On the Application of Formal Methods for Specifying and Verifying Distributed Protocols[*]

Marina Gelastou, Chryssis Georgiou, and Anna Philippou
Dept. of Computer Science, University of Cyprus, CY-1678 Nicosia, Cyprus
{gelastoum, chryssis, annap}@cs.ucy.ac.cy

Abstract

In this paper we consider the frameworks of Process Algebra and I/O Automata and we apply both towards the verification of a distributed leader-election protocol. Based on the two experiences we evaluate the approaches and draw initial conclusions with respect to their relative capabilities, strengths and usability.

To the best of our knowledge, this is the first hands-on evaluation of the two models, and we view it as the cornerstone for a wider investigation of the strengths and weaknesses of the two methodologies in specifying and verifying (distributed) protocols.

1 Introduction

Modern distributed systems are intrinsically difficult to develop and reason about. The need for formally verifying the correctness of distributed/parallel systems and protocols has long been realized by the research community, both theoreticians and practicians. In the last two decades, the field of formal methods for system design and analysis has dramatically matured and has reported significant success in the development of theoretical frameworks for formally describing and analyzing complex systems as well as for providing methodologies and practical tools for these purposes. More specifically, during the last twenty years, significant research efforts were geared towards the development of formal methodologies for system modeling and verification. Two prominent such models are those of Input/Output Automata, IOA [10], and Process Algebra, PA [12, 2]. Both models are equipped with precise semantics, thus providing a solid basis for understanding system behavior and reasoning about correctness. Since their inception, they have been the subject of extensive research and they have been extended in various directions. Furthermore they have been used in the literature for reasoning about a variety of protocols (see, for example, [9, 3, 6, 17] and [1, 16, 18] for I/O automata and PAs respectively).

It is fair to say that these two formalisms are important, well-developed theories that have a lot to offer towards understanding and reasoning about complex systems. However, to date, research carried in each line of work has been quite distinct. At the same time, new variations and extensions of verification formalisms keep cropping up while a thorough investigation into their applicability, strengths and potentials is still missing. Indeed, recently, concerns are being raised with regards to the potentials of formal methodologies towards the verification of today's complex protocols and environments, see e.g. [4].

While the need of applying formal techniques for reasoning about protocol correctness is generally accepted, various questions still remain open and hinder the selection and adoption of these techniques. Questions include: Which formalisms are appropriate to use for distributed and networked protocols? Is there one that is "clearly better" for these protocols, or classes thereof? Which one is "easier" to learn and apply? Would a newcomer (e.g., postgraduate student) be able to apply such methods to specify and verify the protocols we develop?

We begin to consider these questions by verifying a typical distributed protocol in both the IOA and the PA formalisms. In particular we specify and verify a message-passing leader-election protocol [24] with static membership and fault-free components. The choice of the protocol was made based on two facts: (a) the leader election problem is a fundamental problem in distributed computing and hence, an interesting problem to consider, and (b)

[*]This work was supported in part by the Cyprus Research Foundation Grant "ΠΡΟΟΔΟΣ" and the University of Cyprus.

the protocol is simple enough to focus on its specification and verification rather than on its understanding, but at the same time complex enough to enable us to evaluate the two frameworks and draw conclusions. In addition, this protocol has not been verified in the past in either of the two formalisms.

We observe the capabilities of each of the frameworks for modeling the specific protocol. We apply the associated proof techniques for proving the protocol's correctness and we evaluate them with respect to their relative capabilities, strengths and usability. We present and compare the two experiences. Specifically, from our case study we single-out the following:

- Both formalisms were successful in specifying and verifying the protocol under study. For each method, standard/natural specification style and proof techniques were employed, demonstrating that for distributed protocols of a similar nature as the one under study, both methods are applicable.
- The correctness criterion of the protocol consisting of a global property (a common leader is elected) as well as its deterministic behavior, rendered the process-calculus proof methodology very natural to apply. This does not imply that the I/O Automata proof has been less easy to establish; however, it required breaking the proof into two parts, the first of which involving the transformation of local properties into a global one (the creation of a spanning tree).
- As reported by a newcomer to the two formalisms, the programming style of I/O Automata specification and the nature of the I/O Automata proofs (induction and code inspection) enable the easier understanding and use of this framework. This does not imply that Process Algebras are a difficult tool to employ. It does appear, however, that greater expertise and investment of time is required in order to learn and apply this latter methodology which may yield more rigorous proofs.

To the best of our knowledge, this is the first such *hands-on* evaluation of the two formalisms (in general – not only for distributed protocols). We believe that this line of work benefits both theoreticians and practicians of the distributed and networked community.

2 Prelimilaries

In this section we present the formalisms of IOA and PA and the protocol to be verified.

2.1 I/O Automata

We begin with an overview of the I/O Automata formalism of Lynch and Tuttle [10], focusing on notions used in this study. For more detailed presentations we refer the reader to [9, 10, 11].

An I/O Automaton is a labeled state transition system. It consists of three type of atomic transitions which are named *actions*: input, output and internal. The input actions of an I/O automaton are generated by the environment and are transmitted instantaneously to the automaton. In contrast, the automaton can generate the output and internal actions autonomously and can transmit output actions instantaneously to its environment. Actions are described in a *precondition-effect* style. An action π is *enabled* if its preconditions are satisfied. Input actions are always enabled. A *signature* of an I/O automaton consists of three disjoint sets of input, output and internal actions. The operation of an I/O automaton is described by its *executions* and *traces*. An *execution fragment* of an automaton A is a finite sequence $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, \ldots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a transition or *step* of A, for every $i \geq 0$. An *execution* is an execution fragment that starts with an initial state (i.e. s_0 is an initial state). A *fair* execution is an execution in which if the automaton enables its locally-controlled actions infinitely often then it executes them infinitely often. A *trace* is an external behavior of an automaton A that consists of the sequence of input and output actions occurring in an execution of A. I/O automata can be composed to create more complex I/O automata. The (parallel) *composition* operator allows an output action of one automaton to be identified with the input actions in other automata; this operator respects the trace semantics.

Within the I/O automata framework, proving the correctness of an automaton is often deduced to showing *safety* and *liveness* properties of the automaton. Informally speaking, a safety property specifies a property that must hold in *every* state of an execution (i.e., something "bad" never happens) and a liveness property specifies events that must *eventually* be performed (i.e., something "good" eventually happens). Liveness properties can only be satisfied by fair executions. An *invariant* is a property that is true in all states of an automaton. Invariants are typically proved by induction on the length of an execution leading to the state in question. Several invariants are usually combined in proving (mainly) safety properties of a given automaton. A common technique for reasoning about the behavior of a composed automaton is *modular decomposition*: first, one proves less complex invariants for the automata of the composition, and then it uses the composition of those invariants to reason about the composed automaton. Another technique is the use of *simulation relations* [11], which we do not discuss here as it is not employed in this work.

2.2 The Process Algebra

Many process algebras have been proposed in the literature. For our purposes, we have found one of the most basic ones, CCS$_v$, to suffice. CCS$_v$ is a value-passing calculus [12, 22] which includes conditional agents. For a more detailed presentation we refer to these works as well as [18].

We begin by describing the basic entities of the calculus. We assume a set of *constants*, ranged over by v, a set of functions, ranged over by f, operating on these constants and a set of *variables*, ranged over by x. These give rise to the set of *terms* of CCS$_v$ ranged over by e, in the expected way. Moreover, we assume a set of *channels*, $\mathcal{L}$, ranged over by a, b. Channels provide the basic communication and synchronization mechanisms in the language. A channel a can be used in *input position*, denoted by a, and in *output position*, denoted by $\overline{a}$. This gives rise to the set of *actions Act* of the calculus, ranged over by α, β, containing (1) the set of *input actions* which have the form $a(\tilde{v})$ representing the input along channel a of a tuple $\tilde{v}$, (2) the set of *output actions* which have the form $\overline{a}(\tilde{v})$ representing the output along channel a of a tuple $\tilde{v}$, and (3) the *internal action* τ, which arises when an input action and an output action along the same channel are executed in parallel. We say that an input action and an output action on the same channel are *complementary* actions. Finally, we assume a set of process constants $\mathcal{C}$, denoted by C. We assume that each constant C has an associated definition of the form $C\langle\tilde{x}\rangle \overset{\text{def}}{=} P$, where the process P may contain occurrences of C, as well as other constants. The syntax of CCS$_v$ is given as follows:

$$\begin{aligned} P \quad ::= \quad & \mathbf{0} \mid \alpha.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid P\backslash L \mid \\ & \mathsf{cond}\ (e_1 \rhd P_1, \ldots, e_n \rhd P_n) \mid C\langle\tilde{v}\rangle. \end{aligned}$$

Process $\mathbf{0}$ represents the inactive process. Process $\alpha.P$ describes the process which first engages in action α and then behaves as process P. Process $P_1 + P_2$ represents the nondeterministic choice between processes P_1 and P_2. Process $P \| Q$ describes the parallel composition of P and Q: the component processes may proceed independently or interact with one another while executing complementary actions. The conditional process $\mathsf{cond}\ (e_1 \rhd P_1, \ldots, e_n \rhd P_n)$ presents the conditional choice between a set of processes: assuming that all e_i are closed terms, it behaves as P_i, where i is the smallest integer for which e_i evaluates to $\mathtt{true}$. In $P\backslash F$, where $F \subseteq \mathcal{L}$, the scope of channels in F is restricted to process P: components of P may use these channels to interact with one another but not with P's environment. Finally, process constants provide a mechanism for including recursion in the process calculus.

The semantics of the calculus is given by structural operational semantics: each operator is given precise meaning via a set of rules which, given a process P, prescribe the possible transitions of P, where a transition of P has the form $P \xrightarrow{\alpha} P'$, specifying that P can perform action α and evolve into P'. These transitions give rise to a labeled directed graph whose vertices are the possible states of the process and where an edge (s, α, s') signifies that it is possible to evolve from s to s' by executing action α. Based on this transition relation, we write $P \overset{\alpha}{\Longrightarrow} P'$ for $P(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* P'$.

Processes are analyzed and compared on the basis of their state graphs. One common method of performing this is the use of observational equivalences. Observational equivalences are based on the idea that two equivalent systems exhibit the same behavior at their interfaces with the environment. This requirement was captured formally through the notion of *bisimulation* [12, 14]. Bisimulation is a binary relation on processes. Two processes are bisimilar if, for each step of one, there is a matching (possibly multiple) step of the other, leading to bisimilar states. In this work we employ our study on weak bisimulation, $\approx$. We refer the reader to [13] for the full details.

Another concept used in our study is the notion of *confluence*. A process is *confluent* if, from each of its reachable states, "of any two possible actions, the occurrence of one will never preclude the other" [13]. As shown in [13, 12] for pure CCS, and generalized in other calculi (e.g. [7, 22, 8, 15, 19, 18]), confluence implies determinacy and semantic-invariance under internal computation, and it is preserved by several system-building operators. These facts make it possible to reason compositionally that a system is confluent and to exploit this fact while reasoning about its behavior. In particular, for a certain class of confluent processes, in order to check that a property is satisfied in every execution of the system it suffices to show that it is satisfied by a single (arbitrary) execution.

2.3 The Leader-Election Protocol

The protocol we consider, hereafter called LE, is the static version of a distributed leader-election protocol presented in [24]. It operates on an arbitrary topology of nodes with distinct identifiers and it elects as the leader of the network the node with the maximum identifier.

In brief, the protocol operates as follows. In its initial state, a network node may initiate a leader-election computation (note that more than one node may do this) or accept leader-election requests from its neighbors. Once a node initiates a computation, it triggers communication between the network nodes which results into the creation of a spanning tree of the graph: each node picks as its parent the node from which it received the first request, forwards the request to all of its remaining neighbors and ignores all subsequent received requests, with an exception described below. Consequently, each node awaits to receive from each of its children the maximum identifier of the subtrees at which they are rooted and, then, it forwards to its parent the maximum identifier of the subtree rooted at the node. Naturally, this computation begins at the leaves of the tree and proceeds towards the root. Once this information is received by the root all necessary information to elect the leader is available. Thus, the root broadcasts this information to its neighbors who in turn broadcast this to their neighbors, and so on.

Note that if more than one node initiates a leader-election computation then only one computation survives which is the one originating from the node with the maximum identifier. This is established by associating each computation with a source identifier. Whenever a node already in a computation receives a request for a computation with a greater source, it abandons its original computation and it restarts executing a computation with this new identifier.

3 Specification and Verification in IOA

3.1 Specification

The specification of protocol LE in I/O automata is the composition of the LENode$_i$ automata and the Channel automata $C_{i,j}$, $\forall i, j \in I$. The data types, identifiers, signature, and states of the LENode$_i$ automaton are given in Fig. 1 and its transitions (actions) in Fig. 2. Automaton $C_{i,j}$ is the one typically used for non-lossy channels and can be found in [5].

3.2 Correctness Proof

The correctness proof is divided into two main parts. We first show that a unique spanning tree is built, and using this fact we show that a unique common node (the one with the highest id) is elected as the leader. For each part safety and liveness properties are stated. The technique of modular decomposition is used for the final conclusions. Due to space limitations, full proofs are not presented, but can be found in [5]. Invariants are proved by induction on the length of the execution.

Data Types and Identifiers:

I: total ordered set of processes' identifiers

$\mathcal{M}$: messages

$m = \langle type, maxid, leaderid, srcid, mychild\rangle \in \mathcal{M}$, where

$type \in \{election, ack, leader\}$;$maxid, leaderid, srcid \in I \cup \{\bot\}$;

$mychild$: $Boolean$

$i, j \in I$

Signature:

Input:	**Output:**	**Internal:**
receive$(m)_{j,i}$	send$(m)_{i,j}$	beginComputation$_i$
		setAcktoParent$_i$
		setLeader$_i$

States:

$max_i \in I \cup \{\bot\}$, initially $\bot$

$src_i \in I \cup \{\bot\}$, initially $\bot$

$leader_i \in I \cup \{\bot\}$, initially $\bot$

$parent_i \in I \cup \{\bot\}$, initially $\bot$

$Nbrs_i \in 2^I$: Neighbors of i

$inElection_i$: $Boolean$, initially $false$

$sentAcktoParent_i$: $Boolean$, initially $true$

$toBeAcked_i \in 2^I$, initially $\emptyset$

$tosend_i$, a vector of queues of messages, initially $tosend_i[j] = null,\ \forall j \in I$

Figure 1. The LENode$_i$ automaton's types, identifiers and states.

3.2.1 A Unique Spanning Tree is Built

We state the safety and liveness properties that lead to the conclusion that protocol LE builds a unique spanning tree.

Safety Properties

The first invariant states that once a node enters a leader-election computation, it adopts a parent and a source (root) of a potential spanning tree.

Invariant 1 *Given any execution of* LE*, any state s, and any $i \in I$,*

(a) if $s.inElection_i = false$ and $s.leader_i = \bot$ then $s.src_i = \bot$ and $s.parent_i = \bot$.

(b) if $s.inElection_i = true$ then $s.src_i \neq \bot$ and $s.parent_i \neq \bot$.

Transitions:

```
input receive$(m)_{j,i}$
Effect:
    if $m.type = election$ then
      if $(inElection_i{=}false \lor (inElection_i{=}true \land m.srcid > src_i))$ then
        $src_i := m.srcid$
        for all $k \in Nbrs_i - \{j\}$ do
          enque $m$ to $tosend_i[k]$
        $toBeAcked_i := Nbrs_i - \{j\}$
        $sentAcktoParent_i := false$
        $inElection_i := true$
        $parent_i := j$
        $max_i := i$
      elseif $(sentAcktoParent_i = false \land\ src_i = m.srcid)$ then
        enque $\langle ack, max_i, *, src_i, false\rangle$ to $tosend_i[j]$
    elseif $m.type = ack$ then
      if $sentAcktoParent_i = false \land m.srcid = src_i$ then
        remove $j$ from $toBeAcked_i$
        if $m.mychild = true \land m.maxid > max_i$ then
          $max_i := m.maxid$
    elseif $m.type = leader$ then
      if $sentAcktoParent_i = true \land inElection_i = true$
      $\land m.srcid = scri$ then
        $leader_i := m.leaderid$
        $inElection_i := false$
        for all $k \in Nbrs_i - \{j\}$ do
          enque $m$ to $tosend_i[k]$

output send$(m)_{i,j}$
Precondition:
    $m$ first on $tosend_i[j]$
    $j \in Nbrs_i$
Effect:
    deque $m$ from $tosend_i[j]$

internal beginComputation$_i$
Precondition:
    $inElection_i = false \land leader_i = \perp$
Effect:
    $scri = i$
    for all $k \in Nbrs_i$ do
      enque $\langle election, *, *, src_i, *\rangle$ to $tosend_i[k]$
    $toBeAcked_i := Nbrs_i$
    $sendAcktoParent := false$
    $inElection_i := true$
    $parent_i := i$
    $max_i := i$

internal setAcktoParent$_i$
Precondition:
    $toBeAcked_i = \emptyset \land src_i \neq i \land sentAcktoParent_i = false$
Effect:
    $sentAcktoParent_i = true$
    enque $\langle ack, max_i, *, src_i, true\rangle$ to $tosend_i[parent_i]$

internal setLeader$_i$
Precondition:
    $toBeAcked_i = \emptyset \land src_i = i \land sentAcktoParent_i = false$
Effect:
    $sentAcktoParent_i = true$
    $inElection_i = false$
    $leader_i = max_i$
    for all $k \in Nbrs_i$ do
      enque $\langle leader, *, leader_i, src_i, *\rangle$ to $tosend_i[k]$
```

Figure 2. The LENODE$_i$ automaton's transitions.

The next lemma states that source nodes do not appear "out of the blue". The proof is by code inspection and use of Invariant 1.

Lemma 3.1 *In any given state s of an execution of* LE*, for any $i, j \in I$ if $s.src_i = j$, then there exists a step (s_1, π, s_2), $s_1 < s$, $s_2 \leq s$ and $\pi =$ beginComputation$_j$.*

Let $exec_{i_0}$ be any execution of LE where only a single node i_0 begins computation. We call i_0 the *initiator* of the computation. The next invariant states that once a process enters a computation with a unique initiator, it becomes part of the spanning tree rooted at the initiator.

Invariant 2 *Given any execution $exec_{i_0}$ of* LE*, any state s, and for all $i \in I$ such that $s.parent_i \neq \perp$, then the edges defined by all $s.parent_i$ variables form a spanning tree of the subgraph of G rooted at i_0.*

The following invariant states that a node adopts a new source only if it is higher than its current source.

Invariant 3 *For any process $i \in I$ and for any two states s, s' s.t. $s < s'$ of any execution of* LE*, if $s'.src_i \neq s.src_i$, then $s'.src_i > s.src_i$.*

Liveness Properties

This lemma states that in executions with a single initiator a unique spanning tree is eventually built rooted at the initiator.

Lemma 3.2 *In any fair execution $exec_{i_0}$, all nodes $i \in I$ eventually belong to a unique spanning tree rooted at i_0.*

Proof. For any node $j \in I$, let D_j denote the length (in terms of hops) of the longest loop-free path from i_0 to j. We show that eventually j belongs to the spanning tree rooted at i_0. The proof is by induction on D_j and uses Lemma 3.1 and Invariant 2. □

If more than one beginComputation$_i$ actions occur, let i_{smax} be the node with the maximum i value among them. The following theorem, the core result of this section, shows that a unique spanning tree is eventually built.

Theorem 3.3 *Protocol* LE *eventually builds a unique spanning tree rooted at i_{smax}.*

Proof. The proof makes use of Lemma 3.2 and Invariant 3. The idea is that if more than one initiators begin computation, by Invariant 3, only the computation with the highest id survives and as per Lemma 3.2 a unique spanning tree rooted at that node is eventually built. □

3.2.2 A Unique Common Leader is Elected

We now state the safety and liveness properties that lead to the correctness of protocol LE.

Safety Properties

The following invariant states that a node adopts a new max value only if it is higher than its current one.

Invariant 4 *For any node $i \in I$ and for any two states s, s' s.t. $s' < s$ of any execution of* LE*, if $s.src_i = s'.src_i$ and $s.max_i \neq s'.max_i$, then $s'.max_i > s.max_i$.*

The following lemma states that each child propagates to its parent the maximum value of its subtree. The proof is by code inspection and use of Invariant 4.

Lemma 3.4 *In any state s of an execution of* LE*, if $s.toBeAcked_i = \emptyset$ and $s.sentAcktoParent_i = false$ then $s.max_i$ is the greatest value among i and the values that i has "seen" from its children.*

Let i_{max} denote the process with the maximum value i. The next theorem (which is actually an invariant) states that if a node elects a leader, this can only be i_{max}.

Theorem 3.5 *For any node i and state s of any execution of* LE*, if $s.leader_i \neq \perp$, then $s.leader_i = i_{max}$.*

Liveness Properties

We now give the main result that states that protocol LE indeed solves the Leader Election problem.

Theorem 3.6 *Given a fair execution of* LE *there exists a state s where $\forall\, i \in I$, $s.leader_i = i_{max}$.*

Proof. The proof makes use of Theorem 3.3 stating that a unique spanning tree is built. Then it proceeds by induction on the depth of the spanning tree and by making use of Lemma 3.4 it is shown that eventually the max value is propagated to the root of the tree. Then it is argued that the leader message sent by the root is eventually received by all nodes and by Theorem 3.5 all nodes elect i_{max} as the leader, as desired. □

4 Specification and Verification in PA

4.1 Specification

In this section we give a description of the LE protocol in CCS$_v$. We assume a set K consisting of the node unique identifiers and a set of channels $F = \{election_{i,j}, ack0_{i,j}, ack1_{i,j}, leader_{i,j} \mid i,j \in K, i \neq j\}$ where $x_{i,j}$ refers to the channel from node i to node j of type x. The system is described as the following parallel composition of its constituent nodes:

$$P_0 \overset{\text{def}}{=} (\prod_{k \in K} \text{NoLeader}\langle u_k, N_k\rangle)\backslash F$$

Initially, all nodes are of type $\text{NoLeader}\langle i, N\rangle$ but may evolve into processes $\text{InComp}\langle i, f, s, N, S, R, A, max\rangle$, $\text{LeaderMode}\langle i, s, N\rangle$ and $\text{ElectedMode}\langle i, s, N, S, l\rangle$, where i represents the identifier of the process, N the set of its neighbors, f and s are the father of the node and the source of the computation, respectively, S the set of request messages the node has still to send, R the set of potential children of the node from which it is waiting to hear and A the set of acknowledgement messages the process has still to send. The specification of these processes can be found in Fig. 3.

$$\begin{array}{l}
\text{NoLeader}\langle i, N\rangle \overset{\text{def}}{=} \tau.\,\text{InComp}\langle i, i, i, N, N, N, \emptyset, i\rangle \\
\quad + \sum_{j \in N} election_{j,i}(s).\,\text{InComp}\langle i, j, s, N, N - \{j\}, N - \{j\}, \emptyset, i\rangle
\end{array}$$

$$\begin{array}{l}
\text{InComp}\langle i, f, s, N, S, R, A, max\rangle \overset{\text{def}}{=} \\
\quad \sum_{j \in S} \overline{election_{i,j}}(s).\,\text{InComp}\langle i, f, s, N, S - \{j\}, R, A, max\rangle \\
\quad + \sum_{j \in A} \overline{ack0_{i,j}}(s).\,\text{InComp}\langle i, f, s, N, S, R, A - \{j\}, max\rangle \\
\quad + \sum_{j \in N} ack0_{j,i}(s'). \\
\qquad \textsf{cond}\ ((s = s') \ \triangleright\ \text{InComp}\langle i, f, s, N, S, R - \{j\}, A, max\rangle, \\
\qquad\qquad true \ \triangleright\ \text{InComp}\langle i, f, s, N, S, R, A, max\rangle) \\
\quad + \sum_{j \in N} ack1_{j,i}(s', max'). \\
\qquad \textsf{cond}\ ((s = s' \wedge max' > max) \triangleright \text{InComp}\langle i, f, s, N, S, R–\{j\}, A, max'\rangle, \\
\qquad\qquad (s = s' \wedge max' \leq max) \triangleright \text{InComp}\langle i, f, s, N, S, R–\{j\}, A, max\rangle, \\
\qquad\qquad true \ \triangleright\ \text{InComp}\langle i, f, s, N, S, R, A, max\rangle) \\
\quad + \sum_{j \in N} election_{j,i}(s'). \\
\qquad \textsf{cond}\ ((s' > s) \ \triangleright\ \text{InComp}\langle i, j, s', N, N - \{j\}, N - \{j\}, \emptyset, i\rangle, \\
\qquad\qquad (s' = s) \ \triangleright\ \text{InComp}\langle i, f, s, N, S, R, A \cup \{j\}, max\rangle, \\
\qquad\qquad true \ \triangleright\ \text{InComp}\langle i, f, s, N, S, R, A, max\rangle)
\end{array}$$

$$\begin{array}{l}
\text{InComp}\langle i, f, s, N, \emptyset, \emptyset, \emptyset, max\rangle \overset{\text{def}}{=} \\
\quad \overline{ack1_{i,f}}(s, max).\,\text{LeaderMode}\langle i, s, N\rangle
\end{array}$$

$$\begin{array}{l}
\text{InComp}\langle i, i, i, N, \emptyset, \emptyset, \emptyset, max\rangle \overset{\text{def}}{=} \\
\quad \overline{leader}(max).\,\text{ElectedMode}\langle i, i, N, N, max\rangle
\end{array}$$

$$\begin{array}{l}
\text{LeaderMode}\langle i, s, N\rangle \overset{\text{def}}{=} \\
\quad \sum_{j \in N} leader_{j,i}(s', max'). \\
\qquad \textsf{cond}\ ((s = s') \ \triangleright\ \text{ElectedMode}\langle i, s, N, N - \{j\}, max'\rangle, \\
\qquad\qquad true \ \triangleright\ \text{LeaderMode}\langle i, s, N\rangle)
\end{array}$$

$$\begin{array}{l}
\text{ElectedMode}\langle i, s, N, S, l\rangle \overset{\text{def}}{=} \\
\quad \sum_{j \in S} \overline{leader_{i,j}}(s, l).\,\text{ElectedMode}\langle i, s, N, S - \{j\}, l\rangle \\
\quad + \sum_{j \in N} leader_{j,i}(s', l').\,\text{ElectedMode}\langle i, s, N, S, l\rangle
\end{array}$$

Figure 3. The node process

4.2 Correctness Proof

The correctness criterion of our protocol is expressed as the following bisimulation equivalence between the system and its specification. The specification consists of the process that elects as a leader the node with the maximum identifier and terminates.

Theorem 4.1 $P_0 \approx \overline{\text{leader}}(\mathbf{max}).0$ *where* $\mathbf{max} = max\{u_i|\ i \in K\}$.

The proof is established in two phases. In the first phase we consider a simplification of P_0 where a single initiator begins computation and where the spanning tree on which the protocol operates is pre-determined. We show, by employing the notion of confluence, that this restricted system satisfies the above correctness requirement. It then remains to establish a correspondence between the general system P_0 and these restricted type of agents which leads to the desired result. Due to space limitations, full proofs are omitted but can be found in [5].

The restricted type of systems employed in the first phase of the proof uses the following processes:

$$\begin{aligned}
&\text{NoLeader}'\langle i, f, N, l\rangle \overset{\text{def}}{=} election_{f,i}(s).\\
&\quad \text{InComp}'\langle i, f, s, N, N-\{f\}, N-\{f\}, \emptyset, i\rangle\\
&\text{InComp}'\langle i, f, s, N, S, R, A, max\rangle \overset{\text{def}}{=} \ldots\\
&\quad + \textstyle\sum_{j\in N} election_{j,i}(s').\\
&\qquad \text{InComp}'\langle i, f, s, N, S, R, A\cup\{j\}, max\rangle\\
&\quad \ldots\\
&\text{LeaderMode}'\langle i, s, N\rangle \overset{\text{def}}{=} \textstyle\sum_{j\in N} leader_{j,i}(s', max').\\
&\quad \text{ElectedMode}\langle i, s, N, N-\{j\}, max'\rangle
\end{aligned}$$

Thus, $\text{NoLeader}'$ is similar to NoLeader except that it may only be activated by a signal from a specified node, f. Similarly, InComp' and $\text{LeaderMode}'$ are similar to InComp and LeaderMode, respectively, except that they do not take into account the source node of incoming $leader$ and $election$ messages.

Let $\mathcal{T}$ be the set of agents of the form

$$\begin{aligned}
T_0 \overset{\text{def}}{=} &(\textstyle\prod_{i\in K-\{\nu\}} \text{NoLeader}'\langle i, f_i, N_i, l_i\rangle \mid\\
&\text{InComp}'\langle \nu, \nu, \nu, N_\nu, N_\nu, N_\nu, \emptyset, \nu\rangle)\backslash F
\end{aligned}$$

where $\{(i, f_i)|i \in K - \{\nu\}\}$ is a spanning tree of the network rooted at node ν, for some $\nu \in K$. Our first result shows that T_0 has an execution where the maximum node is elected as a leader.

Lemma 4.2 $T_0 \overset{\overline{\text{leader}}(\mathbf{max})}{\Longrightarrow} \approx \mathbf{0}$.

Proof. The proof consists of the construction of an appropriate execution. The execution considered follows the intuitive break down of the protocol in its three phases and involves an induction on the height of the tree. □

Lemma 4.3 T_0 *is confluent.*

Proof. We may check that processes $\text{NoLeader}'$, InComp', $\text{LeaderMode}'$ and ElectedMode, are confluent by construction. By compositionality results (see [5]) the claim follows. □

From these two results we have that T_0 satisfies the protocol correctness criterion.

Corollary 4.4 $T_0 \approx \overline{leader}(\mathbf{max}).\mathbf{0}$ *where* $\mathbf{max} = max\{u_i|i \in K\}$.

Having used confluence to analyze the behavior of T_0, we can now relate it to that of P_0. Let P range over derivatives of P_0 and T range over derivatives of T_0. First, we introduce a notion of *similarity* between derivatives of P_0 and T_0. We say that P and T are *similar* if the computation initiator in T coincides with the maximum source node present in P and, additionally, the set of nodes in P that have this source form a subtree of the spanning tree of T. All such nodes are in the same state in both P and T whereas the remaining nodes are idle in T no matter their status in P.

Lemma 4.5 $\{\langle T, P\rangle | P \textit{ and } T \textit{ are similar}\}$ *is a strong simulation.*

Proof. The proof is a case analysis of the possible actions of the form $T \xrightarrow{\alpha} T'$. □

By Corollary 4.4 and Lemma 4.5 we have that $P_0 \overset{\overline{leader}(\mathbf{max})}{\Longrightarrow} \mathbf{0}$. Our final result establishes a correspondence between P_0 and agents $T_0 \in \mathcal{T}$.

Lemma 4.6 *If* $P_0 \overset{w}{\Longrightarrow} P$ *then there exists* T_0 *such that,* $T_0 \overset{w}{\Longrightarrow} T$ *and* P *and* T *are similar.*

Proof. The proof is by induction of the length of the transition $P_0 \overset{w}{\Longrightarrow} P$. □

We can now prove our main theorem. We have seen that $P_0 \overset{\overline{leader}(\mathbf{max})}{\Longrightarrow} \mathbf{0}$. Further, suppose that $P_0 \overset{\alpha}{\Longrightarrow}$ with $\alpha \neq \overline{leader}(\mathbf{max})$. Then, there exists T_0 such that $T_0 \overset{\alpha}{\Longrightarrow}$. However, this is in conflict with Corollary 4.4. Finally, for the same reason, it is not possible that $P_0 \Longrightarrow P_1' \not\longrightarrow$. This implies that $P_0 \approx T_0$, as required.

5 Framework Evaluation

In this section we evaluate the two approaches taken for reasoning about the LE protocol and draw conclusions regarding their applicability and relative strengths. We begin with some general observations on the two frameworks and then we evaluate them based on our experiences of specifying and verifying our case study.

One may observe that work in I/O Automata and Process Algebras was mostly carried out independently and that focus on each of them has been quite distinct. Work on PAs has concentrated on enhancing the expressive power of the associated languages, developing their semantic theories, and constructing automated analysis tools. On the other hand, work on I/O automata placed emphasis on application of the basic model and its proposed extensions to prove by hand the correctness of protocols. One of the few cross-points between the two lines of work was [23] where the semantic relationship between the formalisms was investigated. In that paper, I/O automata are recasted as a De Simone calculus and it is shown that, due to the input-enabledness of input actions and the non-blocking properties of the output actions, certain trace equivalences are substitutive for IOA. This fact also enables reasoning about fairness within I/O automata. In contrast, to provide compositional theories for typical PAs, it is necessary to consider the branching structure of processes. Thus, process algebras are typically given bisimulation or failure equivalence semantics based on which algebraic/compositional theories are built.

5.1 Specification

Beginning with the specifications developed in the two frameworks, we note that they have many similarities as well as points of distinction. For instance, they both consider the system as the parallel composition of the constituent components described as processes/automata. The nature of these processes/automata does not include any internal concurrency. Although this was expected in the I/O automata model, in process algebra there was an alternative option of firing all acknowledgement and election messages in processes concurrently to the main body of a node process. It turned out that the imposition of sequentiality and the maintenance of sets containing this information enabled the trackability of the system derivatives and a smoother proof. On the other hand, the models depart from each other in a number of ways.

Language syntax. The languages of the two formalisms differ substantially. The main differences concern the language constructs, the granularity of the actions, and the methodology used for describing flow of behavior. On the one hand, process algebras are based on a set of primitives and a fairly large and expressive set of constructs with the notions of communication and concurrency at the core of their languages (as it can be observed in Fig. 3). A system is modeled as a process which itself can be a composition of subprocesses representing further constituent components. Action granularity is very fine: actions can be input or output on channels and internal actions.

On the other hand, I/O automata feature a more "relaxed" type of language, quite close to imperative programming (as it can be observed in Fig. 2). It enables a limited (in comparison to PAs) set of operators: renaming and parallel composition. A system in this formalism is described as an I/O automaton which, as with process algebras, is built as the parallel composition of the system's sub-components. However, in contrast to PAs, an I/O automaton possesses a *state* and its behavior is prescribed by the set of actions the automaton may engage in. Input actions are always enabled, and output and internal actions cannot be prevented from arising. The effect of an action can be a complex behavior described as a sequence of simple instructions that involve operating on the automaton's state. This may result in a less fine granularity of actions in comparison to PA's.

State. As noted above, the I/O automata model builds on the notion of a state. The state of an automaton consists of a set of variables which can be accessed and updated by the automaton's actions even if these constitute a set of independent parallel threads. In the context of process algebras, the presence of independent parallel threads sharing a common set of variables creates the need to build mechanisms for state maintenance or resort to alternative means of structuring the model which can be quite taxing. In our case-study there were no parallel threads needing to access the same set of variables, which rendered such mechanisms unnecessary. Instead, processes carried and updated their store within process constant names as, for example, process constant $\mathrm{InComp}\langle i, f, s, N, \emptyset, \emptyset, \emptyset, max\rangle$.

Execution flow. Moving on, we note that the CCS model imposes a sequential structure to a node that captures its flow of execution: in the protocol's model, a node normally proceeds through the sequence of processes $\mathrm{NoLeader}$, InComp, $\mathrm{LeaderMode}$, $\mathrm{ElectedMode}$. As computation proceeds the possible behaviors a process may engage in are explicitly encoded in the process's description. On the other hand, in the I/O automata model, the flow of execution is determined by the state of an automaton: any action whose precondition is satisfied, may take place. Thus, one has to look into the code carefully to build the node's behavior as a flow diagram which can increase the effort required to debug the specification. For example, the execution flow of the above-mentioned CCS sequence of processes is realized in IOA with the following values of the state-variable tuple $\langle leader_i,\ inElection,\ sentAcktoParent\rangle$: NoLeader $\equiv \langle \bot,\ false,\ true\rangle$, InComp $\equiv \langle \bot,\ true,\ false\rangle$, LeaderMode $\equiv \langle \bot,\ true,\ true\rangle$, and ElectedMode $\equiv \langle max_i,\ false,\ true\rangle$. Obviously, one needs to carefully check the IOA specification to observe this flow, as opposed to the PA specification where it is straightforward.

Channels. Another interesting point is that the two formalisms differ in their adoption of channels: In CCS, channels are a first-class entity (see set F in the PA specification) and communication between processes is carried out by a handshake mechanism over their connecting channels. This means that if one needs to employ a more involved type of a channel (e.g. buffer or lossy channel), then special processes need to be described for connecting the original sender and receiver. In contrast, in the I/O automata model, channels are modeled as automata which execute complementary actions with their source and destination. For simple types of channels, this machinery is standard and becomes almost invisible to the main body of an application but has as a consequence that in a proof one needs to assume the proper delivery of messages, assuming of course that channels are intended to be reliable (as was the case in the IOA proofs presented in this paper).

Learning curve. A newcomer to both of the formalisms who was involved in the development of the two specifications reported the I/O automata model to be easier to understand and use. This is mainly due to the programming style of I/O automata which does not place great demands on a newcomer as opposed to the unfamiliar nature of PAs (language and semantics).

5.2 Verification

Moving on to the verification we again observe that the two proofs build on a number of common ideas (e.g. both proofs consider the case that the protocol contains a unique initiator before moving on to the general case). However, the approaches taken are quite distinct.

Global vs. local properties and Proof methods. As it can be observed in Section 4.2, the PA proof is based on the use of bisimulation for establishing the equivalence between the system and its perceived intended behavior (Theorem 4.1). As already noted, bisimulations place the emphasis on the behavior a system exhibits on the interface with its environment, that is, on global system properties. With regards to the establishment of the correctness criterion, the PA proof took advantage of the nature of the protocol: it is a deterministic protocol that essentially concerns the computation of a global function (election of the maximum leader). Given this, efforts were geared towards establishing the confluence of the system and demonstrating that there exists an execution where the desired leader election is observed.

On the other hand, as it can be observed in Section 3.2, the I/O proof uses assertional techniques for the proof of a number of safety and liveness properties which establish that in every execution, eventually, all processes will know a common leader. To achieve this, the global criterion had to be decomposed into local properties of the constituent components of the system. This task required a careful consideration of the protocol's behavior and some ingenuity on behalf of the prover. As a result, the proof had to be broken into two parts: in the first part, the internal state of the nodes was "transformed" into to a global system behavior (by the existence of a globally common spanning tree) and then, in the second part, the uniqueness of the leader was shown.

A general conclusion that emanates from this observation is that process algebras are especially suited for applications where the correctness requirement can be expressed as a global property of a system, whereas I/O Automata can more naturally handle the establishment of local properties of the component automata, or, where the overall requirement can be easily decomposed into such properties.

One may argue that perhaps it would be possible to use different IOA proof methods geared towards reasoning about global properties, e.g. simulation relations [11]. For the specific protocol, our experience tells us that this would be laborious to establish. It would be interesting to look into whether a notion similar to *confluence* would aid such reasoning. Nonetheless, we feel that it is questionable that it would result in easier-to-produce or more comprehensible proofs.

Proof style and Applicability. Looking at Sections 3.2 and 4.2, one may argue that the process calculus proof appears to be more technical in comparison to the IOA one. While, the PA proof took advantage of compositionality results for facilitating the verification process, it took some effort for the newcomer to become familiar with them as well as some ingenuity for choosing and adopting them. On the other hand, the IOA proof was more intuitive, closer to the "way of thinking" of the protocol, and did not require any specialized techniques, thus it seemed easier to apply. The challenge being to identify the appropriate safety and liveness properties (which for the specific protocol were not very difficult), the rest of the process was guided by checking for missing information towards reaching the intended goal and subsequently expressing it as additional lemmas and invariants. The proofs were mainly carried out by induction and code investigation. However, the verbose style employed in the IOA liveness proofs (which is the typical style generally used for such proofs in IOA) could allow a less mature prover to fall into pitfalls. In contrast, in the process-algebraic proof, safety and liveness properties are paired together and their proof follows the formal nature of the semantics. This results in a continuous rigidity in the proof as well as a higher awareness on the part of the prover when an argument is becoming "loose".

Learning curve. From the above discussion, perhaps it is not a surprise that the newcomer reported that the process of carrying out the proof within the I/O automata framework seemed easier than in the PA framework.

References

[1] R. M. Amadio and S. Prasad. Modelling IP mobility. In *Proceedings of CONCUR'98*, LNCS 1466, pages 301–316, 1998.

[2] J. A. Bergstra, A. Ponse, and S. A. Smolka. *Handbook of Process Algebra*. North-Holland, 2001.

[3] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L.Welch. Virtual mobile nodes for mobile ad hoc networks. In *Proceedings of DISC'04*, LNCS 3274, pages 230–244, 2004.

[4] R. Fuzzati and U. Nestmann. Much ado about nothing? *Electronic Notes of Theoretical Computer Science*, 162:167–171, 2005.

[5] M. Gelastou, Ch. Georgiou, and A. Philippou. On the application of formal methods for specifying and verifying distributed protocols. Technical Report UCY-TR-07-04, Dept. of Computer Science, University of Cyprus. (Available at `http://www.cs.ucy.ac.cy/~annap/nca-full.pdf`.)

[6] Ch. Georgiou, N. A. Lynch, P. Mavrommatis, and J. A. Tauber. Automated implementation of complex distributed algorithms specified in the IOA language. In *Proceedings of PDCS'05*, pages 128–134, 2005.

[7] J. F. Groote and M. P. A. Sellink. Confluence for process verification. In *Proceedings of CONCUR'95*, LNCS 962, pages 152–168, 2005.

[8] X. Liu and D. Walker. Confluence of processes and systems of objects. In *Proceedings of TAPSOFT'95*, LNCS 915, pages 217–231, 1995.

[9] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[10] N. A. Lynch and M. R. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[11] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[12] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

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

[14] S. Nanz and C. Hankin. Static analysis of routing protocols for ad hoc networks. In *Proceedings of WITS'04*, pages 141–152, 2004.

[15] U. Nestmann. *On Determinacy and Non-determinacy in Concurrent Programming*. PhD thesis, University of Erlangen, 1996.

[16] U. Nestmann, R. Fuzzati, and M. Merro. Modeling consensus in a process calculus. In *Proceedings of CONCUR'03*, LNCS 2671, pages 393–407, 2003.

[17] C. Newport. Consensus and collision detectors in wireless ad hoc networks. Master's thesis, MIT, 2006.

[18] A. Philippou and G. Michael. Verification techniques for distributed algorithms. In *Proceedings of OPODIS'06*, LNCS 4305, pages 172–186, 2006.

[19] A. Philippou and D. Walker. On confluence in the π-calculus. In *Proceedings of ICALP'97*, LNCS 1256, pages 314–324, 1997.

[20] B. C. Pierce and D. N. Turner. Pict: A programming language based on the π-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[21] M. Sanderson. *Proof Techniques for CCS*. PhD thesis, University of Edinburgh, 1982.

[22] C. Tofts. *Proof Methods and Pragmatics for Parallel Programming*. PhD thesis, Univ. of Edinburgh, 1990.

[23] F. W. Vaandrager. On the relationship between process algebra and input/output automata. In *Proceedings of LICS'91*, pages 387–398. IEEE Computer Society, 1991.

[24] S. Vasudevan, J. Kurose, and D. Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *Proceedings of ICNP'04*, pages 350–360. IEEE Computer Society, 2004.