```
process link[send,recv](n_i, n_j: Name) : noexit
      send !n_i !n_j ?Mes:Message;
      recv !n_i !n_j !Mes;
      link[send,recv]
endproc
```

link will transfer every message sent by n_i to n_j. As mentioned above, a link cannot accept any messages from its in-node if there is a message that was not yet received by the out-node.

To specify composition of nodes of a network with all the necessary links, we can use the following LOTOS process, showing the set of nodes in dependent composition with the set of links:

```
choice Nt:Network []
   (
      Nodes[send,recv](Nt)
   ||
      all_links[send,recv](Nt)
   )
```

where:

Nt is the network configuration, *Nodes[send,recv](Nt)* is the interleaved compositions of all nodes in *Nt*, and *all_links[send,recv](Nt)* is the interleaved compositions of all possible links specified in *Nt.* The **choice** statement chooses a network configuration among all possible ones.

To specify the composition of the nodes and links in the network, we proceed as follows. First of all, we use recursion to create an instance of process *Nd* for each node.

```
process Nodes[send,recv](Nt: Network): noexit:=
      [not(is_null(Nt))] ->
      (
         Nd[send,recv] (head(Nt))
         |||
         Nodes[send,recv] (tail(Nt))
      )
endproc
```

Next, we must create an instance of process *link* for each link. This is done in two steps. First, we iterate over all nodes.

```
process all_links[send,recv](Nt: Network): noexit :=
         [not(is_null(Nt)) ->
         (node_links [send,recv] (name(head(Nt)), out(head(Nt)))
         |||
          all_links[send,recv](tail(Nt))
         )
endproc
```

(note that this intermediate process could be avoided at the price of some extra functionality in the data part: this is left as an exercise for the reader). Then we iterate over each node to create instances of process *link* for each outgoing link of the node

```
process node_links [send,recv](Nname:Name, Neighbours:LinkList): noexit:=
      [not(is_null(neighbours))] ->
         (
            link [send,recv](Nname,head(Neighbours))
            |||
            node_links [send,recv](Nname,tail(Neighbours))
         )
endproc
```

We have already seen process *link*:

```
process link[send,recv](From,To:Name): noexit:=
      send !From !To ?Msg:Message;
      recv !From !To !Msg;
      link[send,recv](From,To)
endproc
```

Each node in the network executes the same algorithm:

```
process Nd[send,recv](Ninfo:Node): exit(Result):=
      Algorithm[send,recv](Ninfo, <List of Local non-shared Variables>)
endproc
```

The initial values of local non-shared variables for a node have to be passed as process parameters <List of Local non-shared Variables> , initially when executing the Algorithm process, or when another state process is called.

One of our assumptions is that the network is strongly connected. We can ensure that this is true by using the following construct:

```
   choice Nt:Network []
      [is_strongly_connected(Nt)] ->
      (
         Nodes[send,recv](Nt)
         ||
         all_links[send,recv](Nt)
         )
```

This behavior expression selects a strongly connected network and then sets up a process structure for it.The boolean operation *is_strongly_connected* should be defined in the data part of the specification.

4. EXAMPLE: ELECTION ALGORITHM IN UNIDIRECTIONAL RING

Sections 3.2 and 3.3 show how a network configuration can be specified in LOTOS. The specifi-

cation method is general and applies to any algorithm based on a model similar to the one described in Section 3.1. The examples given in this section and in the next use this specification method.

One of the basic problems in distributed computing is the *election of a leader*: initially, all nodes are in the same state (say, *available*); an algorithm is wanted which, once executed, will move exactly one node in a specified state (say, *leader*) and all others in a different one (say, *processor*). Any number of nodes may independently start the process of electing a *leader*. Note that there is no a priori restriction on which node should become the *leader*. Since every node has a unique key, thus the general idea adopted for such algorithms is that the node with the highest key value become the *leader*.

We now consider the problem of *electing a leader in a unidirectional ring* where every node in the network has exactly one out-neighbour and one in-neighbour in the form of a ring. We specify in LOTOS the version of the algorithm presented in [CR] . Other election algorithms can be found in [AG][GM][KRS].

4.1 Informal Description of the Algorithm

The possible states of a node while executing the algorithm are: *initiator*, *available*, *candidate*, *processor*, or *leader*.

We want to elect a *leader*, which must be the node with the highest key value. An *initiator* sends its key to its out-neighbour and becomes a *candidate* for leadership. Any *available* node wakes up upon receiving a message (a key value, say *k*) from its in-neighbour. If *k* is less than its own key then the node becomes a *candidate* and sends its own key forward. If *k* is greater than its own key then obviously the node is not a good candidate for leadership. In this case the node becomes a *processor* for the candidate with the key *k* and passes forward the key *k*.

A processor *P* for a candidate with key *m*, when it receives a key *k* from its in-neighbour, it compares *k* with *m*. If *k* is greater than *m* then *P* passes forward *k* and becomes a processor for the candidate with the key *k*. If *k* is less than *m* then it is ignored. If *k* is equal to *m* then *P* knows that the candidate with the key *m* (or *k*) is the *leader* and terminates the execution.

A candidate *C*, with a key *n*, when it receives a key *k* from its in-neighbour also compares it with *n*. If *k* is greater than *n* then *C* passes forward *k* and becomes a processor for the candidate with key *k*. If *k* is less than *n* then it is ignored. If *k* is equal to *n* then *C* has the largest key in the network, so *C* passes forward *n* to its out-neighbour and becomes a *leader*.

A *leader* has to wait until it receives its key back, that is, it terminates after all nodes have terminated their execution.

4.2 Formal Description of the Algorithm

The formal description of the algorithm in LOTOS is given below. It must be included within the description of the system configuration, given in Section 3. In this example, every state in the algo-

rithm is represented as a LOTOS process.

(* Initially a node can be an *initiator* or *available*. A node can become an *initiator* by executing an internal decision, represented by the internal action **i** *)

```
process Algorithm[send,recv](N:Node):noexit:=
      i; initiator[send,recv](N)
 []
      available[send,recv](N)
endproc
```

(* If *initiator* then send own key and become a candidate *)

```
process initiator[send,recv](N:Node):noexit:=
      send !name(N) !head(out(N)) !key(N);
      candidate[send,recv](N)
endproc
```

(* *Available*: receive key and, if smaller than own key, propagate own key and become candidate, else propagate received key and become processor *)

```
process available[send,recv](N:Node): noexit:=
      recv ?Right_node:Name !name(N) ?Key_value:Key;
      (
      [key(N) > Key_value] ->
            send !name(N) !head(out(N)) !key(N);
            candidate[send,recv](N)
      []
      [key(N) < Key_value] ->
             send !name(N) !head(out(N)) !Key_value;
             processor[send,recv](N,Key_value)
      )
endproc
```

(* Processor: accept key and, if greater than current key, propagate received key and remain processor; if smaller than current key, do nothing; if equal to current key, propagate received key and stop *)

```
process processor[send,recv](N:Node, Current_Key:Key): noexit:=
    recv ?Right_node:Name !name(N) ?Key_value:Key;
    (
    [Key_value > Current_Key] ->
        send !name(N) !head(out(N)) !Key_value;
        processor[send,recv](N,Key_value)
    []
    [Key_value < Current_Key] ->
        processor[send,recv](N,Current_Key)
    []
    [Key_value == Current_Key] ->
        send !name(N) !head(out(N)) !Key_value;
        stop
    )
endproc
```

(* Candidate: receive key and, if greater than own key, propagate received key and become processor; if smaller than own key, do nothing; if equal to own key, propagate received key and become *leader* *).

```
process candidate[send,recv](N:Node): noexit :=
    recv ?Left_proc:Name !name(N) ?Key_value:Key;
    (
    [Key_value > key(N)] ->
        send !name(N) !head(out(N)) !Key_value;
        processor[send,recv](N,Key_value)
    []
    [Key_value < key(N)] ->
        candidate[send,recv](N)
    []
    [Key_value == key(N)] ->
        send !name(N) !head(out(N)) !Key_value;
        leader[recv](N,Key_value)
    )
endproc
```

(* *Leader:* receive expected key and stop *)

```
process leader[recv](N:Node, Key_value:Key) : noexit:=
    recv ?Right_node:Name !name(N) !Key_value;
    stop
endproc
```

5. EXAMPLE: SINGLE INITIATOR SPANNING TREE CONSTRUCTION

Consider the problem of *constructing a spanning tree* **T** of a graph **G**. In a distributed environment,

the fact that **T** has been constructed means that every node knows which of its neighbours in **G** are also neighbours in **T**.

The following algorithm has a single *initiator* and works for any connected network with bidirectional *full-duplex* links. This single *initiator* can be the elected leader from executing an election algorithm. Other, more sophisticated, algorithms can be found in [GHS][Hum][Se].

5.1 Informal Description of the Algorithm

This algorithm has the following main states: *initiator* (only one node can be in this state), *available*, *parent*, *shutting_down*, and *terminated*.

Initially all nodes are in *available* state, except the *initiator*. The *initiator* will be the root of the resulting spanning tree, initially it broadcasts a request for all its neighbours to be its children (*are_you_my_child*), and becomes a *parent*.

An *available* node awakens upon receiving a request for becoming a child. It replies with an acceptance (*yes*), it requests to the remaining neighbours to be its children (*are_you_my_child*), and becomes a *parent*.

A *parent* node will reply *no* to all requests of becoming a child. It waits for a response from all the neighbours. Its children will be the nodes that reply *yes* to its request. It then becomes a *shutting_down* node, where it waits for all its children to terminate their construction of their subspanning trees. When a child has completed the construction of its own tree, it sends a message to its *parent* that it is terminating. That is to say, all nodes terminate the algorithm before their *parent* and the *initiator* terminates last.

5.2 Formal Description of the Algorithm

Since we have a single *initiator*, then the overall composition of nodes will be slightly different with respect to the previous example because initially all nodes except the *initiator* should be in *available* state.

Here is one way of doing it:

```
choice Nt:Network , Initiator:Name []
  (
    Nodes[send,recv](Nt, Initiator)
  ||
    all_links[send,recv](Nt)
  )
```

The *all_links* process that sets up all the links in the network is the process defined in section 3.3.

The *Nodes* process has an extra parameter, the name of the *initiator*

```
process Nodes[send,recv](Nt: Network, Initiator:Name): noexit:=
    [not(is_null(Nt))] ->
    (
      Nd[send,recv] (head(Nt), Initiator)
      |||
      Nodes[send,recv] (tail(Nt), Initiator)
    )
endproc
```

Every node will be in *available* state except the *initiator*:

```
process Nd[send,recv](Ninfo:Node, Initiator:Name): noexit(Result):=
      ([name(Ninfo) == Initiator] ->
        Algorithm[send,recv](initiator,name(Ninfo),true,root,NIL2,out(Ninfo),out(Ninfo)))
      []
      ([name(Ninfo<> Initiator] ->
         Algorithm[send,recv](available,name(Ninfo),false,root,NIL2,out(Ninfo),out(Ninfo)))
endproc
```

The local variables of a node are:

S:State:	The state of the node
Nname:Name:	The node name
My_start:Bool:	*true* for *initiator*, and *false* for non-*initiator* node
My_parent:Name:	Name of the *parent* node in the spanning tree; the *initiator*'s *parent* is a special node that we have called *root*
Term_set:LinkList	The set of children that have terminated
My_children:LinkList:	The set of neighbours that have agreed on being children or have not yet replied
Expected_res:LinkList:	The set of neighbours that have not yet replied

In the previous example we have used different processes for different states. In this example we use only one process *Algorithm* that behaves depending on the state *S* (this is a classical example of state-oriented specification as defined in [VSV]). The *Algorithm* process is recursive; when a node changes its state, it calls *Algorithm* with the new state and the modified local variables. This process has the following format:

```
process Algorithm[send, recv](<local non-shared memory>): noexit:=

      [S == initiator] -> B1
      []
      [S == available] -> B2
      []
      ..
```

..
[]
[S == shutting_down] -> B_n
endproc

Here we show the top level only of the algorithm. An important process, which is not shown below, is *process_child_list*. The function of this process is to determine the next state of the caller node. If *Expected_res* is empty and there are still children that have not yet terminated then the next state is *shutting_down*, however if *Expected_res* is empty and all children nodes have terminated then the node terminates by notifying its parent. If *Expected_res* is nonempty and the caller is a *parent* then it stays a *parent* otherwise if the caller is *available* then it becomes a *parent* after broadcasting the request to all its neighbours, except its parent node.

process Algorithm[*send,recv*](
S:State,
Nname:Name,
My_start:Bool,
My_parent:Name,
Term_set:LinkList,
My_children:LinkList,
Expected_res:LinkList): **noexit**:=

(* An *initiator* broadcasts and becomes a *parent*, then repeats the algorithm *)

[S == initiator] ->
(
broadcast[*send*](Nname,are_you_my_child,My_children)
>>
Algorithm[*send,recv*](
parent,
Nname,
My_start,
My_parent,
Term_set,
My_children,
Expected_res)
)

(* An *available* node receives a request to become a child, answers positively, and proceeds to find its own children. Before doing this, it takes note of the name of its father and it removes it from the list of children and the list of nodes from which an answer is expected. It will change state by executing *process_child_list* *)

[]
[S == available] ->
(

```
            recv ?From:Name !Nname !are_you_my_child;
            send !Nname !From !yes;
            process_child_list[send,recv](
                  available,
                  Nname,
                  My_start,
                  From,
                  Term_set,
                  My_children \ From,
                  Expected_res \ From)
      )
```

(* A *parent* node will answer negatively a request to be a child, and repeats the algorithm *)

```
      []
      [S == parent] ->
      (
            recv ?From:Name !Nname ?Mes:Message;
            (
            [Mes == are_you_my_child] -> (
                  send !Nname !From !no;
                  Algorithm[send,recv](
                  parent,
                  Nname,
                  My_start,
                  My_parent,
                  Term_set,
                  My_children,
                  Expected_res)
                  )
```

(* A *parent* node that has received a *yes* will execute *process_child_list*, after removing the node from which the reply was received from the list of nodes from which a reply was expected *)

```
            []
            [Mes == yes] -> (
                  process_child_list[send,recv](
                  parent,
                  Nname,
                  My_start,
                  My_parent,
                  Term_set,
                  My_children,
                  Expected_res \ From)
                  )
```

(* A *parent* node that has received a *no* will execute *process_child_list*, after removing the node from which the answer was received from the list of children nodes, and from the list of nodes from which an answer was expected *)

```
                []
                [Mes == no] -> (
                        process_child_list[send,recv](
                        parent,
                        Nname,
                        My_start,
                        My_parent,
                        Term_set,
                        My_children \ From,
                        Expected_res \ From)
                        )
```

(* A *parent* node that hears that one of its children is terminating adds the name of this child to the list of children that have *terminated*. It will itself terminate when all its children have *terminated* (this is not shown) *)

```
                []
                [Mes == terminating] -> (
                        Algorithm[send,recv](
                        parent,
                        Nname,
                        My_start,
                        My_parent,
                        (From ++ NIL2) v Term_set,
                        My_children,
                        Expected_res)
                        )
                )
        )
```

(* A shutting_down node waits for all its children to terminate the construction of their sub-spanning trees, and then terminates *)

```
        []
        [S == shutting_down] ->
                . . .
                . . .
        )
endproc
```

6. TESTING DISTRIBUTED ALGORITHMS SPECIFIED IN LOTOS

As mentioned earlier, one of the features of LOTOS is the (partial) executability of the language. LOTOS specifications can be written to be executable like programs, and the specifications given above were written in this way. Therefore, such specifications can be tested for design errors by using strategies similar to the ones familiar for conventional programs. Using tools such as the University of Ottawa LOTOS interpreter [GHL], this can be done in several ways:

a. Step by step execution: the user plays the role of the environment; non-determinism and choice of values are resolved by the user. At each step, the interpreter provides the user with a menu of possible next actions, and requests the values needed.
b. Random walk execution: the system randomly resolves the non-determinism and proceeds automatically as far as possible in a randomly chosen execution path, but the user still has to choose values when required [LOBF]. Random walk is a technique studied by C. West for testing protocol specifications[W][W1]. It was shown in these papers that random walk can allow satisfactory coverage of a large global state space, impossible to explore exhaustively.
c. Symbolic execution tree [GHL] or symbolic expansion [QP]: in this case the tree of all possible behaviours of the specification is computed. Symbolic values are used instead of user-provided values. This type of execution provides the most complete information about the behavior of a specification, however unfortunately for most specifications execution trees grow very quickly.

Many distributed algorithms have little or no interaction with the environment. Apart from some parameters provided at the beginning, the system evolves independently until a solution is obtained. This can be shown in the specification by hiding all message exchanges from the environment, e.g.

hide *send, recv* **in**
Algorithm[*send,recv*]

It can be seen that for this class of algorithms the *random walk* mode of execution becomes the most efficient way for testing the specification, once the initial bugs are removed.

A random walk execution trace for the election algorithm using the network of Figure 2 is given in Fig. 4.

choice(N:Network = $net1)	network configuration is provided
i (specified explicitly)	n1 initiates
send !n1:Name !n2:Name !pos(0,2,0,5):Key	n1 sends own key to n2, becoming *candidate*
i (specified explicitly)	n2 initiates (before receiving key)
i (specified explicitly)	n3 initiates
send !n2:Name !n3:Name !pos(1,0,0,5):Key	n2 sends own key to n3, becoming *candidate*
recv !n1:Name !n2:Name !pos(0,2,0,5):Key	n2 receives key from n1, remains *candidate*
send !n3:Name !n1:Name !neg(0,0,0,5):Key	n3 sends own key to n1, becoming *candidate*
recv !n3:Name !n1:Name !neg(0,0,0,5):Key	n1 receives from n3, remains *candidate*
recv !n2:Name !n3:Name !pos(1,0,0,5):Key	n3 receives from n2, becomes *processor* for n2
send !n3:Name !n1:Name !pos(1,0,0,5):Key	n3 forwards n2's key to n1
recv !n3:Name !n1:Name !pos(1,0,0,5):Key	n1 receives from n3, becomes *processor* for n2
send !n1:Name !n2:Name !pos(1,0,0,5):Key	n1 forwards n2's key to n2
recv !n1:Name !n2:Name !pos(1,0,0,5):Key	n2 receives own key from n1, becomes *leader*

send !n2:Name !n3:Name !pos(1,0,0,5):Key	n2 sends own key to n3
recv !n2:Name !n3:Name !pos(1,0,0,5):Key	n3 receives n2's key
send !n3:Name !n1:Name !pos(1,0,0,5):Key	n3 forwards n2's key to n1 and stops
recv !n3:Name !n1:Name !pos(1,0,0,5):Key	n1 receives n2's key from n3
send !n1:Name !n2:Name !pos(1,0,0,5):Key	n1 forwards n2's key to n2 and stops
recv !n1:Name !n2:Name !pos(1,0,0,5):Key	n2 receives again own key and stops
>>>>> DEADLOCK <<<<<	all processes stopped

Figure 4: execution trace for the election algorithm

The only value provided by the user is $net1, which denotes the system configuration shown in Fig. 2. This configuration was defined previously by the user and stored in a data base of constants.

Fig.5 shows an execution trace for the spanning tree algorithm using the network of figure 3 and having n1 as *initiator*.

choice(N:Network = $net2:, Initiator:Name =n1)	network config. and *initiator* (n1) are provided
send !n1:Name !n2:Name !are_you_my_child:Message	n1 sends a request to all its neighbours (n2 and n3)
send !n1:Name !n3:Name !are_you_my_child:Message	
i (enable: exit)	and becomes a *parent*
recv !n1:Name !n2:Name !are_you_my_child:Message	n2 receives the request from n1
send !n2:Name !n1:Name !yes:Message	n2 replies *yes*
send !n2:Name !n3:Name !are_you_my_child:Message	n2 sends the same request to all its neighbours
send !n2:Name !n4:Name !are_you_my_child:Message	except n1 (i.e. n3 and n4)
i(enable: exit)	and becomes a *parent*
recv !n2:Name !n1:Name !yes:Message	n1 receives *yes* from n2
recv !n2:Name !n3:Name !are_you_my_child:Message	n3 receives a request from n2 before receiving the request from n1
send !n3:Name !n2:Name !yes:Message	n3 replies *yes* to n2
recv !n2:Name !n4:Name !are_you_my_child:Message	n4 receives the request from n2
send !n4:Name !n2:Name !yes:Message	and replies *yes*
recv !n4:Name !n2:Name !yes:Message	n2 receives *yes* from n4
recv !n3:Name !n2:Name !yes:Message	n2 also receives *yes* from n3 and becomes *shutting_down*
send !n3:Name !n1:Name !are_you_my_child:Message	n3 broadcasts a request to all its neigbours except n2
send !n3:Name !n4:Name !are_you_my_child:Message	
i (enable: exit)	n3 becomes a *parent*
send !n4:Name !n3:Name !are_you_my_child:Message	n4 broadcasts a request to all its neighbours except n2
i (enable: exit)	and becomes a *parent*
(* at this stage all nodes are in *parent* state and waiting for a *complete* reply from neighbours *)	
recv !n3:Name !n1:Name !are_you_my_child:Message	n1 receives a request from n3 and
recv !n1:Name !n3:Name !are_you_my_child:Message	n3 receives a request from n1
send !n1:Name !n3:Name !no:Message	n1 replies *no* to n3 and stays in state *parent*
send !n3:Name !n1:Name !no:Message	n3 replies *no* to n1 and stays in state *parent*
recv !n1:Name !n3:Name !no:Message	n3 receives *no* from n1
recv !n3:Name !n1:Name !no:Message	n1 receives *no* from n3 and becomes *shutting_down*
recv !n4:Name !n3:Name !are_you_my_child:Message	n3 receives a request from n4
recv !n3:Name !n4:Name !are_you_my_child:Message	and vice versa
send !n3:Name !n4:Name !no:Message	they reply *no* to each other and they stay in *parent* state
send !n4:Name !n3:Name !no:Message	
recv !n4:Name !n3:Name !no:Message	n3 receives *no* from the last expected neighbour n4
send !n3:Name !n2:Name !terminating:Message	and sends a terminating signal to its *parent* n2
recv !n3:Name !n2:Name !terminating:Message	n2 receives the terminating signal from n3
recv !n3:Name !n4:Name !no:Message	n4 receives *no* from the only expected neghbour n3
send !n4:Name !n2:Name !terminating:Message	and sends a terminating signal to its *parent* n2

recv !n4:Name !n2:Name !terminating:Message	n2 receives the terminating signal from n4
send !n2:Name !n1:Name !terminating:Message	and sends a terminating signal to its *parent* n1
recv !n2:Name !n1:Name !terminating:Message	n1 receives a terminating signal from the only expected neighbour n2
i (explicit: send !node(n1, root, ++(n2,NIL2)): Node)	n1 knows that it is the root and its child is n2
i (explicit: send !node(n2, n1, ++(n4,++(n3,NIL2))): Node)	n2 has n1 as *parent* and n4 and n3 as children
i (explicit: send !node(n3, n2, NIL2): Node)	n3 has n2 as *parent* and has no children
i (explicit: send !node(n4, n2, NIL2): Node)	n4 has n2 as *parent* and has no children
>>>>> DEADLOCK <<<<<	all processes stopped

Figure 5: Execution trace for the Spanning Tree construction algorithm

The last four actions show the *parent* process and the children of every process in the constructed spanning tree after their termination. This corresponds to the tree of Fig. 6.

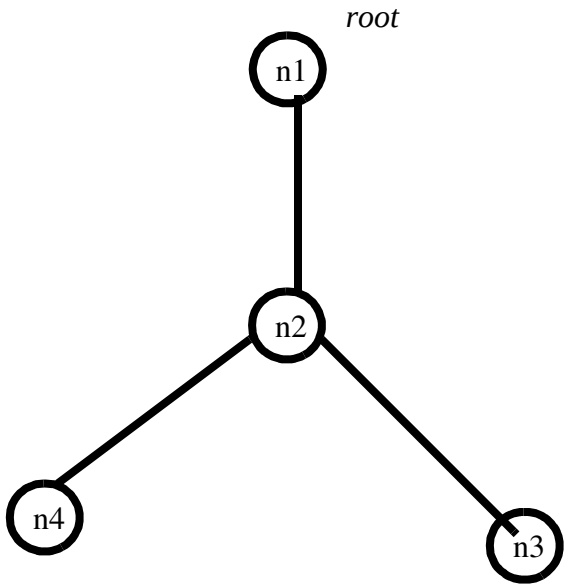

Figure 6

6. CONCLUSION

We have shown how a wide class of distributed algorithms can be specified in LOTOS. First of all, we have shown how a general network with unidirectional or bidirectional links can be specified. Further, we have specified two algorithms for: (1) leader election in a unidirectional ring (2) single-initiator spanning tree construction. Finally, we have shown how the algorithms so specified can be tested for design errors by using random-walk execution. Readers who might have become worried by the excessive length of traces such as the one of Fig. 5, will be relieved to hear that automated trace-checking tools are available [BBe]. And so are model-checking tools able to perform

automatic verification of temporal logic properties [GS].

The complete LOTOS specifications for the two examples given here are available from the authors.

Algorithms such as the one presented in this article are sometimes presented in a synchronous form, where processes communicate directly without links. This can also be done in LOTOS, and is left as an exercise for the reader. An advantage of this solution is that execution traces are about half as long.

It should be mentioned at this point that considerable theoretical and practical developments are taking place within the realm of LOTOS research on such items as LOTOS specification styles, LOTOS test methods, and other topics related to the topic of this paper. We have chosen a pragmatic and tutorial approach, however readers wanting to become users of the language should become acquainted with this literature, to understand the methodology associated with the language.

Acknowledgment. This research was supported in part by the Telecommunications Research Institute of Ontario, by the Natural Sciences and Engineering Research Council of Canada, and by Bell-Northern Research.

References

[AG] Afek, Y., and Gafni, E. Election and Traversal in Unidirectional Networks, Proc. 3rd ACM Symp. on Principles of Distributed Computing, Vancouver, Aug. 1984, 190-198.

[BBe] Bochmann, G.v., and Bellal, O. Test Result Analysis in Respect to Formal Specifications, Proc. 2nd Int. Workshop on Protocol Test Systems, Berlin, Oct. 1989, pp.272-294.

[BBr] Bolognesi, T., and Brinksma, E. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems 14* (1987) 25-59. Also reprinted in [VVD] 23-73.

[BC] Bolognesi, T., and Caneve, M. Equivalence Verification: Theory, Algorithms, and a Tool. In: [VVD] 303-326.

[CR] Chang, E., and Roberts, R. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configuration of Processes. *Comm. ACM 22,* 5 (May 79), 281-283.

[EM] Ehrig, H., Mahr, B., *Fundamentals of Algebraic Specification 1*, SpringerVerlag,Berlin,1985.

[FLS] Faci, M., Logrippo, L., and Stepien, B. Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Approach. To appear in Computer Networks and ISDN Systems.

[GHS] Gallager, R.G., Humblet, P.A. and Spira, P.M. A Distributed Algorithm for Minimum-weight Spanning Trees, *ACM Transactions on Programming Languages and Systems 5*, 1, Jan. 1983, 66-77.

[GLO] Gallouzi, S., Logrippo, L., and Obaid, A. A Hoare-Style Proof System for LOTOS. To

appear in: Quemada, J., Manas, J., and Vazquez, E. (eds.) *Formal Description Techniques.* North-Holland, 1991.

[GS] Garavel, H., and Sifakis, J. Compilation and Verification of LOTOS Specifications. In: Logrippo, L., Probert, R.L., and Ural, H. (eds.) *Protocol Specification, Testing, and Verification, X.* North-Holland, 1990, 379-394.

[GM] Garcia-Molina, H. Elections in Distributed Computing Systems, *IEEE Transactions on Computers C31*, 1, Jan. 1982, 48-59.

[GHL] Guillemot, R., Haj-Hussein, M., and Logrippo, L. Executing Large LOTOS Specifications. In: Aggarwal, S., and Sabnani, K. (eds.) *Protocol Specification, Testing, and Verification VII*. North-Holland, 1988, 399-410.

[Hoa] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hum] Humblet, P.A. A Distributed Algorithm for Minimum Weight Directed Spanning Trees, *IEEE Trans. on Communication Comm-31*, 6, June 1983, 756-762.

[ISO] International Organization for Standardization. Information Processing Systems. Open Systems Interconnection. Basic Reference Model for Open Systems Interconnection (ISO International Standard 7498), 1984.

[ISO1] International Organization for Standardization. Information Processing Systems. Open Systems Interconnection. LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour (ISO International Standard 8807), 1988.

[KRS] Korach, E., Rotem, D., and Santoro, N. Distributed election in a circle without a global sense of orientation, *Int. J. of Computer Mathematics 16*, 1984, 115-124.

[Kut] Kutten, S. A unified approach to the efficienr construction of distributed leader-finding algorithms, *Proc. IEEE COnf. on Communication and Energy*, Montreal, Oct. 1984.

[LFH] Logrippo, L., Faci, M., and Haj-Hussein, M. An Introduction to LOTOS: Learning by Examples. To appear in Computer Networks and ISDN Systems.

[LOBF] Logrippo, L., Obaid, A., Briand, J.P., and Fehri, M.C. An Interpreter for LOTOS, a Specification Language for Distributed Systems. *Software-Practice and Experience, 18* (1988) 365-385.

[MM] Mañas, J.A., and de Miguel-More, T. From LOTOS to C. In: K.J.Turner (ed.) *Formal Description Techniques.* North-Holland, 1989, 79-84.

[Mil] Milner, R. *Communication and Concurrency*. Prentice-Hall, 1989.

[Par] Park, D. Concurrency and Automata on Infinite Sequences, Proc. 5th GI Conference, *Lecture Notes in Computer Science 104*,167-183, 1981.

[QP] Quemada, J, Pavon, S., and Fernandez, A. Transforming LOTOS Specifications with LOLA. The Parameterised Expansion. In: K.J.Turner (ed.) *Formal Description Techniques.* North-Holland, 1989, 45-54.

[Se] Segall, A. Distributed Network Protocols. *IEEE Trans. on Information Theory IT-29,*1,1983,

23-35.

[VSV] Vissers, C., Scollo, G., and Van Sinderen, M. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In Aggarwal, S., and Sabnani, K., (eds.) *Protocol Specification, Testing, and Verification, VIII,* North-Holland, 1988, 189-204

[VVD] van Eijk, P., Vissers, C.A., and Diaz, M.*The Formal Description Technique LOTOS.* NorthHolland, 1989.

[W] West, C. Protocol Validation by Random State Exploration. *Protocol Specification, Testing, and Verification, VI.* North-Holland, Amsterdam, 1986, 233-242.

[W1] West, C. Protocol Validation in Complex Systems. SIGCOMM'89 Computer Communications Review, Vol. 19, no. 4, Sept. 89, 303-312.