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

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

Abstract

This paper studies the applicability of two methods for formally specifying and verifying distributed algorithms. Specifically, we consider the frameworks of Process Algebra and I/O Automata and we apply both towards the verification of a distributed leader-election algorithm. Based on the two experiences we evaluate the approaches and draw conclusions with respect to their relative capabilities, strengths and usability.

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 algorithms has long been realized by the research community. 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 modelling 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 algorithms (see, for example, [9, 3, 6, 17] and [1, 16, 18] for I/O automata and process algebra 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 algorithms and environments. Characteristically, [4] questions the suitability of process algebras for reasoning about certain classes of distributed algorithms.

These concerns have a great impact on the Distributed and Parallel Computing community. While the need of applying formal techniques for reasoning about algorithm correctness is generally accepted

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

by members of the community, various questions still remain open and hinder the selection and adoption of these techniques. Questions include: Which formalisms are appropriate to use for distributed/parallel algorithms? Is there one that is "clearly better" for these algorithms, or classes thereof? Which one is "easier" to learn and apply? Would a newcomer (postgraduate student) be able to apply such methods to specify and verify the distributed/parallel algorithms we develop? There is no doubt that answers to such questions can only benefit the Distributed and Parallel Computing community.

We begin to consider these questions by verifying a typical distributed algorithm in both the IOA and the PA formalisms. In particular we specify and verify a leader-election algorithm [24] with static membership and fault-free components. The choice of the algorithm 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) the algorithm is simple enough thus allowing us 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.

We observe the capabilities of each of the frameworks for modeling the specific algorithm. We apply the associated proof techniques for proving the algorithm's correctness and we evaluate them with respect to their relative capabilities, strengths and usability. We compare and present the two experiences. To the best of our knowledge, this is the *first* such hands-on evaluation of the two formalisms.

Document Structure. In the following section we give an overview of the two formalisms as well as the algorithm to be verified. Sections 3 and 4 contain the specifications and verifications of the algorithms in I/O automata and PA, respectively. In Section 5 we contrast and evaluate the two verification experiences and in Section 6 we conclude the paper with our conclusions and a discussion of future work. **Missing proofs and some useful background material can be found in the Appendix that has been sent to the PC chair, or in the full version of the paper [5].**

2 Prelimilaries

In this section we present the two formalisms that will be used for the specification and verification of the leader-election algorithm, as well as its description.

2.1 I/O Automata

In this section we overview the I/O Automata formalism of Lynch and Tuttle [10, 9] focusing on notions used in this study. For a more detailed presentation we refer to these papers as well as [5, 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 *external signature* consists only of the sets of input and output 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). We denote by $exec(A)$ the set of all executions of A. 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.

Since the input actions of an I/O automata are always enabled, the specification cannot prevent the occurrence of an infinite sequence of input actions, which could prevent the automaton from performing locally-controlled actions (internal and output actions). This can lead to executions that are not *fair*, so we are often interested in reasoning about *fair executions* defined as those executions where if the automaton enables its locally-controlled actions infinitely often then it executes them infinitely often.

We conclude this section by presenting proof-related notions used in the subsequent study. Within the I/O automata framework, the 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. In particular, it is required that something "bad" never happens. A liveness property specifies events that must *eventually* be performed. In particular, it is required that something "good" eventually happens, which in turn means that no matter what has happened up to a certain point, there is still the possibility that something good will happen. Clearly this is a property that can only be satisfied by fair executions.

An *invariant* is a property that is true in all reachable 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*, in which we reason about the behavior of the composition by reasoning about the behavior of the component automata of the composition. First, one proves less complex invariants (or properties in general) for the automata of the composition, and then it uses the composition of those invariants to reason about the composed automaton.

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, namely 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, 5].

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 $C\langle\tilde{x}\rangle \stackrel{\text{def}}{=} P$, where the process P may contain occurrences of C, as well as other constants. The syntax of CCS$_v$ is given as follows:

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

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 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 (v, α, v') signifies that it is possible to evolve from v to v' by executing action α (that is, $v \xrightarrow{\alpha} v'$ is enabled by the semantic rules).

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 states of systems. Two processes are bisimilar if, for each step of one, there is a matching (possibly multiple) step of the other, leading to bisimilar states. Below, we introduce a well-known such relation on which we base our study. First, let us recall that Q is a *derivative* of P, if there are $\alpha_1, \ldots, \alpha_n \in \mathit{Act}$, $n \geq 0$, such that $P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$. Moreover, given $\alpha \in \mathit{Act}$ we write $P \Longrightarrow Q$ for $P(\xrightarrow{\tau})^*Q$, $P \overset{\alpha}{\Longrightarrow} Q$ for $P \Longrightarrow\xrightarrow{\alpha}\Longrightarrow Q$, and $P \overset{\hat{\alpha}}{\Longrightarrow} Q$ for $P \Longrightarrow Q$ if $\alpha = \tau$ and $P \overset{\alpha}{\Longrightarrow} Q$ otherwise.

Definition 2.1 *Bisimilarity* is the largest symmetric relation, denoted by $\approx$, such that, if $P \approx Q$ and $P \xrightarrow{\alpha} P'$, there exists Q' such that $Q \overset{\hat{\alpha}}{\Longrightarrow} Q'$ and $P' \approx Q'$.

Typically, bisimulation relations are used to establish that a system satisfies its specification by describing the two as process-calculus processes and discovering a bisimulation that relates them. Their theory has been developed into two directions. On the one hand, sound and complete axiom systems have been developed for establishing algebraically the equivalence of processes. On the other hand, proof techniques that ease the task of showing two processes to be equivalent have been proposed. We point out that bisimilarity implies trace equivalence but the converse does not hold.

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. (More details on confluence can be found in Appendix A.1 and [5].)

2.3 The Algorithm

The algorithm we consider for our case study, which we hereafter call LE, is the static version of a distributed leader-election algorithm 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 algorithm 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 father 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 father 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. A more detailed description of the algorithm can be found in Appendix A.2 or in the full paper [5].

3 Specification and Verification in IOA

3.1 Specification

The specification of algorithm 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 signature, state, and transitions of the LENODE_i automaton are given in Fig. 1 and of automaton $C_{i,j}$ in Fig. 2.

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 Appendix A.3 or in the full paper [5]. Invariants are proved by induction on the length of the execution.

3.2.1 A Unique Spanning Tree is Built

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

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
$type \in \{election, ack, leader\}$; $maxid, leaderid, srcid \in I \cup \{\perp\}$;
$mychild$: $Boolean$
$i, j \in I$

Signature:

Input:
$\mathsf{receive}(m)_{j,i}$

Output:
$\mathsf{send}(m)_{i,j}$

Internal:
$\mathsf{beginComputation}_i$
$\mathsf{setAcktoParent}_i$
$\mathsf{setLeader}_i$

States:

$max_i \in I \cup \{\perp\}$, initially $\perp$
$src_i \in I \cup \{\perp\}$, initially $\perp$
$leader_i \in I \cup \{\perp\}$, initially $\perp$
$parent_i \in I \cup \{\perp\}$, initially $\perp$
$Nbrs_i \in 2^I$: 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 $tosend_i[j] = null$, $\forall j \in I$

Transitions:

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

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

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

internal setLeader$_i$
Precondition:
    $toBeAcked_i = \emptyset \wedge src_i = i \wedge sentAcktoParent_i = false$
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 1: The LENODE$_i$ automaton.

Signature:

Input:
send$(m)_{j,i}$, where $m \in \mathcal{M}$ and $i, j \in I$

Output:
receive$(m)_{i,j}$, where $m \in \mathcal{M}$ and $i, j \in I$

States:
MSG, a set of messages, initially $\emptyset$

Transitions:

```
input send$(m)_{j,i}$
Effect:
    put $m$ in $MSG$

output receive$(m)_{i,j}$
Precondition:
    $m \in MSG$
Effect:
    remove $m$ from $MSG$
```

Figure 2: The Channel Automaton $C_{i,j}$

Safety Properties

The first invariant states that once a node enters a leader-election computation, it adapts 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 $s.leader_i = \perp$ then $s.src_i = \perp$ and $s.parent_i = \perp$.

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

The next lemma states that source nodes do not appear "out of the blue". The proof is by investigation of the code and makes 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 adapts 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 $\mathsf{beginComputation}_i$ 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 *Algorithm* 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 algorithm LE.

Safety Properties

The following invariant states that a node adapts 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 the each child propagates to its parent the maximum value of its subtree. The proof is by code investigation and it makes 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 algorithm 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 algorithm in the CCS$_v$ calculus. 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, and, once the node is in computation mode, 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.

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

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

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

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

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

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

Figure 3: The node process

4.2 Correctness Proof

The correctness criterion of our algorithm 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 $P_0 \approx \overline{\mathrm{leader}}(\mathbf{max}).0$ *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 algorithm operates is predetermined. We show that this restricted system is capable of producing the required leader message and terminate. Then we observe that this system is confluent and thus it is in fact bisimilar to the process $\overline{\mathrm{leader}}(\mathbf{max}).0$. 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 Appendix A.4 or in the full paper [5].

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

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

Thus, $\mathrm{NoLeader}'$ is similar to $\mathrm{NoLeader}$ except that it may only be activated by a signal from a specified node, f. Similarly, InComp' and $\mathrm{LeaderMode}'$ are similar to InComp and $\mathrm{LeaderMode}$, 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

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

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 results shows that T_0 has an execution where the maximum node is elected as a leader.

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

Proof. The proof consists of the construction of an appropriate execution. The execution considered follows the intuitive break down of the algorithm 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 $\mathrm{NoLeader}'$, InComp', $\mathrm{LeaderMode}'$ and $\mathrm{ElectedMode}$, are confluent by construction and satisfy the remaining conditions of Theorem A.4. Thus, the result follows. □

From these two results we have that T_0 satisfies the algorithm specification.

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 $\{\langle T, P\rangle | P\ and\ T\ are\ similar\}$ *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

Having presented the models and correctness proofs of the LE algorithm in the two formalisms, in this section we evaluate the two approaches 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 algorithms. 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 the quiescent trace equivalence (an adaptation of the completed trace equivalence) and the fair trace equivalence are substitutive. Indeed, as a consequence of the input-enabledness of input actions and the non-blocking properties of the output actions, trace semantics is compositional for I/O automata. The specific semantics also enables reasoning about fairness within I/O automata models. 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. Finally, we point out that PAs have a rich algebraic structure and they are associated with axioms systems which can be used for reasoning algebraically/compositionally about system behavior.

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 on channels, 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. 1). 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. As with process algebras, such an automaton is built compositionally 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 algorithm'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 $\langle leader_i,\ inElection,\ sentAcktoParent\rangle$: $\mathrm{NoLeader} \equiv \langle \bot,\ false,\ true\rangle$, $\mathrm{InComp} \equiv \langle \bot,\ true,\ false\rangle$, $\mathrm{LeaderMode} \equiv \langle \bot,\ true,\ true\rangle$, and $\mathrm{ElectedMode} \equiv \langle max_i,\ false,\ true\rangle$. 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 (as demonstrated in Fig. 2). 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. The specifications were produced by a newcomer to both of these formalisms who reported the I/O automata model to be easier to produce. This is mainly due to the programming style of I/O automata which does not place great demands on a newcomer to the formalism 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 algorithm 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 process calculus 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. By adding an advertisement of the election of a leader in the specification of the processes, the correctness criterion was straightforward to capture in terms of the existence of a bisimulation relation. With regards to the establishment of the correctness criterion, the PA proof took advantage of the nature of the algorithm: it is a deterministic algorithm that essentially concerns the computation of a global function (election of the maximum leader). Given this fact, 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 algorithm'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 algorithm, 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 algorithm, 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 algorithm 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 the process of carrying out the proof within the I/O automata framework to be easier than in the PA framework.

6 Conclusions

The purpose of this work was to study the applicability of two prominent methods for formally specifying and verifying distributed algorithms. Specifically we specified and verified a leader-election algorithm using the Input/Output Automata and Process Algebra frameworks and we evaluated the two methods with respect to their capabilities, strengths and usability. Based on this case-study we can, in summary, conclude the following:

- Both formalisms were successful in specifying and verifying the algorithm under study. For each method, standard/natural specification style and proof techniques were employed, demonstrating that for distributed algorithms of a similar nature as the one under study, both methods are applicable.

- The correctness criterion of the algorithm consisting of a global property (a common leader is elected) as well as its confluent behavior, rendered the process-calculus proof methodology very natural to apply. This does not imply that the IOA proof has been any 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) to allow the prover to reach the desired result.

- 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 PAs 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 rigid proofs.

For future work we plan to evaluate the IOA and PA frameworks for specifying and verifying distributed algorithms in more complex systems where component failures or mobility is present. We expect that in such systems our findings will be different both with respect to the capabilities as well as the usability of the frameworks.

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 algorithms. Available at `http://www.cs.ucy.ac.cy/~annap/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.

On the Application of Formal Methods for Specifying and Verifying Distributed Algorithms

A Appendix

A.1 Confluence

In this part of the Appendix we recall some useful results regarding the theory of confluence that are employed in this study. For further information we refer the reader to [13, 22, 21, 7, 19].

In [12, 13], Milner introduced and studied a precise notion of *determinacy* of CCS processes. The same notion carries over straightforwardly to the CCS$_v$-calculus. It is expressed as follows:

Definition A.1 P is *determinate* if, for every derivative Q of P and for all $\alpha \in Act$, whenever $Q \xrightarrow{\alpha} Q'$ and $Q \overset{\hat{\alpha}}{\Longrightarrow} Q''$ then $Q' \approx Q''$.

This definition makes precise the requirement that, when an experiment is conducted on a process it should always lead to the same state up to bisimulation. Determinacy has been extended into the notion of *confluence* as follows:

Definition A.2 P is *confluent* if it is determinate and, for each of its derivatives Q and distinct actions α, β, where α and β are not input actions on the same channel, if $Q \xrightarrow{\alpha} Q_1$ and $Q \overset{\beta}{\Longrightarrow} Q_2$ then, there are Q_1' and Q_2' such that $Q_2 \overset{\hat{\alpha}}{\Longrightarrow} Q_2'$, $Q_1 \overset{\hat{\beta}}{\Longrightarrow} Q_1'$ and $Q_1' \approx Q_2'$.

Its essence, to quote [13], is that “of any two possible actions, the occurrence of one will never preclude the other”. As shown in [13, 12], for pure CCS processes 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. These results were extended and generalized in various other calculi (see, for example, [7, 22, 8, 15, 19, 20, 18]).

In this work, we will employ the following additional notion and result for aiding the verification process [19].

Definition A.3 P is *o-determinate* if, for every derivative Q of P and for all channels a, whenever $Q \xrightarrow{\overline{a}(\tilde{x})} Q'$ and $Q \overset{\overline{a}(\tilde{y})}{\Longrightarrow} Q''$, then $\tilde{x} = \tilde{y}$ and $Q' \approx Q''$.

Theorem A.4 Suppose $P = (P_1 \mid \ldots \mid P_n)\backslash L$, where each P_j is confluent and o-determinate and each channel in L is used by at most two components of the composition. Then P is confluent.

A.2 Algorithm LE

In this part of the appendix we present the details of algorithm LE.

Each node i operates as follows:

- If i realizes that it has lost its leader, then it moves to *computation mode* in order to select a new leader. It broadcasts an $election$ message to all its neighbors which are considered its potential children. This message contains the node's identifier, id, which is considered to be the source identifier of the computation, $scrid$, and denotes which node has started this procedure. In this case source id is equal to i. Then it waits to receive acknowledgment messages, ack, from all its neighbors.

- If i receives an $election$ message from a neighbor j and it is not in computation mode then it sets j to be its parent and enters computation mode. It then forwards the $election$ message to all its neighbors except its parent. All these neighbors are considered its potential children and it waits to receive ack messages from them.
- If i receives an $election$ message and it is already in computation mode with $scrid$ smaller than the source identifier contained in the message, then it abandons its current computation and proceeds according to the previous step.
- If i receives an $election$ message and it is already in computation mode with $srcid$ equal to the source identifier contained in the message, then it replies with an ack message informing the sender that it is already in computation mode with a different parent node.
- If i receives an $election$ message and it is already in computation mode with $srcid$ larger than the source id contained in the message, then it simply ignores the message.
- For any ack message that i receives, it removes the sender from its children list. The content of the message can be distinguished in two categories. Either the sender is informing i that it does not accept i as its parent, in which case i simply continues its computation. Otherwise, this message contains an identifier which i compares with its known *maximum value*, initially set as i's identifier, and keeps as its known maximum value the largest of the two.
- When i receives ack messages from all its children or if it does not have any children, then it sends an ack message to its parent. With this message it informs its parent about maximum value/node identifier it is aware of. In the case that i has no children, then this identifier is i itself.
- If i is the node that started the computation (thus node i is the root of the spanning tree that was created) and it has received ack messages from all its children nodes, then it decides which is the leader node (the one with the maximum value) and informs all its neighbors about the new leader node through a $leader$ message.
- If a $leader$ message is received and i has not learned its leader yet, it adapts the leader contained in the message and forwards the message to all its neighbors (except from the sender of this message).
- If a $leader$ message is received and i already has a leader then it simply ignores this message.

A.3 The Input/Output Automata Omitted Proofs

In this part of the appendix we present the missing proofs from the I/O automata correctness proof of the algorithm.

Proof of Invariant 1

We first prove part (a) of the invariant. The proof is by induction on the length of the execution. The invariant holds in the base case since initially $s_0.inElection_i = false$, $s_0.leader_i = \perp$, $s_0.src_i = \perp$ and $s_0.parent_i = \perp$. Let the invariant hold for state s and consider step (s, π, s'). If $\pi = \mathsf{send}(m)_{i,j}$ or $\mathsf{setAcktoParent}_i$ then $s'.inElection_i = s.inElection_i$, $s'.leader_i = s.leader_i$, $s'.src_i = s.src_i$ and $s'.parent_i = s.parent_i$ thus the statement holds by the inductive hypothesis. For the rest of the cases:

- If $\pi = \mathsf{beginComputation}_i$ then, by the preconditions of π it holds that $s.inElection_i = false$ and $s.leader_i = \perp$ and by the inductive hypothesis, $s.src_i = \perp$ and $s.parent_i = \perp$. From the effects of π, we get that $s'.inElection_i = true$, $s'.src_i = i$ and $s'.parent_i = i$ thus the statement holds.

- If $\pi = \mathsf{setLeader}_i$ then by the effects of this action $s'.inElection_i = false$ and $s'.leader_i \neq \perp$ hence the statement holds.

- If $\pi = \mathsf{receive}(m)_{j,i}$ and $m.type = ack$ or $m.type = leader$ then $s'.src_i = s.src_i$ and $s'.parent_i = s.parent_i$ thus the statement holds by the inductive hypothesis.

- If $\pi = \mathsf{receive}(m)_{j,i}$ and $m.type = election$ and $s.inElection_i = true$ then $s'.inElection_i = true$ thus the statement holds.

- If $\pi = \mathsf{receive}(m)_{j,i}$ and $m.type = election$ and $s.inElection_i = false$, then from the effects of this action we have that $s'.inElection_i = true$, thus the statement holds.

We now prove part (b) of the invariant. The proof is by induction on the length of the execution. Initially $s_0.inElection_i = false$ thus the statement trivially holds. Let the invariant hold for state s and consider step (s, π, s'). If $\pi = \mathsf{send}(m)_{i,j}$ or $\mathsf{setAcktoParent}_i$ then $s'.inElection_i = s.inElection_i$, $s'.src_i = s.src_i$ and $s'.parent_i = s.parent_i$ thus the statement holds by the inductive hypothesis. For the rest of the cases:

- If $\pi = \mathsf{beginComputation}_i$ then, by the preconditions of π it holds that $s.inElection_i = false$. From the effects of π, we get that $s'.inElection_i = true$, $s'.src_i = i$ and $s'.parent_i = i$ thus the invariant is re-established.

- If $\pi = \mathsf{setLeader}_i$ then by the effects of this action $s'.inElection_i = false$ hence the statement holds.

- If $\pi = \mathsf{receive}(m)_{j,i}$ and $m.type = ack$ or $m.type = leader$ and $s.inElection_i = false$ then $s'.inElection_i = s.inElection_i$, $s'.src_i = s.src_i$ and $s'.parent_i = s.parent_i$ thus the statement holds by the inductive hypothesis.

- If $\pi = \mathsf{receive}(m)_{j,i}$ and $m.type = leader$ and $s.inElection_i = true$, then $s'.inElection_i = false$ and hence the statement holds.

- If $\pi = \mathsf{receive}(m)_{j,i}$ and $m.type = election$ and $s.inElection_i = true$ then by the inductive hypothesis $s.src_i \neq \perp$ and $s.parent_i \neq \perp$. From the effects of π, $s'.inElection_i = true$ and $s'.src_i \neq \perp$ and $s'.parent_i \neq \perp$, thus the invariant is re-established.

- If $\pi = \mathsf{receive}(m)_{j,i}$ and $m.type = election$ and $s.inElection_i = false$, then by part (a) of this invariant it holds that $s.src_i = \perp$ and $s.parent_i = \perp$. From the effects of π we have that $s'.inElection_i = true$, $s'.src_i = m.srcid$ and $s'.parent_i = j$, thus the invariant is re-established.

This completes the proof. □

Proof of Lemma 3.1

The proof is by investigation of the code. Since the initial value of $s_0.src_i = \perp$ we have to check under which cases node i changes the value of src_i. From the code there are two cases:

- In the internal action $\mathsf{beginComputation}_i$. By the preconditions of this action, for a state $s_1 < s$, $s_1.inElection_i = false$ and $s_1.leader_i = \perp$, thus from Invariant 1(a), $s_1.src_i = \perp$. From the effects of π we get $s_2.src_i = j$ where $i = j$ as required.
- In input action $\mathsf{receive}(m)_{k,i}$ where $m.type = election$ and $m.srcid = j$, $i \neq j$ and for a state $s' < s$ such that $s'.inElection_i = false$ or $s'.inElection_i = true \wedge m.srcid > src_i$. This implies a preceding $\mathsf{send}(m)_{k,i}$ event such that $m.type = election \wedge m.srcid = j$. From the code we find two cases for such action to occur:
  - In internal action $\mathsf{beginComputation}_k$ where $k = j$, thus there exists a step (s_1, π, s_2), $s_1, s_2 < s'$ such that $\pi = \mathsf{beginComputation}_j$ as shown in the first bullet above.
  - In input action $\mathsf{receive}(m)_{\ell,k}$ where $m.type = election$, $m.srcid = j$, $k \neq j$ and $inElection_k = false$ or $inElection_k = true \wedge m.srcid > src_k$. The proof continues recursively on ℓ. □

Proof of Invariant 2

The proof is by induction on the length of the execution. The base case is trivial since $parent_i = \perp$, $\forall\, i \in I$. Let the invariant hold for state s and consider step (s, π, s'). If $\pi = \mathsf{send}(m)_{i,j}$, $\mathsf{setAcktoParent}_i$ or $\mathsf{setLeader}_i$ then $s'.parent_i = s.parent_i$ thus the statement holds by the inductive hypothesis. For the rest of the cases we have:

- If $\pi = \mathsf{beginComputation}_i$ then it must be that $i = i_0$ by our assumption. Moreover from the preconditions of π we have that $s.inElection_i = false$ and $s.leader_i = \perp$ and by Invariant 1(a) we get that $s.parent_i = \perp$. Since no other $\mathsf{beginComputation}_j$ occurred before state s, it holds also that $s.parent_j = \perp, \forall\, j \in I$. From the effects of π, $s'.parent_{i_0} = i_0$. Thus i_0 is the only node in the spanning tree and is considered to be the root of this spanning tree.
- If $\pi = \mathsf{receive}(m)_{j,i}$ then there are the following cases:
  - If $m.type = ack$ or $m.type = leader$ then $s'.parent_i = s.parent_i$ thus the statement holds by the inductive hypothesis.
  - If $m.type = election$ and $s.inElection_i = false$ then, by the effects of this action we have that $s'.parent_i = j$ and $s'.srcid = m.srcid$. This implies a preceding $\mathsf{send}(m)_{j,i}$ event such that $m.type = election$. From the code we have that such messages are sent by processes that are in election and hence by Invariant 1(b) and Lemma 3.1 we have that $parent_j \neq \perp$ and $m.srcid = i_0$. Thus j belongs to the spanning tree rooted at i_0 according to the inductive hypothesis. Since i is a neighbor of j (by the preconditions of the action $\mathsf{send}(m)_{j,i}$) then the new edge defined by $parent_i$ extends the spanning tree to include node i. Moreover, since $s.inElection_i = false$ and by Invariant 1(a) it holds that $s.parent_i = \perp$ thus node i did not belong to the spanning tree previously and in addition to the uniqueness of the variable $parent_i$ we conclude that i cannot cause loops in the spanning tree.

  - If $m.type = election$ and $s.inElection_i = true$ and $m.srcid = s.src_i$ then $s'.parent_i = s.parent_i$ thus the statement holds by the inductive hypothesis.
  - If $m.type = election$ and $s.inElection_i = true$ and $m.srcid > s.src_i$, from Lemma 3.1 we have that there must have existed a step (s_1, π', s_2), $s_1, s_2 < s$ where $\pi' = \mathsf{beginComputation}_{m.srcid}$ and $m.srcid \neq i_0$. But this contradicts the assumption of unique initiator and hence this case is not possible.

This completes the proof. □

Proof of Invariant 3

The proof is an induction on the length of the execution. The base case holds trivially, as initially, $\forall\, i \in I$, $s_0.src_i = \perp$. We suppose that the invariant holds for state s and we examine step (s, π, s'). If $\pi = \mathsf{send}(m)_{i,j}$, $setAcktoParent_i$ or $setLeader_i$ then $s'.src_i = s.src_i$ and the statement holds. For the remaining cases we have:

- If $\pi = \mathsf{beginComputation}_i$ then from the preconditions of π we have that $s.inElection_i = false$ and $s.leader_i = \perp$ and by Invariant 1(a) we get that $s.src_i = \perp$. By the effects of π, $s'.parent_i = i$ and hence the invariant is re-established (by convention, $i > \perp, \forall\, i \in I$).
- If $\pi = \mathsf{receive}(m)_{j,i}$ then there are the following cases:
  - If $m.type = ack$ or $m.type = leader$ then $s'.src_i = s.src_i$ thus the statement holds.
  - If $m.type = election$ and $s.inElection_i = false$ then by Invariant 1(a) we have that $s.src_i = \perp$ and by the effects of this action we get that $s_1.srcid = m.srcid$. This implies a preceding $\mathsf{send}(m)_{j,i}$ event that by the inductive hypothesis, $m.type = election$ and $m.srcid > \perp$. Hence the invariant is re-established.
  - If $m.type = election$ and $s.inElection_i = true$ we notice that $s'.src_i = m.srcid$ only if $m.srcid > s.src_i$. Hence $s'.src_i > s.src_i$ as required.

This completes the proof. □

Proof of Lemma 3.2

For any node $j \in I$, we denote as D_j the length in hops of the maximum path among the set of loop-free paths from i_0 to j. We will prove that eventually j belongs in the spanning tree rooted at i_0. The proof is by induction on D_j. For $D_j = 0$, let s_0 be an initial state and a step (s_0, π, s_1) such that $\pi = \mathsf{beginComputation}_{i_0}$. Notice that π is the only action possible in $exec_{i_0}$. From the effects of action π we get that $s_1.src_i = i_0$, $s_1.inElection_i = true$, $s_1.parent_i = i_0$ and some messages m such that $m.type = election$ and $m.srcid = i_0$ are prepared to be sent to the neighbors of i_0. ¿From Invariant 2 i_0 forms a spanning tree rooted at i_0 thus the statement holds.

Assume that for any k, $0 < k < D_j$, any node u such that $D_u \leq k$ belongs to the spanning tree rooted at i_0. For $k + 1 = D_j$, we have two cases:

- j is a neighbor of i_0, hence j has received or receives a message m from i_0 such that $m.type = election$ and $m.srcid = i_0$.

- j is a neighbor of a node $v \neq i_0$ such that $D_v \leq k$. By the induction hypothesis, v belongs to the spanning tree rooted at i_0. This implies a step (s, π, s') such that $\pi = \mathsf{receive}(m)_{u,v}$ where $m.type = election$ and $inElection_v = false$. From the effects of π, a set of messages m are sent to the neighbors of v, including j such that $m.type = election$ and $m.srcid = i_0$.

Upon receiving m from v, that is $\pi = \mathsf{receive}(m)_{v,j}$ there are two cases for process j:

- j is in election, that is, $inElection_j = true$. Then $parent_j \neq \perp$ and since only node i_0 started the computation, by Invariant 2 j already belongs to the spanning tree rooted at i_0 and hence the statement holds.

- j is not in election, that is, $inElection_j = false$. By the effects of π, $src_j = i_0$ and $parent_j = v$. By the inductive hypothesis v belongs to the spanning tree and per Invariant 2, $parent_j$ forms an edge of the spanning tree rooted at i_0. Hence process j belongs to the spanning tree as desired.

Since j is an arbitrary node of the network, we conclude that every $j \in I$ eventually belongs in the spanning tree rooted at i_0 in at most $D = \max_{j \in I} D_j$ hops. □

Proof of Theorem 3.3

If only one process executes $\mathsf{beginComputation}_i$ then $i = i_{smax}$ and by Lemma 3.2, eventually a spanning tree covering all the network will be built rooted at i. Trivially, this spanning tree is unique.

Assume that exactly two processes i_0, i_1 begin computation and without loss of generality let $i_0 > i_1$ thus $i_0 = i_{smax}$. By Lemma 3.2, each one of these computations tends to cover the whole network. Hence at least one node receives election messages for both computations. Let s the first state in which a process i that belongs to the one computation receives an election message to enter the second computation. For each state $s' < s$, we can find a partition of the network such that in each part only one $\mathsf{beginComputation}_i$ occurs. Thus, we can apply Lemma 3.2 in each subgraph of the network, hence there will be two spanning trees under formation, covering different parts of the network.

At state s there is a process i such that $s.src_i \neq \perp$ and receives a message m s.t. $m.type = election$ and $m.srcid \neq s.src_i$. By Invariant 3, i will change the value of the src_i variable only if $m.srcid > s.src_i$. In other words, if process i belongs to the spanning tree of i_1 then by Invariant 3 it changes the value of src_i variable to i_0, hence it enters the spanning tree of i_0. If process i belongs to the spanning tree of i_0 then by Invariant 3 it will not change the value of src_i variable to i_1 since $i_1 < i_0$, hence i remains in the spanning tree of i_0. The same holds for every process j that receives election messages for both computations, thus the spanning tree of i_{smax} is never blocked by any other spanning tree in the network. Thus, by Lemma 3.2 eventually every process in the network belongs in the spanning tree of i_{smax} and this spanning tree is unique.

The case of two starting processes can be easily generalized to any number of starting processes and the result follows. □

Proof of Invariant 4

The proof of this invariant is done in a similar manner to the proof of Invariant 3. The only difference is that in the inductive step while considering action $\pi = \mathsf{receive}(m)_{j,i}$, we need to investigate $m.type = ack$ instead of $m.type = election$ and specifically the case where $m.srcid = src_i$. □

Proof of Lemma 3.4

The proof is by code investigation. Initially $toBeAcked_i = \emptyset$ and $sentAcktoParent_i = true$. From the code we observe that variable $sentAcktoParent_i$ is set to $false$ in two cases:

- In the internal action $\mathsf{beginComputation}_i$ where $toBeAcked_i$ is set to $Nbrs_i$ and
- In the input action $\mathsf{receive}(m)_{j,i}$ where $m.type = election$ and $inElection_i = false$ or $inElection_i = true \wedge m.srcid > src_i$. In the second case $toBeAcked_i = Nbrs_i - \{j\}$.

In both cases above $toBeAcked_i \neq \emptyset$ unless $Nbrs_i = \emptyset$ or $Nbrs_i = \{j\}$ respectively. In such cases, $max_i = i$ and the lemma holds. In any other case, each $k \in toBeAcked_i$ has to be removed. A process k is removed from $toBeAcked_i$ only if an input action $\mathsf{receive}(m)_{k,i}$ occurs where $m.type = ack$, $sentAcktoParent_i = false$ and $m.srcid = src_i$. Moreover in the same action, if $m.mychild = true$ and $m.maxid > max_i$ then $max_i = m.maxid$. As a result of these actions and by the Invariant 4, when $toBeAcked_i = \emptyset$ then max_i is the greatest value among i and the values that i has "seen" from its children. □

Proof of Theorem 3.5

The proof is by induction on the length of the execution. The base case holds trivially, as initially $\forall i \in I$, $leader_i = \perp$. Assume that the statement holds for a state s and we examine step (s, π, s'). If $\pi = \mathsf{beginComputation}_i$, $\mathsf{send}(m)_{i,j}$ or $\mathsf{setAcktoParent}_i$ then $s'.leader_i = s.leader_i$ and the statement holds. For the remaining cases:

- $\pi = \mathsf{setLeader}_i$. One of the preconditions of this action requires that $s.src_i = i$. This holds only for the root of the spanning tree, which by Theorem 3.3 is unique. Furthermore, by the preconditions it holds that $s.toBeActed_i = \emptyset$ and $s.sentAcktoParent_i = false$. By Lemma 3.4 we have that $s.max_i$ is the greatest value among i and the values that i has "seen" from its children.

  Since i is the root of the spanning tree, $s.max_i$ is the maximum value of all nodes in the network, hence $s.max_i = i_{max}$. From the effects of π we have that $s'.leader_i = s.max_i = i_{max}$, and the statement holds.

- $\pi = \mathsf{receive}(m)_{j,i}$ where $m.type = leader$, $m.srcid = src_j$, $m.leaderid = leader_j$ and $s.inElection_i = true$. Since $s.inElection_i = true$ then this leader message is the first received by i thus $s'.leader_i = m.leaderid$. Since m was sent by j at a prior state, by inductive hypothesis, $m.leaderid = i_{max}$, thus $s'.leader_i = i_{max}$ and the statement holds.

- $\pi = \mathsf{receive}(m)_{j,i}$ where $m.type \neq leader$, then $s'.leader_i = s.leader_i$ and the statement holds.

This completes the proof. □

Proof of Theorem 3.6

Starting from an initial state s_0 the only possible action to occur is the $\mathsf{beginComputation}_i$ action. From Theorem 3.3 we have that eventually a unique spanning tree is built rooted at a node i_{smax}. Then, from the code it can be observed that $\forall\, i \in I$, $inElection_i = true$ and $sentAcktoParent_i = false$.

We denote as δ_j the depth of node j in the spanning tree and δ_{tree} the depth of the spanning tree. Fix a node $j \neq i_{smax}$. We prove that eventually j sends a message to its parent node such that $m.type = ack$,

$m.mychild = true$ and $m.maxid$ is the maximum value among j and the values that j has "seen" from its children. The proof is by induction on δ_{tree}. The base case is when $\delta_j = \delta_{tree}$, that is j is a leaf of the spanning tree. In that case $toBeAcked_j = \emptyset$. Since $j \neq i_{smax}$ and $sentAcktoParent_i = false$ then the preconditions of $\mathsf{setAcktoParent}_j$ are satisfied. By the effects of this action a message m such that $m.type = ack$ and $m.mychild = true$ is sent to node $parent_j$. Trivially, $m.maxid = j$.

Assume that the statement holds for any $\delta_j < k < \delta_{tree}$. That is, every node u with $\delta_u > \delta_j$ eventually sends a message to its parent node such that $m.type = ack$, $m.mychild = true$ and $m.maxid$ is the maximum value among j and the values that j has "seen" from its children. Since each child u of j has $\delta_u = k > \delta_j$, by the inductive hypothesis u eventually sends to j a message m such that $m.type = ack$, $m.mychild = true$ and $m.maxid$ is the maximum value of the subtree of u. In the worst case, $k-1 = \delta_j$, j has collected from all its children such messages m. Upon receiving such a message m from u, j removes u from $toBeAcked_j$ and changes the value of max_j only if the maximum value of the subtree of u is greater than max_j according to Invariant 4. Hence, at $k-1 = \delta_j$ hops, $toBeAcked_j = \emptyset$ and per Lemma 3.4, max_j is the greatest value among j and the values that j has "seen" from its children. Thus the preconditions of $\mathsf{setAcktoParent}_j$ are satisfied and by the effects of this action a message m such that $m.type = ack$, $m.mychild = true$ and $m.maxid = max_j$ is sent to node $parent_j$ and the statement holds.

Since j is an arbitrary node of the network, we conclude that every $j \neq i_{smax}$ eventually sends a message to their parent node such that $m.type = ack$, $m.mychild = true$ and $m.maxid$ is the maximum value of their subtree. Hence, eventually, i_{smax} receives these messages from all its children and $toBeAcked_{i_{smax}}$ becomes empty. This enables the internal action $\mathsf{setLeader}_{i_{smax}}$ that sets $leader_{i_{smax}} \neq \perp$, and particularly, per Theorem 3.5, $leader_{i_{smax}} = i_{max}$.

Then, i_{smax} broadcasts a message m s.t. $m.type = leader$ and $m.leaderid = i_{max}$ to its neighbors. Its neighbors, upon receiving a message m for the first time, that is, $inElection = true$, set $leader = m.leaderid = i_{max}$, $inElection = false$ and forward the message to their neighbors. Given that the graph is connected, this message is received by all nodes, in D hops in the worst case, where D is the length of the maximum path among the sets of loop-free paths from i_{smax} to any node i. □

A.4 The Process Algebra Omitted Proofs

In this part of the appendix we present the missing details and proofs from the process-algebraic correctness proof of the algorithm. We begin with a useful lemma.

Lemma A.5 *Let P be an arbitrary derivative of P_0:*

$$\begin{aligned} P \overset{\text{def}}{=} \; & (\prod_{m\in M_1} \text{NoLeader}\langle u_m, N_m\rangle \mid \prod_{m\in M_2} \text{InComp}\langle u_m, f_m, s_m, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m\in M_3} \text{LeaderMode}\langle u_m, s_m, N_m\rangle \mid \prod_{m\in M_4} \text{ElectedMode}\langle u_m, s_m, N_m, S_m, l_m\rangle)\backslash F \end{aligned}$$

and $M = \{m \in M_2 \cup M_3 \cup M_4 | s_m = max(M_2, M_3, M_4)\}$. Then $\{(u_m, f_m) | m \in M\}$ is a spanning tree of the nodes in M.

Proof. The proof is by induction on the length, n, of the derivation $P_0 \Longrightarrow P$. For $n = 0$, $M = \emptyset$ and the proof follows. Now suppose that the claim holds for $n = k-1$ and consider a derivation $P_0 \Longrightarrow P$

of length $n = k$. It then holds that $P_0 \Longrightarrow P' \xrightarrow{\tau} P$ where the claim holds for P'. Let us write

$$\begin{aligned} P' \overset{\text{def}}{=} & (\prod_{m \in M'_1} \text{NoLeader}\langle u_m, N_m\rangle \mid \prod_{m \in M'_2} \text{InComp}\langle u_m, f_m, s_m, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m \in M'_3} \text{LeaderMode}\langle u_m, s_m, N_m\rangle \mid \prod_{m \in M'_4} \text{ElectedMode}\langle u_m, s_m, N_m, S_m, l_m\rangle)\backslash F \end{aligned}$$

$mx = max(M'_2, M'_3, M'_4)$ and $M' = \{m \in M'_2 \cup M'_3 \cup M'_4 | s_m = mx\}$. Then $\{(u_m, f_m) | m \in M'\}$ is a spanning tree of the nodes in M'. The proof is a case analysis on the transition $P' \xrightarrow{\alpha} P$. We consider the two most interesting cases:

- $\alpha = \tau$, where

  $$NoLeader\langle u_k, N_k\rangle \xrightarrow{\tau} \text{InComp}\langle u_k, u_k, u_k, N_k, N_k, N_k, \emptyset, u_k\rangle$$

  and $u_k > mx$. Then $M' = \{u_k\}$ and the claim follows.

- $\alpha = \tau$ and for some $x \in M'_2$, $y \in M'_1 \cup M'_3 \cup M'_4$,

  $$\text{InComp}\langle u_x, f_x, mx, N_x, S_x, R_x, A_x, max_x\rangle \xrightarrow{\overline{election_{u_x,u_y}}(mx)} \text{InComp}\langle \ldots, S_x - \{u_y\}, \ldots\rangle$$

  and
  $$X \xrightarrow{election_{u_x,u_y}(\mathbf{max})} \text{InComp}\langle u_y, u_y, mx, N_y, N_y - \{u_x\}, N_y - \{u_x\}, \emptyset, u_y\rangle,$$

  where X is one of $\text{NoLeader}\langle u_y, N_y\rangle$, $\text{InComp}\langle u_y, f_y, s_y, N_y, S_y, R_y, A_y, max_y\rangle$, $\text{LeaderMode}\langle u_y, s_y, n_y\rangle$, where $s_y < mx$. Then, $M_2 = M_2 \cup \{u_y\}$ and $M_1 = M_1 - \{u_y\}$ $M_3 = M_3 - \{u_y\}$, $M_4 = M_4 - \{u_y\}$, whereas $M = \{m \in M_2 \cup M_3 \cup M_4 | s_m = mx\} = M' \cup \{u_y\}$. Furthermore, $\{(u_m, f_m) | m \in M\} = \{(u_m, f_m) | m \in M'\} \cup \{(u_x, u_y)\}$. Note that since $u_y \notin M'$, this latter set forms a spanning tree of the nodes in M. This completes the proof. □

Proof of Lemma 4.2

Let D be the maximum distance of a node from the root ν of the spanning tree. Fix sets $M_d, 0 \leq d \leq D$, such that:

$$M_d = \begin{cases} \{\nu\} & d = 0 \\ \{i \in K | f_i \in M_{d-1}\} & d > 0 \end{cases}$$

In other words, M_1 contains the nodes that have ν as their father, M_2 the nodes whose father is a node of M_1, and so on. Further, let us write $Ch_i = \{j \mid f_j = i\}$ and T^d, $0 \leq d \leq D$ for the process

$$\begin{aligned} T^d \overset{\text{def}}{=} & (\prod_{i \in M_0 \cup \ldots \cup M_{d-1}} \text{InComp}'\langle u_i, f_i, \nu, N_i, N_i - Ch_i, N_i - \{f_i\}, \emptyset, u_i\rangle \\ & \mid \prod_{i \in M_d} \text{InComp}'\langle u_i, f_i, \nu, N_i, N_i - \{f_i\}, N_i - \{f_i\}, \emptyset, u_i\rangle \\ & \mid \prod_{i \in M_{d+1} \cup \ldots \cup M_D} \text{NoLeader}'\langle u_i, f_i, N_i\rangle)\backslash F \end{aligned}$$

We will show that

$$T_0 = T^0 \Longrightarrow T^1 \Longrightarrow \ldots \Longrightarrow T^D.$$

To begin with, note that $M_{d+1} = \bigcup_{i \in M_d} Ch_i$. Furthermore, for any $i \in M_d$, if $Ch_i = \{j_1, \ldots, j_i\}$, we have that

$$\begin{array}{rcl}
\mathrm{InComp}'\langle u_i, \ldots\rangle & \xrightarrow{\overline{election_{i,j_1}}(\nu)} & \ldots \\
& \xrightarrow{\overline{election_{i,j_i}}(\nu)} & \prod_{k \in N - Ch_i} \mathrm{InComp}'\langle u_i, f_i, \nu, N_i, N_i - Ch_i, N_i - \{f_i\}, \emptyset, u_i\rangle \\
\mathrm{NoLeader}'\langle u_{j_1}, u_i, N_{j_1}\rangle & \xrightarrow{election_{i,j_1}(s)} & \mathrm{InComp}'\langle u_{j_1}, u_i, \nu, N_{j_1}, N_{j_1} - \{u_i\}, N_{j_1} - \{u_i\}, \emptyset, l_{j_1}\rangle \\
& \vdots & \\
\mathrm{NoLeader}'\langle u_{j_i}, u_i, N_{j_i}\rangle & \xrightarrow{election_{i,j_i}(s)} & \mathrm{InComp}'\langle u_{j_i}, u_i, \nu, N_{j_i}, N_{j_i} - \{u_i\}, N_{j_i} - \{u_i\}, \emptyset, l_{j_i}\rangle
\end{array}$$

Consequently, we have that for any d, $T^d \Longrightarrow T^{d+1}$.

At this point, all pending $election$ messages can be emitted, and the corresponding ack_0 acknowledgements returned, yielding

$$\begin{array}{rcl}
T^D & \Longrightarrow & (\prod_{i \in K - M_D} \mathrm{InComp}'\langle u_i, f_i, \nu, N_i, \emptyset, N_i - Ch_i, \emptyset, u_i\rangle \\
& & \mid \prod_{i \in M_D} \mathrm{InComp}'\langle u_i, f_i, \nu, N_i, \emptyset, \emptyset, \emptyset, u_i\rangle)\backslash F.
\end{array}$$

Now, let us write R^d, $0 \le d \le D$, for the process

$$\begin{array}{rcl}
R^d & \stackrel{\text{def}}{=} & (\prod_{i \in M_0 \cup \ldots \cup M_{d-1}} \mathrm{InComp}'\langle u_i, f_i, \nu, N_i, \emptyset, N_i - Ch_i, \emptyset, u_i\rangle \\
& & \mid \prod_{i \in M_d} \mathrm{InComp}'\langle u_i, f_i, \nu, N_i, \emptyset, \emptyset, \emptyset, u_i\rangle \\
& & \mid \prod_{i \in M_{d+1} \cup \ldots \cup M_D} \mathrm{LeaderMode}'\langle u_i, f_i, \nu, N_i, max_i\rangle)\backslash F
\end{array}$$

where max_i is the maximum identifier of all nodes in the subtree rooted at node i. We will show that

$$T^D = R^D \Longrightarrow \ldots \Longrightarrow R^0 .$$

In particular, for any $0 \le d < D$, and $i \in M_{d-1}$, if $Ch_i = \{j_1, \ldots, j_i\}$ we have that

$$\begin{array}{rcl}
\mathrm{InComp}'\langle u_i, f_i, \nu, N_i, \emptyset, N_i - Ch_i, \emptyset, u_i\rangle & \xrightarrow{ack1_{i,j_1}(max_{j_1})} & \ldots \\
& \xrightarrow{ack1_{i,j_i}(max_{j_i})} & \mathrm{InComp}'\langle u_i, f_i, \nu, N, \emptyset, \emptyset, \emptyset, m_i\rangle \\
\mathrm{InComp}'\langle u_{j_1}, u_i, \nu, N_{j_1}, \emptyset, \emptyset, \emptyset, max_{j_1}\rangle & \xrightarrow{\overline{ack1_{i,j_1}}(max_{j_1})} & \mathrm{LeaderMode}'\langle u_{j_1}, u_i, \nu, N_{j_1}\rangle \\
& \vdots & \\
\mathrm{InComp}'\langle u_{j_i}, u_i, \nu, N_{j_i}, \emptyset, \emptyset, \emptyset, max_{j_i}\rangle & \xrightarrow{\overline{ack1_{i,j_i}}(max_{j_i})} & \mathrm{LeaderMode}'\langle u_{j_i}, u_i, \nu, N_{j_i}\rangle
\end{array}$$

where $m_i = max\{u_i, max_{j_1}, \ldots, max_{j_i}\}$. Consequently, we have that for any d, $R^d \Longrightarrow R^{d-1}$. It is now trivial to see that R^0 can produce the required transition

$$R^0 \xrightarrow{\overline{\text{leader}}(\mathbf{max})} S_0$$

where

$$S_0 \stackrel{\text{def}}{=} (\text{ElectedMode}\langle \nu, \nu, \nu, N, N, \mathbf{max}\rangle \mid \prod_{i \neq \nu} \text{LeaderMode}'\langle u_i, f_i, \nu, N_i\rangle)\backslash F$$

It is now straightforward to verify that after a number of communications along channels $leader_{i,j}$, the system will evolve into state

$$(\prod_{i \in K} \text{ElectedMode}\langle u_i, f_i, \nu, N_i, \emptyset, \mathbf{max}\rangle)\backslash F \approx \mathbf{0}$$

which completes the proof. □

The following definition gives a precise explanation of the notion of similarity between agents.

Definition A.6 *Let*

$$\begin{aligned} P \stackrel{\text{def}}{=} & (\prod_{m \in M_1} \text{NoLeader}\langle u_m, N_m\rangle \mid \prod_{m \in M_2} \text{InComp}\langle u_m, f_m, s_m, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m \in M_3} \text{LeaderMode}\langle u_m, s_m, N_m\rangle \mid \prod_{m \in M_4} \text{ElectedMode}\langle u_m, s_m, N_m, S_m, l_m\rangle)\backslash F \end{aligned}$$

$$\begin{aligned} T \stackrel{\text{def}}{=} & (\prod_{m \in M_1'} \text{NoLeader}'\langle u_m, f_m, N_m, l_m\rangle \mid \prod_{m \in M_2'} \text{InComp}'\langle u_m, f_m, \nu, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m \in M_3'} \text{LeaderMode}'\langle u_m, \nu, N_m\rangle \mid \prod_{m \in M_4'} \text{ElectedMode}\langle u_m, \nu, N_m, S_m, l_m\rangle)\backslash F \end{aligned}$$

where, $\{M_1, M_2, M_3, M_4\}$ *and* $\{M_1', M_2', M_3', M_4'\}$ *are partitions of set* K*,* $\{(u_m, f_m) \mid m \in K\}$ *forms a spanning tree of the network rooted at* ν*, and*

$$\begin{aligned} \nu &= max(M_2 \cup M_3 \cup M_4) \\ M_1' &= M_1 \cup \{u \mid u \in (M_2 \cup M_3 \cup M_4), s_u \neq \nu\} \\ M_2' &= \{u \in M_2 \mid s_u = \nu\} \\ M_3' &= \{u \in M_3 \mid s_u = \nu\} \\ M_4' &= \{u \in M_4 \mid s_u = \nu\} \end{aligned}$$

Then we say that P *and* T *are* similar *processes.*

Proof of Lemma 4.5

Let $\mathcal{R} = \{\langle T, P\rangle | P \textit{ and } T \textit{ are similar}\}$. Consider processes T and P with $(T, P) \in \mathcal{R}$ and suppose that $T \xrightarrow{\alpha} T'$. We will show that $P \xrightarrow{\alpha} P'$ and $(T', P') \in \mathcal{R}$. This can be proved by a case analysis on the possible actions of T.

- If $\alpha = \tau$ and for $x \in M_1'$, $y \in M_2'$,

$$\text{NoLeader}'\langle u_x, u_y, N_x\rangle \xrightarrow{election_{u_y,u_x}(\nu)} \text{InComp}'\langle u_x, u_y, \nu, N_x, N_x - \{u_y\}, N_x - \{u_y\}, \emptyset, u_x\rangle,$$

$$\text{InComp}'\langle u_y, f_y, \nu, N_y, S_y, R_y, A_y, max_y\rangle \overset{\overline{election_{u_y,u_x}}(\nu)}{\longrightarrow} \text{InComp}'\langle \ldots, S_y - \{u_x\}, \ldots\rangle$$

and

$$\begin{aligned} T \overset{\tau}{\longrightarrow} & (\prod_{m\in M_1'-\{u_x\}} \text{NoLeader}'\langle u_m, f_m, N_m\rangle \\ & \mid \text{InComp}'\langle u_x, u_y, \nu, N_x, N_x - \{u_y\}, N_x - \{u_y\}, \emptyset, u_x\rangle \\ & \mid \text{InComp}'\langle u_y, f_y, \nu, N_y, S_y - \{s_x\}, R_y, A_y, max_y\rangle \\ & \mid \prod_{m\in M_2'-\{u_y\}} \text{InComp}'\langle u_m, f_m, \nu, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m\in M_3'} \text{LeaderMode}'\langle u_m, f_m, \nu, N_m\rangle \\ & \mid \prod_{m\in M_4'} \text{ElectedMode}\langle u_m, f_m, \nu, N_m, S_m, l_m\rangle)\backslash F \end{aligned}$$

By the definition of similar processes it must be that $u_y \in M_2$ and either $u_x \in M_1$ or $u_x \in M_2 \cup M_3 \cup M_4$ and $s_x \neq \nu$. Suppose that $u_x \in M_3$ (the remaining cases are similar). Then we have:

$$\text{InComp}\langle u_y, f_y, \nu, N_y, S_y, R_y, A_y, max_y\rangle \overset{\overline{election_{u_y,u_x}}(\nu)}{\longrightarrow} \text{InComp}\langle \ldots, S_y - \{u_x\}, \ldots\rangle$$

and, since $\nu = max(M_2 \cup M_3 \cup M_4)$, $s_x < \nu$ and

$$\text{LeaderMode}\langle u_x, f_x, s_x, N_x\rangle \overset{election_{u_y,u_x}(\nu)}{\longrightarrow} \text{InComp}\langle u_x, u_y, \nu, N_x, N_x - \{u_y\}, N_x - \{u_y\}, \emptyset, u_x\rangle\ .$$

Consequently,

$$\begin{aligned} P \overset{\tau}{\longrightarrow} & (\prod_{m\in M_1} \text{NoLeader}\langle u_m, N_m\rangle \\ & \mid \prod_{m\in M_2-\{u_y\}} \text{InComp}'\langle u_m, f_m, s_m, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \text{InComp}\langle u_y, f_y, \nu, N_y, S_y - \{u_x\}, R_y, A_y, max_y\rangle \\ & \mid \text{InComp}\langle u_x, u_y, \nu, N_x, N_x - \{u_y\}, N_x - \{u_y\}, \emptyset, u_y\rangle \\ & \mid \prod_{m\in M_3-\{u_x\}} \text{LeaderMode}\langle u_m, f_m, s_m, N_m\rangle \\ & \mid \prod_{m\in M_4} \text{ElectedMode}\langle u_m, f_m, s_m, N_m, S_m, l_m\rangle)\backslash F \end{aligned}$$

and T and P are similar to each other.

- If $\alpha = \tau$ and for $x \in M_3'$, $y \in M_4'$,

$$\text{LeaderMode}'\langle u_x, u_y, \nu, N_x\rangle \overset{leader_{u_y,u_x}(\nu,l_y)}{\longrightarrow} \text{ElectedMode}\langle u_x, u_y, \nu, N_x, N_x - \{u_y\}, l_y\rangle,$$

$$\text{ElectedMode}\langle u_y, f_y, \nu, N_y, S_y, l_y\rangle \overset{\overline{leader_{u_y,u_x}}(\nu,l_y)}{\longrightarrow} \text{ElectedMode}\langle \ldots, S_y - \{u_x\}, \ldots\rangle$$

and

$$\begin{aligned} T \quad \xrightarrow{\tau} \quad & (\prod_{m\in M_1'} \text{NoLeader}'\langle u_m, f_m, N_m\rangle \\ & \mid \prod_{m\in M_2'} \text{InComp}'\langle u_m, f_m, \nu, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m\in M_3'-\{u_x\}} \text{LeaderMode}'\langle u_m, f_m, \nu, N_m\rangle \\ & \mid \text{ElectedMode}\langle u_x, u_y, \nu, N_x, N_x-\{u_y\}, l_y\rangle \\ & \mid \text{ElectedMode}\langle u_y, f_y, \nu, N_y, S_y-\{u_x\}, l_y\rangle \\ & \mid \prod_{m\in M_4'-\{u_y\}} \text{ElectedMode}\langle u_m, f_m, \nu, N_m, S_m, l_m\rangle)\backslash F \end{aligned}$$

By the definition of similar processes it must be that $u_y \in M_4$ and $u_x \in M_3$ with $s_x = s_y = \nu$, and we have:

$$\text{LeaderMode}\langle u_x, u_y, \nu, N_x\rangle \xrightarrow{leader_{u_y,u_x}(\nu, max_y)} \text{ElectedMode}\langle u_x, u_y, \nu, N_x, N_x-\{u_y\}, max_y\rangle.$$

Consequently,

$$\begin{aligned} P \quad \xrightarrow{\tau} \quad & (\prod_{m\in M_1} \text{NoLeader}\langle u_m, N_m\rangle \\ & \mid \prod_{m\in M_2} \text{InComp}\langle u_m, f_m, s_m, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m\in M_3-\{u_x\}} \text{LeaderMode}\langle u_m, f_m, s_m, N_m\rangle \\ & \mid \text{ElectedMode}\langle u_x, u_y, \nu, N_x, N_x-\{u_y\}, l_y\rangle \\ & \mid \text{ElectedMode}\langle u_y, f_y, \nu, N_y, S_y-\{u_x\}, l_y\rangle) \\ & \mid \prod_{m\in M_4-\{u_x\}} \text{ElectedMode}\langle u_m, f_m, s_m, N_m, S_m, l_m\rangle\backslash F \end{aligned}$$

and T and P are similar to each other.

- If $\alpha = \tau$ and the action has arisen from a communication along a channel of type $ack0$ or $ack1$ then the proof follows similarly to the previous two cases.

- If $\alpha = \overline{leader}(max)$ then there exists $x \in M_2'$ such that

$$\text{InComp}'\langle u_x, u_x, u_x, N_x, \emptyset, \emptyset, \emptyset, max\rangle \xrightarrow{\overline{leader}(max)} \text{ElectedMode}\langle u_x, u_x, u_x, N_x, N_x, max\rangle$$

and

$$\begin{aligned} T \ \xrightarrow{\tau}\ & (\prod_{m\in M_1'} \mathrm{NoLeader}'\langle u_m, f_m, N_m\rangle \\ & \mid \prod_{m\in M_2'-\{u_x\}} \mathrm{InComp}'\langle u_m, f_m, \nu, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m\in M_3'} \mathrm{LeaderMode}'\langle u_m, f_m, \nu, N_m\rangle \\ & \mid \prod_{m\in M_4'} \mathrm{ElectedMode}\langle u_m, f_m, \nu, N_m, S_m, l_m\rangle \\ & \mid \mathrm{ElectedMode}\langle u_x, u_x, u_x, N_x, N_x, max\rangle)\backslash F \end{aligned}$$

By the definition of similar processes it must be that $x = \nu \in M_2$ and

$$\mathrm{InComp}\langle \nu, \nu, \nu, N_x, \emptyset, \emptyset, \emptyset, max\rangle \xrightarrow{\overline{leader}(max)} \mathrm{ElectedMode}\langle \nu, \nu, \nu, N_x, N_x, max\rangle$$

and

$$\begin{aligned} P \ \xrightarrow{\tau}\ & (\prod_{m\in M_1} \mathrm{NoLeader}\langle u_m, N_m\rangle \\ & \mid \prod_{m\in M_2-\{u_x\}} \mathrm{InComp}'\langle u_m, f_m, \nu, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m\in M_3} \mathrm{LeaderMode}\langle u_m, f_m, \nu, N_m\rangle \\ & \mid \prod_{m\in M_4} \mathrm{ElectedMode}\langle u_m, f_m, \nu, N_m, S_m, l_m\rangle \\ & \mid \mathrm{ElectedMode}\langle \nu, \nu, \nu, N_x, N_x, max\rangle)\backslash F \end{aligned}$$

and T and P are similar to each other.

This completes the proof. $\square$

Proof of Lemma 4.6

Given a computation $P_0 \stackrel{w}{\Longrightarrow} P$, where P is as in the Definition A.6 above, we say that $T_0 \in \mathcal{T}$, is *compatible* with the computation, if

$$T_0 \stackrel{\mathrm{def}}{=} (\prod_{i\in K-\{\nu\}} \mathrm{NoLeader}'\langle i, p_i, N_i, l_i\rangle \mid \mathrm{InComp}'\langle \nu, \nu, \nu, N_\nu, N_\nu, N_\nu, \emptyset, \nu\rangle)\backslash F\ ,$$

where $\nu = max(M_2 \cup M_3 \cup M_4)$ and, for all $i \in M_2 \cup M_3 \cup M_4$ such that $s_i = \nu$, $p_i = f_i$. Note that by Lemma A.5, $\{i, f_i | i \in K, s_i = \nu\}$ is a spanning tree of the network, hence, there exists a compatible T_0 process for every derivative P of P_0.

We will prove the result by induction on the length, n, of the transition $P_0 \stackrel{w}{\Longrightarrow} P$.

The base case $n = 0$ is trivially true for any $T_0 \in \mathcal{T}$. Suppose that the result holds for $n = k - 1$ and consider $P_0 \stackrel{w}{\Longrightarrow} P' \xrightarrow{\alpha} P$ a transition of length k. Let T_0 be compatible with the computation. Then, T_0 is also compatible with the computation $P_0 \stackrel{w}{\Longrightarrow} P'$ and, by the induction hypothesis, $T_0 \stackrel{w}{\Longrightarrow} T'$ where P' and T' are similar. Now, consider the transition $P' \xrightarrow{\alpha} P$. The following cases exist:

- $\alpha = \tau$ and the internal action took place on a channel in F with object $s \neq \nu$. Then, we may see that for $T = T'$, $T' \overset{\epsilon}{\Longrightarrow} T'$ with P and T being similar.
- $\alpha = \tau$ and the internal action took place on a channel in F with source $s = \nu$. Then, using a case analysis similar to the one found in the proof of Lemma 4.5, we may find appropriate T such that $T' \xrightarrow{\tau} T$ and T, P similar.
- $\alpha = \overline{leader}(m)$. Then there must exist a process $\mathrm{InComp}\langle u, u, u, N, \emptyset, \emptyset, \emptyset, m\rangle$ in P'. Further, it must be that the process has received a message along channel $ack1_{v,u}(u, m_v)$ for all $v \in N$. In turn, this implies that all $v \in K$ received a message along channel $ack1_{w,v}(u, m_w)$ for all $w \in N_v$, and so on. Since the network is connected, this implies that all nodes except u have, at some point in the past, entered state $\mathrm{LeaderMode}\langle i, u, N_i\rangle$. Once in such a mode, a node can either maintain this state or evolve into a process of the form $\mathrm{ElectedMode}\langle i, u, N_i, S_i, l\rangle$. Thus,

$$\begin{aligned}
P' &\stackrel{\text{def}}{=} (\mathrm{InComp}\langle u, u, u, N, \emptyset, \emptyset, \emptyset, m\rangle \mid \prod_{m \in L_1} \mathrm{LeaderMode}\langle u_m, u, N_m\rangle \\
&\quad \mid \prod_{m \in L_2} \mathrm{ElectedMode}\langle u_m, u, N_m, S_m, l_m\rangle)\backslash F \\
&\xrightarrow{\alpha} P = (\mathrm{ElectedMode}\langle u, u, N, N, m\rangle \mid \prod_{m \in L_1} \mathrm{LeaderMode}\langle u_m, u, N_m\rangle \\
&\quad \mid \prod_{m \in L_2} \mathrm{ElectedMode}\langle u_m, u, N_m, S_m, l_m\rangle)\backslash F
\end{aligned}$$

and consequently,

$$\begin{aligned}
T' &\stackrel{\text{def}}{=} (\mathrm{InComp}'\langle u, u, u, N, \emptyset, \emptyset, \emptyset, m\rangle \mid \prod_{m \in L_1} \mathrm{LeaderMode}'\langle u_m, u, N_m\rangle \\
&\quad \mid \prod_{m \in L_2} \mathrm{ElectedMode}\langle u_m, u, N_m, S_m, l_m\rangle)\backslash F \\
&\xrightarrow{\alpha} T = (\mathrm{ElectedMode}\langle u, u, N, N, m\rangle \mid \prod_{m \in L_1} \mathrm{LeaderMode}'\langle u_m, u, N_m\rangle \\
&\quad \mid \prod_{m \in L_2} \mathrm{ElectedMode}\langle u_m, u, N_m, S_m, l_m\rangle)\backslash F
\end{aligned}$$

and the result follows. □