Formal Methods for Specifying and Verifying Distributed Algorithms Process Algebra vs I/O Automata *

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

Abstract

This paper embarks on a comparative study between two well-developed theories for formally specifying and verifying concurrent systems. 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 contrast the approaches and draw initial conclusions with respect to their relative capabilities and strengths.

1 Introduction

Modern distributed systems are intrinsically difficult to develop and reason about. The need for formally verifying the correctness of distributed systems and algorithms has long been recognized 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. One such model is that of Process Algebra, PA [23, 4]. PAs constitute a formal framework with well-defined semantics which allows the compositional modelling and analysis of concurrent systems. They are equipped with precise semantics, thus providing a solid basis for understanding system behavior and reasoning about their correctness. The first PAs were proposed in the 1980s [23, 13, 3]. Since then, they have been the subject of extensive research and they have been extended in various directions. The result is the creation of PAs capable of describing and analyzing systems that include characteristics like time, constant or mobile topologies, probabilistic behavior, multi-casting and locations (e.g. [12, 25, 14, 5]) as well as tools for automatically reasoning about system behavior. They have been used in the literature for reasoning about a variety of algorithms including [1, 28, 30].

The model of Input/Output Automata of Lynch and Tuttle [21] is another popular formal framework for modelling distributed systems. Intensive research carried out during the last decade (e.g. [22, 15, 19, 20, 10]) has enriched this model with a number of mathematical tools and methodologies (e.g. forward simulations, invariant-checking, etc) which allow carrying out correctness proofs and precise analysis of complex static and dynamic distributed algorithms. IOA [8] is a modelling and programming

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

language based on I/O automata. The Theory of Distributed Systems group at MIT has designed and partially implemented an IOA toolkit to support the development and analysis of IOA programs. In the last few years, various extension of the I/O automata formalism have been developed to deal with different types of systems (e.g., Timed I/O Automata [15], Probabilistic I/O Automata [19], Hybrid I/O Automata [20], and Dynamic I/O Automata [2]) and corresponding verification methodologies have been proposed. Finally, it is worth noting that recently, the formalism has been employed for the specification and verification of ad-hoc network algorithms (e.g., [6, 29]).

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, [7] questions the suitability of process algebras for reasoning about certain classes of distributed algorithms. In our opinion, the time is ripe for such evaluations, the outcome of which will give significant feedback for directing future research efforts. This work aims to embark on such a comparative study and assess the strengths and weaknesses of the two formalisms of process algebra and I/O automata for reasoning about distributed systems based on hands-on experience. Based on this study we would like to eventually be able to answer questions such as: Does one formalism perform "better" than the other (for some type of problems)? Is there a characterization of problems that are more amenable to verification by one of the two formalisms? Can ideas originating in one of the formalisms be carried over to the other to improve the verification process?

We begin to consider these questions by verifying a typical distributed algorithm in both of the formalisms. In particular we specify and verify a leader-election algorithm [36] with static membership and fault-free components by applying each of the frameworks for its correctness. We observe the limits/capabilities of each of the frameworks for modeling the algorithm. We apply the associated proof techniques for proving the algorithm's correctness and we evaluate them. We compare and present the two experiences. To the best of our knowledge this is the *first* such hands-on evaluation and comparison of the two formalisms.

Document Structure. In the following section we present the two formalisms, an introductory comparison between them as well as the algorithm to be verified. Sections 3 and 4 contain the specifications and verifications of the algorithms in PA and I/O automata, respectively. In Section 5 we contrast the two verification experiences and in Section 6 we conclude the paper with some final remarks and a discussion of future work. Missing proofs and some useful background material can be found in the Appendix.

2 Prelimilaries

In this section we present the two formalisms that will be used for the specification and verification of our algorithm, an initial comparison between the two, as well as the description of the algorithm we will consider.

2.1 The Calculus

The calculus employed in this work is the CCS_v process calculus, which is a value-passing CCS calculus [23, 34] including a kind of conditional agents.

2.1.1 The syntax

We begin by describing the basic entities of the calculus. We assume a set of *constants*, ranged over by u, v, including the positive integers and the boolean values and a set of functions, ranged over by f, operating on these constants. Moreover, we assume a set of *variables* $\mathcal{V}$, ranged over by x, y. Then, the set of *terms* of CCS_v, ranged over by e, is given by (1) the set of constants, (2) the set of variables, and (3) function applications of the form $f(e_1, \ldots e_n)$, where the e_i are terms. We say that a term is *closed* if it contains no variables. The evaluation relation $\twoheadrightarrow$ for closed terms is defined in the expected manner.

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 o*utput position*, denoted by $\overline{a}$. This gives rise to the set of *actions*, *Act*, of the calculus, ranged over by α, β, containing

- the set of *input actions* which have the form $a(\tilde{v})$ representing the input along channel a of a tuple $\tilde{v}$, where we write $\tilde{r}$ for a tuple of syntactic entities $r_1, \ldots, r_n$,
- 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
- 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. Further, given a non-internal action α we denote by $\ell(\alpha)$ the channel of action α.

Finally, we assume a set of process constants $\mathcal{C}$, denoted by C. The syntax of CCS_v is given by the following BNF definition

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

Process $\mathbf{0}$ represents the inactive process. Process $\alpha.P$ describes the prefix process which first engages in action α and then behaves as the continuation process P. Process $P_1 + P_2$ represents the nondeterministic choice between processes P_1 and P_2. The process $P \| Q$ describes the concurrent 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 $e_i \twoheadrightarrow \mathsf{true}$.

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. This gives rise to the free and bound channels of a process.

Finally, process constants provide a mechanism for including recursion in the process calculus. 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.

2.1.2 The 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 process P can perform action α and evolve into process P'. The rules themselves have the form

$$\frac{T_1, \ldots, T_n}{T} \quad \phi$$

(In)	$a(\tilde{x}).P \xrightarrow{a(\tilde{v})} P\{\tilde{v}/\tilde{x}\}$	(Out)	$\overline{a}(\tilde{v}).P \xrightarrow{\overline{a}(\tilde{v})} P$
(Sum)	$\dfrac{P_i \xrightarrow{\alpha} P_i'}{P_1 + P_2 \xrightarrow{\alpha} P_i'} \quad i \in \{1,2\}$	(Par1)	$\dfrac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \| P_2 \xrightarrow{\alpha} P_1' \| P_2}$
(Par2)	$\dfrac{P_2 \xrightarrow{\alpha} P_2'}{P_1 \| P_2 \xrightarrow{\alpha} P_1 \| P_2'}$	(Par3)	$\dfrac{P_1 \xrightarrow{a(\tilde{v})} P_1',\ P_2 \xrightarrow{\overline{a}(\tilde{v})} P_2'}{P_1 \| P_2 \xrightarrow{\tau} P_1' \| P_2'}$
(Res)	$\dfrac{P \xrightarrow{\alpha} P'}{P \backslash F \xrightarrow{\alpha} P' \backslash F} \quad l(a) \notin F$	(Const)	$\dfrac{P\{\tilde{v}/\tilde{x}\} \xrightarrow{\alpha} P'}{C\langle\tilde{v}\rangle \xrightarrow{\alpha} P} \quad C\langle\tilde{x}\rangle \stackrel{\text{def}}{=} P$
(Cond)	$\dfrac{P_i \xrightarrow{\alpha} P_i'}{\mathsf{cond}\ (e_1 \rhd P_1, \ldots, e_n \rhd P_n) \xrightarrow{\alpha} P_i}$	$e_i \twoheadrightarrow true,\ \forall j < i, e_j \twoheadrightarrow false$	

Table 1: **The operational semantics**

which is interpreted as follows: if transitions $T_1, \ldots, T_n$, can be derived, and condition ϕ holds, then we may conclude transition T. The semantics of the CCS$_v$ operators are given in Table 1.

We discuss some of the rules below:

- (IN). This axiom employs the notion of *substitution*, a partial function from variables to values. We write $\{\tilde{v}/\tilde{x}\}$ for the substitution that maps variables $\tilde{x}$ to values $\tilde{v}$. Thus, the input-prefixed process can receive a tuple of values $\tilde{v}$ along channel a, and then continue as process P, with the occurrences of the variables $\tilde{x}$ in P substituted by values $\tilde{v}$. For example: $a(x,y).\overline{b}(x).\mathbf{0} \xrightarrow{a(2,5)} \overline{b}(2).\mathbf{0}$.
- (Cond). This axiom formalizes the behavior of the conditional operator. An example of the rule follows: $\mathsf{cond}\ (2 = 3 \rhd \overline{b}(3).\mathbf{0}, \mathsf{true} \rhd \overline{c}(4).\mathbf{0}) \xrightarrow{\overline{c}(4)} \mathbf{0}$.
- (Par1). This axiom (and its symmetric version (Par2)) expresses that a component in a parallel composition of processes may execute actions independently: $\overline{a}(3).\mathbf{0} \,\|\, a(v).\overline{b}(c).\mathbf{0} \xrightarrow{\overline{a}(3)} a(v).\overline{b}(c).\mathbf{0}$.
- (Par3). This axiom expresses that two parallel processes executing complementary actions may synchronize with each other producing the internal action τ: $\overline{a}(3).\mathbf{0} \,\|\, a(v).\overline{b}(v).\mathbf{0} \xrightarrow{\tau} \overline{b}(3).\mathbf{0}$.
- (Const). This axiom stipulates that, given a process constant and its associated definition $C\langle\tilde{x}\rangle \stackrel{\text{def}}{=} P$, its instantiation $C\langle\tilde{v}\rangle$ behaves as process P with variables $\tilde{x}$ substituted by $\tilde{v}$. For example, if $C\langle x,y\rangle \stackrel{\text{def}}{=} \mathsf{cond}\ (x = y \rhd \overline{b}(x).\mathbf{0}, \mathsf{true} \rhd \overline{c}(y).\mathbf{0})$, then $C(2,2) \xrightarrow{\overline{b}(2)} \mathbf{0}$.

We recall some useful definitions. We say that Q is a *derivative* of P, if there are $\alpha_1, \ldots, \alpha_n \in Act$, $n \geq 0$, such that $P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$. Moreover, given $\alpha \in Act$ we write $\Longrightarrow$ for the reflexive and transitive closure of τ, $\stackrel{\alpha}{\Longrightarrow}$ for the composition $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$, and $\stackrel{\hat{\alpha}}{\Longrightarrow}$ for $\Longrightarrow$ if $\alpha = \tau$ and $\stackrel{\alpha}{\Longrightarrow}$ otherwise.

We conclude this section by presenting a notion of process equivalence in the calculus. Observational equivalence is 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* [23, 26]. 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.

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 $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $P' \approx Q'$.

Note that bisimilarity abstracts away from internal computation by focusing on *weak* transitions, that is, transitions of the form $\stackrel{\hat{a}}{\Longrightarrow}$ and requires that bisimilar systems can match each other's *observable* behavior. Bisimilarity implies trace equivalence but the opposite does not hold.

Bisimulation relations have been studied widely in the literature. They have been used to establish that a system satisfies its implementation by describing the two as process-calculus processes and discovering a bisimulation that relates them. Their theory has been developed mainly into two directions. On 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. The results reviewed in the next section belong to the latter type.

2.1.3 Confluence

In this subsection we recall some useful results regarding the theory of confluence that are employed in this study. For further information we refer the reader to [24, 34, 33, 11, 31].

In [23, 24], 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 2.2 P is *determinate* if, for every derivative Q of P and for all $\alpha \in Act$, whenever $Q \xrightarrow{\alpha} Q'$ and $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q''$ 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 2.3 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 $Q \stackrel{\beta}{\Longrightarrow} Q_2$ then, there are Q_1' and Q_2' such that $Q_2 \stackrel{\hat{\alpha}}{\Longrightarrow} Q_2'$, $Q_1 \stackrel{\hat{\beta}}{\Longrightarrow} Q_1'$ and $Q_1' \approx Q_2'$.

Its essence, to quote [24], is that "of any two possible actions, the occurrence of one will never preclude the other". As shown in [24, 23], 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, [11, 34, 16, 27, 31, 32, 30]).

In this work, we will employ the following additional notion and result for aiding the verification process [31].

Definition 2.4 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 $Q \stackrel{\overline{a}(\tilde{y})}{\Longrightarrow} Q''$, then $\tilde{x} = \tilde{y}$ and $Q' \approx Q''$.

Theorem 2.5 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.

2.2 I/O Automata

In this section we present the Input/Output Automata formalism of Lynch and Tuttle [21, 18].

2.2.1 Description

Actions, Signatures, Executions and Traces. An Input/Output Automaton (or I/O Automaton or 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 required to be enabled in all states. Thus automata are said to be *input-enabled*. In other words, the automaton is unable to block its input (of course it can ignore it) a fundamental assumption of the I/O Automata framework.

A *signature* S of an I/O Automaton A denoted by $sig(A)$, is a triple of disjoint set of the input actions $in(S)$, the output actions $out(S)$, and the internal actions $int(S)$ of that automaton. The set of all actions is denoted by $act(S)$. The *external signature* consists only of the sets of input and output actions. We denote by $local(S) = out(S) \cup int(S)$ the set of *locally-controlled actions*, those actions under the local control of any automaton having S as its action signature. Given an automaton A with signature S we abuse notation and denote $in(S)$ by $in(A)$, etc.

Formally an I/O Automaton A consists of the following:

- an action signature $sig(A)$,
- a set $states(A)$ of states,
- a nonempty set $start(A) \subseteq states(A)$ of start states,
- a transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for every state s and input action π there is a transition or *step* (s, π, s'), in $trans(A)$, and
- an equivalence relation $tasks(A)$ partitioning the set $local(A)$ into at most a countable number of equivalence classes.

The equivalence relation $tasks(A)$ is tightly coupled with the notion of fairness, presented later. It is used to identify the primitive components of the system being modeled by the automaton: each class is thought of as the set of actions under the local control of one system component.

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 $(s_i, \pi_{i+1}, s_{i+1}) \in trans(A)$, for every $i \geq 0$. An *execution* is an execution fragment that starts with a start state (i.e. $s_0 \in start(A)$). We denote by $exec(A)$ the set of all executions of A. We say that a state is *reachable* if it is the final state of a finite execution 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. We denote the set of all traces of A by $traces(A)$. Finally, we say that an automaton *implements* another automaton, if any trace of the former is a trace of the latter.

Composition of I/O Automata. I/O Automata can be composed to create more complex I/O Automata. The (parallel) composition operator essentially allows an output action of one automaton to be identified with the input actions in other automata; this operator *respects* the trace semantics. In more detail, during the composition of several I/O Automata to a more complex I/O Automaton we identify the same-named actions of the different automata. Then if an automaton has some output

action π and one or more automata of the composition have as input that same action π, the first automaton will generate the output action π and instantaneously transmit action to all automata that have the input action π, while the rest of the automata in the composition do nothing.

Certain restrictions are imposed in order for the composition of a set of automata to be possible. In particular, the automata need to be *compatible*. Automata are said to be compatible if their signatures are compatible. Hence, the following definition: A countable collection $\{S_i\}_{i \in I}$ of signatures is said to be *compatible* if for all $i, j \in I,\ i \neq j$, we have

1. $out(S_i) \cap out(S_j) = \emptyset$,
2. $int(S_i) \cap acts(S_j) = \emptyset$, and
3. No action is contained in infinitely many sets $acts(S_i)$.

This leads to the definition of composition of signatures: The *composition* $S = \prod_{i \in I} S_i$ of a countable collection of compatible signatures $\{S_i\}_{i \in I}$ is defined to be the signature

1. $in(S) = \bigcup_{i \in I} in(S_i) - \bigcup_{i \in I} out(S_i)$,
2. $out(S) = \bigcup_{i \in I} out(S_i)$, and
3. $int(S) = \bigcup_{i \in I} int(S_i)$.

From this we obtain the formal definition of automata composition. The *composition* $A = \prod_{i \in I} A_i$ of a countable collection of compatible automata $\{A_i\}_{i \in I}$ is the automaton defined as follows:

1. $sig(A) = \prod_{i \in I} sig(A_i)$,
2. $states(A) = \prod_{i \in I} states(A_i)$,
3. $start(A) = \prod_{i \in I} start(A_i)$,
4. $trans(A)$ is the set of triples $(\vec{s_1}, \pi, \vec{s_2})$ such that, for all $i \in I$, if $\pi \in acts(A_i)$ then $(\vec{s_1}[i], \pi, \vec{s_2}[i]) \in steps(A_i)$, otherwise $\vec{s_1}[i] = \vec{s_2}[i]$ ($\vec{s}[i]$ denotes the the ith component of the state vector $\vec{s}$), and
5. $tasks(A) = \bigcup_{i \in I} tasks(A_i)$.

It has been shown [18] that an execution (or trace) of a composition projects to yield executions (or traces) or the component automata. Inversely, under certain conditions, executions (or traces) of component automata can be pasted together to for an execution (or trace) if the composition.

Fairness. Since I/O Automata are input-enabled, the specification cannot prevent the occurrence of an infinite sequence of input actions, which could prevent the automaton from performing locally-controlled actions. This could result to executions that are not *fair*, and by fair we mean that the automaton has the property that infinitely often is presented with the opportunity to perform one of its local actions. Therefore, being fair to each system component means being fair to each equivalence class of locally-controlled actions. More formally, a *fair execution* of an Automaton A is an execution α of A such that the following conditions hold for each class C of $tasks(A)$:

1. If α is finite, then no action of C is enabled in the final state of α.
2. if α is infinite, then α either contains infinitely many events from C, or infinitely many occurrences of states in which no action of C is enabled.

In other words, a fair execution gives fair turns to each class C of $tasks(A)$, and therefore to each component for the system being modeled. In the next section we will relate fair executions with what we will call *liveness properties* of an automaton specification.

It has been shown [18] that every finite execution (or trace) can be extended to a fair execution (or trace).

2.2.2 Proof Methods

I/O Automata can be used not only to formally specify distributed and concurrent systems and algorithms, but also to formulate and prove precise claims about what systems and algorithms do.

Safety and Liveness Properties. Given a problem specification, that is a set of allowable behaviors, we say that an automaton *solves* the specification if each of the automaton's behaviors is contained in this set. In other words, the automaton solves the problem in the sense that every behavior it exhibits is a behavior allowed by the problem specification. Recall that automaton behaviors are characterized by the execution traces. Hence, within the I/O framework, the notion of proving the correctness of an automaton (that it solves the problem) is usually deduced in 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. Note that an infinite execution satisfies a safety property if and only if every finite prefix of it does so. A liveness property specifies events that must *eventually* be performed. In particular, it is required that something "good" eventually happens, which in return means that no matter what has happened up to this point, there is still the possibility that something good will happen. Clearly this is a property that can only be satisfied by infinite execution, and more specifically by fair executions. Therefore, in the context of liveness properties, only fair executions of an automaton are considered.

Taking safety and liveness properties together one can prove claims such as "Eventually the system will exhibit some required behavior".

Invariant Assertions. A fundamental technique for reasoning about the behavior of an automaton is by associating the automaton with *invariant assertions*. 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. More generally, invariants are used to prove other invariants which in turn are used when carrying out subsequent inductive proofs. Several invariants are usually combined in proving (mainly) safety properties of a given automaton.

Modular Decomposition. 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. This is very useful, especially when dealing with complex composed automata. 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. It is sometimes the case that reasoning about the behavior of only certain automata of the composition is enough to reason about the behavior of the composed automaton.

Hierarchical Proofs. Another common technique for reasoning about the behavior of automata is by *hierarchical decomposition*. A hierarchy represents a series of descriptions of a system or algorithm

at different levels of abstraction. We begin at the highest level of abstraction where we have the specification of an automaton, and through successive refinements we continue to lower levels of abstraction by introducing more of the detail that exists in the final system or algorithm. Because of the extra detail, lower levels of abstraction are usually harder to understand than higher levels. In order now to prove properties of the lower level we try to relate these lower level automata to the higher level automata.

In order to establish the correspondence between two different automata at different levels of abstraction we use simulation techniques. A *simulation* between two automata A and B (with the same external signature) involves establishing a correspondence (mapping) between A and B. The existence of a simulation between A and B is used to show that any behavior that can be exhibit by A, can also be exhibit by B. This means that if B solves a particular problem, so does A. In particular we say that automaton A *implements or simulates* automaton B if any trace of A is a traces of B. There are several simulation types [22].

2.3 Pre-Case-Study Comparison

One may observe that work in these frameworks 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 and developing their semantic theories, providing solid understanding to system features 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 [35] where the semantic relationship between the two 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.

Despite this semantic closeness between the two formalisms, from a practical point of view, they appear to have some significant differences. To begin with, at the specification level, 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. A system is modeled as a process which itself is a composition of subprocesses representing the system's constituents components. Action granularity is very fine: actions can be input on channels, output on channels and internal actions. As computation proceeds the possible behaviors a process may engage in are explicitly encoded in the process's description.

On the other hand, I/O automata feature a more "relaxed" type of language, quite close to imperative programming. It additionally enables a limited (in comparison to PAs) set of operators to be applied at the automaton level: renaming and parallel composition which incorporates a notion of action hiding. 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 subcomponents. 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. In contrast to PAs, 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 results in a less fine granularity of actions in comparison to PA's. Flow of execution is determined by the state of an automaton: any enabled action, that is, any action whose precondition is satisfied, may take place.

Moving on to the semantics of the two frameworks, some essential differences can be observed. While process algebras are typically given bisimulation or failure equivalence semantics, I/O automata are given trace semantics. It turns out that to provide compositional theories for typical PAs, it is necessary to consider the branching structure of processes as implemented, for example, by bisimulation. On the other hand, trace semantics is compositional for I/O automata. As it was shown in [35], this fact is a consequence of the input-enabledness of input actions and the non-blocking properties of the output actions. A further point to note is that the specific semantics enables reasoning about fairness within I/O automata models. Finally we point out that PAs have a rich algebraic structure and they are associated with axioms systems which prescribe the relations between the various constructs which can be used for reasoning algebraically/compositionally about system behavior.

Concluding with a comparison of verification methodologies, we observe that while PAs revolve around the establishment of equivalences between processes and checking a system against logical properties expressed in temporal logics, I/O automata mainly favor assertional proof techniques for reasoning about trace properties, including the establishment of safety and liveness properties, simulation relations and invariant assertions.

2.4 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 [36]. 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 broadcast 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.

In greater detail, 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.

3 Specification and Verification in PA

3.1 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 $F = \{election_{i,j}, ack0_{i,j}, ack1_{i,j}, leader_{i,j} \mid i, j \in K\}$ where $x_{i,j}$ refers to the channel from node i to node j of type x. Furthermore, channels $election_{i,j}$ are used for sending *election* messages, channels $ack0_{i,j}$ for acknowledging an election message but notifying the sender that the receiver is already in computation, channels $ack1_{i,j}$ for acknowledging an election message and notifying the sender that the receiver is accepting it as its father and channels $leader_{i,j}$ for sending *leader* messages. The system is described as the following parallel composition of its constituent nodes:

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

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 *election* 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 Figure 1.

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

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

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

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

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

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

Figure 1: The node process

3.2 Correctness Proof

The correctness criterion of our algorithm is expressed as the following bisimulation equivalence between the system and its specification.

Theorem 3.1 $P_0 \approx \overline{\text{leader}}(\mathbf{max}).0$ *where* $\mathbf{max} = max\{u_i |\ i \in K\}$.

The proof is established in two phases. In the first phase we consider a simplification of P_0 where a single initiator begins computation and where the spanning tree on which the algorithm operates is pre-determined. 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 $\overline{\text{leader}}(\mathbf{max}).0$. 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.

We begin the proof with a useful lemma.

Lemma 3.2 *Let P be an arbitrary derivative of P_0:*

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

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 $P_0 \Longrightarrow P' \stackrel{\tau}{\longrightarrow} P$ where the claim holds for P'. Let us write

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

$mx = max(M'_2, M'_3, M'_4)$ 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 $P' \stackrel{\alpha}{\longrightarrow} P$. We consider the two most interesting cases:

- $\alpha = \tau$, where
$$NoLeader\langle u_k, N_k\rangle \stackrel{\tau}{\longrightarrow} \text{InComp}\langle u_k, u_k, u_k, N_k, N_k, N_k, \emptyset, u_k\rangle$$
and $u_k > mx$. Then $M = \{u_k\}$ and the claim follows.

- $\alpha = \tau$ and for some $x \in M'_2$, $y \in M'_1 \cup M'_3 \cup M'_4$,
$$\text{InComp}\langle u_x, f_x, mx, N_x, S_x, R_x, A_x, max_x\rangle \stackrel{\overline{election_{u_x,u_y}}(mx)}{\longrightarrow} \text{InComp}\langle \ldots, S_x - \{u_y\}, \ldots\rangle$$
and
$$X \stackrel{election_{u_x,u_y}(\mathbf{max})}{\longrightarrow} \text{InComp}\langle u_y, u_y, mx, N_y, N_y - \{u_x\}, N_y - \{u_x\}, \emptyset, u_y\rangle,$$
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.

□

Returning to our proof, the restricted type of systems employed in the first phase of the proof use the following processes:

$\text{NoLeader}'\langle i, f, N, l\rangle \stackrel{\text{def}}{=} election_{f,i}(s).\ \text{InComp}'\langle i, f, s, N, N - \{f\}, N - \{f\}, \emptyset, i\rangle$

$\text{InComp}'\langle i, f, s, N, S, R, A, max\rangle \stackrel{\text{def}}{=}$

$\ldots$

$+ \sum_{j \in N} election_{j,i}(s').\,\mathrm{InComp}'\langle i, f, s, N, S, R, A \cup \{j\}, max\rangle,$

$\ldots$

$\mathrm{LeaderMode}'\langle i, s, N\rangle \stackrel{\mathrm{def}}{=}$

$\sum_{j \in N} leader_{j,i}(s', max').\,\mathrm{ElectedMode}\langle i, s, N, N - \{j\}, max'\rangle$

$+ \sum_{j \in N} election_{j,i}(s').\,\mathrm{LeaderMode}'\langle i, s, N\rangle$

Thus, NoLeader$'$ is similar to NoLeader except that it may only be activated by a signal from a specified node, f. Similarly, InComp$'$ and LeaderMode$'$ are similar to InComp and LeaderMode, respectively, except that they behave as if the source node of any *election* message is the same as that possessed by the node.

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

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

where $\{(i, f_i) | i \in K - \{\nu\}\}$ is a spanning tree of the network rooted at node ν, for some $\nu \in K$. We show the following sequence of results:

Lemma 3.3 $T_0 \stackrel{\overline{\mathrm{leader}}(\mathbf{max})}{\Longrightarrow} \approx \mathbf{0}$.

Proof. 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

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

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

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

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

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

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

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

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

$$T^D = R^D \implies \ldots \implies R^0\,.$$

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

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

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

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

where

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

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

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

which completes the proof. □

Lemma 3.4 *T_0 is confluent.*

Proof. We may check that processes NoLeader$'$, InComp$'$, LeaderMode$'$ and ElectedMode, are confluent by construction and satisfy the remaining conditions of Theorem 2.5. Thus, the result follows. □

From these two results we have that:

Corollary 3.5 $T_0 \approx \overline{leader}(\mathbf{max}).\mathbf{0}$ *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. The precise definition is as follows:

Definition 3.6 *Let*

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

$$\begin{aligned} T \stackrel{\text{def}}{=} & \Big(\prod_{m\in M'_1} \text{NoLeader}'\langle u_m, f_m, N_m, l_m\rangle \mid \prod_{m\in M'_2} \text{InComp}'\langle u_m, f_m, \nu, N_m, S_m, R_m, A_m, max_m\rangle \\ & \mid \prod_{m\in M'_3} \text{LeaderMode}'\langle u_m, \nu, N_m\rangle \mid \prod_{m\in M'_4} \text{ElectedMode}\langle u_m, \nu, N_m, S_m, l_m\rangle\Big)\backslash F \end{aligned}$$

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.*

Lemma 3.7 $\mathcal{R} = \{\langle T, P\rangle | P \text{ and } T \text{ are similar}\}$ *is a strong simulation.*

Proof. Consider processes T and P as in Definition 3.6 with $(T, P) \in \mathcal{R}$ and suppose that $T \stackrel{\alpha}{\longrightarrow} T'$. We will show that $P \stackrel{\alpha}{\longrightarrow} P'$ 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 $x \in M'_1$, $y \in M'_2$,

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

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

and

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

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:

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

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

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

Consequently,

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

and T and P are similar to each other.

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

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

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

and

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

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,

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

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

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

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

$$\text{InComp}\langle \nu, \nu, \nu, N_x, \emptyset, \emptyset, \emptyset, max\rangle \stackrel{\overline{leader}(max)}{\longrightarrow} \text{ElectedMode}\langle \nu, \nu, \nu, N_x, N_x, max\rangle$$

and

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

and T and P are similar to each other.

This completes the proof. □

By Corollary 3.5 and Lemma 3.7 we have that $P_0 \stackrel{\overline{leader}(\mathbf{max})}{\Longrightarrow} \mathbf{0}$. Our final result establishes a correspondence between P_0 and agents $T_0 \in \mathcal{T}$.

Lemma 3.8 *If $P_0 \stackrel{w}{\Longrightarrow} P$ then there exists T_0 such that, $T_0 \stackrel{w}{\Longrightarrow} T$ and P and T are similar.*

Proof.

Given a computation $P_0 \stackrel{w}{\Longrightarrow} P$, where P is as in the Definition 3.6 above, we say that $T_0 \in \mathcal{T}$, is *compatible* with the computation, if

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

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 3.2, $\{i, f_i | i \in K, s_i = \nu\}$ 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 $P_0 \stackrel{w}{\Longrightarrow} P' \stackrel{\alpha}{\longrightarrow} P$ 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 $P' \stackrel{\alpha}{\longrightarrow} P$. 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 $T = T'$, $T' \stackrel{\epsilon}{\Longrightarrow} T'$ 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 3.7, we may find appropriate T such that $T' \stackrel{\tau}{\longrightarrow} T$ and T, P similar.
- $\alpha = \overline{leader}(m)$. Then there must exist a process $\text{InComp}\langle u, u, u, N, \emptyset, \emptyset, \emptyset, m\rangle$ 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 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 ElectedMode$\langle i, u, N_i, S_i, l\rangle$. Thus,

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

and consequently,

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

and the result follows.

□

We can now prove our main theorem. We have seen that $P_0 \stackrel{\overline{leader}(\mathbf{max})}{\Longrightarrow} \mathbf{0}$. Further, suppose that $P_0 \stackrel{\alpha}{\Longrightarrow}$ with $\alpha \neq \overline{leader}(\mathbf{max})$. Then, there exists T_0 such that $T_0 \stackrel{\alpha}{\Longrightarrow}$. However, this is in conflict with Corollary 3.5. 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.

4 Specification and Verification in IOA

4.1 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 Figure 2. The specification of the Channel automaton $C_{i,j}$ is the one typically used for non-lossy channels and is given in Figure 3.

4.2 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.

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

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

Precondition:

m first on $tosend_i[j]$

$j \in Nbrs_i$

Effect:

deque m from $tosend_i[j]$

internal $\mathsf{beginComputation}_i$

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 $\mathsf{setAcktoParent}_i$

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 $\mathsf{setLeader}_i$

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 2: The LENODE$_i$ automaton.

Signature:

Input:

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

Output:

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

States:

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

Transitions:

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

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

Figure 3: The Channel Automaton $C_{i,j}$

4.2.1 A Unique Spanning Tree is Built

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

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 $s.parent_i \neq \perp$.

Proof. 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. □

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 4.1 *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 $\pi = \mathsf{beginComputation}_j$.*

Proof. 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.

□

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.*

Proof. 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 4.1 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 4.1 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. □

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 $s'.src_i > s.src_i$.*

Proof. 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. □

Liveness Properties

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

Lemma 4.2 *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 $j \in I$, 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. □

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 4.3 *Algorithm* LE *eventually builds a unique spanning tree rooted at* i_{smax}.

Proof. If only one process executes $\mathsf{beginComputation}_i$ then $i = i_{smax}$ and by Lemma 4.2, 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 4.2, 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 4.2 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 4.2 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. □

4.2.2 A Unique Common Leader is Elected

We now state and prove 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 $s'.max_i > s.max_i$.*

Proof. 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 $m.srcid = src_i$. □

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 4.4 *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.*

Proof. 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. □

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

Proof. 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 4.3 is unique. Furthermore, by the preconditions it holds that $s.toBeActed_i = \emptyset$ and $s.sentAcktoParent_i = false$. By Lemma 4.4 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. □

Liveness Properties

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

Theorem 4.6 *Given a fair execution of* LE *there exists a state* s *where* $\forall\, i \in I$, $s.leader_i = i_{max}$.

Proof. Starting from an initial state s_0 the only possible action to occur is the $\mathsf{beginComputation}_i$ action. From Theorem 4.3 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 4.4, 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 4.5, $leader_{i_{smax}} = i_{max}$.

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. □

5 Post-Case-Study Comparison

Having presented the models and correctness proofs of the LE algorithm in the two formalisms, in this section we contrast the two approaches and draw conclusions regarding their applicability and relative

strengths.

Specification. Beginning with the specifications developed in the two frameworks, we note that they have many similarities as well as some 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 the 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. To begin with, as already noted, 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. 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 states NoLeader, InComp, LeaderMode, ElectedMode with the possibility, under certain circumstances, to flow from a LeaderMode to an InComp. The possible actions enabled from each state are specified in the state's definition. On the other hand, in the I/O automata model, the flow of execution has to be captured via an appropriate use of state variables and management of their values. 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. Another interesting point, is that the two formalisms differ in their adoption of channels: In CCS channels are a first-class entity and communication between processes is carried out by a handshake mechanism over their connecting channels. If one needs to employ a more involved type of a channel (e.g. buffer or lossy channel), then special processes need to be described for connecting the original sender and receiver. In contrast, in the I/O automata model, channels are modeled as automata which execute complementary actions with their source and destination. For simple types of channels, this machinery is standard and becomes almost invisible to the main body of an application but has as a consequence that in a proof one needs to assume the proper delivery of messages, assuming of course that channels are intended to be reliable. Finally, we note that the specifications were produced by a newcomer to both of these formalisms who reported the I/O automata model to be easier to produce and understand. This is mainly due to the programming style of I/O automata which does not place great demands on a newcomer to the formalism.

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. The process calculus proof is based on the use of bisimulation for establishing the equivalence between the system and its perceived intended behavior. This approach places the emphasis on the external behavior the system may produce and involved adding an advertisement of the election of a leader in the specification of the processes. On the other hand, the I/O approach concentrates on the internal state of the network's nodes and 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.

In general, we observe that the process-calculus philosophy and proofs methods are geared towards

establishing properties of the external behavior of a system. This can be problematic when the correctness criterion concerns the internal state of the system. Similarly, it is less obvious how easily the I/O automata model can prove a correctness criterion expressed as a complex sequence of external communications. Furthermore, there appears to be a distinction between the two formalisms' ability to deal with global and local properties. On the one hand, using process calculi one tends to manage a global view of the system, which can be convenient when dealing with global properties. On the other hand, in I/O automata, modular decomposition allows to decompose global properties of a composition of automata into local properties of the automata and thus simplify the reasoning. However, this approach assumes the decomposition of the global property into local statements which is not always straightforward.

The above, however, does not imply that the two formalisms are incapable of each performing the type of reasoning natural in the other. In particular, for the specific algorithm, the I/O automata argumentation could be adopted in the process-algebraic setting: the various safety and liveness properties could be rewritten and proved correct by induction on the length of the derivation. On the other hand, translating the process-calculus proof into the I/O automata framework would not be feasible in its entirety. Specifically, the fact that the notion of confluence has not been developed in the setting would preclude "copying" the first phase of the proof. Still the second part, consisting of the establishment of what is called in the I/O automata language a backward simulation [22], would be possible to establish. However, it is questionable, whether such translations would result in easier-to-produce or more comprehensible proofs. For the specific algorithm, it seems that it would not be natural to do so.

Moving on to the application of the two methodologies, one may argue that the proof methodologies of process calculi appear to be more technical. They offer a variety of results such as compositionality results for facilitating the verification process but it may take some ingenuity for choosing and adopting them. On the other hand, I/O automata proofs can be more intuitive, closer to the "way of thinking" of the algorithm, thus easier to apply. The challenge being to identify the appropriate safety and liveness properties (which for the specific algorithm were not difficult), the rest of the process is guided by checking for missing information towards reaching the intended goal and subsequently expressing it as additional lemmas and invariants. However, the descriptive language typically employed in I/O automata liveness proofs (which of course can be made formal by the use of advanced techniques) may 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 language and semantics (much in the same way as the proof of safety properties in I/O automata). 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 informal.

Finally, we point out that the process of carrying out the proof within the I/O automata framework has been a lot more straightforward than in the PA framework. This is probably not surprising since the I/O automata model has been especially developed for by-hand verification of distributed algorithms and is associated with a bulk of work containing exactly this type of proofs [17]. In comparison, there are relatively few similar exercises in the PA model, where more emphasis has been given on automatic model-checking for process verification. The specific approach taken in the proof presented, although certainly not the only viable solution, has also been used in [31, 30, 9].

6 Conclusions

The purpose of this work is the evaluation of two popular formalisms in the domain of reasoning about distributed systems. On embarking on this study a number of choices had to be made with regards to the case study and the tools to be used. Our choice of the algorithm was mostly influenced by the motivation to begin this line of work with a fairly simple algorithm which both formalisms were expected to handle well. (This is a fairly standard algorithm involving essentially the computation of a function based on point-to-point communication.) As expected no surprises were met, a fact that allowed us to focus on the juxtaposition of the two formalisms and the crystallization of some initial conclusions. The choice of the specific instances of the formalisms to be used for our study then followed straightforwardly. In the case of I/O automata the basic model was clearly the right decision, whereas in the case of process algebra a basic calculus that included value-passing provided the simplest formalism for the task.

In current work we are extending this study by considering the mobile version of the algorithm. This algorithm assumes high mobility of the network nodes and applies broadcasting communication between them. These facts place a challenge on the formalisms employed and especially, in the case of process calculi, raises questions whether these features ought to be simulated in the basic CCS_v calculus or whether an alternative option should be used. The correctness proofs also appear to be much harder in both formalisms and it is still uncertain how they will eventually compare with each other. In future work, we plan to extend our study to other problems arising in distributed systems and networks. Our choice of problems will be based upon our initial evaluation regarding the relative strengths of PAs and I/O Automata to better handle global vs local correctness criteria, respectively.

References

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

[2] C. P. Attie and N. A. Lynch. Dynamic Input/Output Automata: a formal model for dynamic systems. In *Proceedings of CONCUR'01*, LNCS 2154, pages 137–151, 2001.

[3] J. Baeten and W. P. Weijland. *Process Algebra*. Cambridge, 1990.

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

[5] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000.

[6] G. Chockler, N. A. Lynch, S. Mitra, and J. A. Tauber. Proving atomicity: An assertional approach. In *Proceedings of Proceedings of DISC'05*, LNCS 3724, pages 152–168, 2005.

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

[8] S. J. Garland, N. A. Lynch, and M. Vaziri. IOA: A language for specifying, programming and validating distributed systems, user and reference manual. Technical report, Massachusetts Institute of Technology, 2004.

[9] M. Gelastou. Formal methods for modeling and verifying an ad hoc network protocol. Master's thesis, University of Cyprus, 2007.

[10] 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.

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

[12] C. Hanson. *Time and Probability in Formal Design of Distributed Systems*. PhD thesis, Uppsala University, 1991.

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

[14] M. Hennessy J. Riely. A typed language for distributed mobile processes. In *Proceedings of POPL'98*.

[15] D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager. Timed I/O Automata: A mathematical framework for modeling and analyzing real-time systems. In *Proceedings of RTSS'03*, pages 166–177. IEEE Computer Society, 2003.

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

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

[18] N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[19] N. A. Lynch, R. Segala, and F. W. Vaandrager. Compositionality for probabilistic I/O Automata. In *Proceedings of CONCUR'03*, LNCS 2761, pages 208–221, 2003.

[20] N. A. Lynch, R. Segala, and F. W. Vaandrager. Hybrid I/O Automata. *Information and Computation*, 185(1):105–157, 2003.

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

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

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

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

[25] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1 and 2. *Information and Computation*, 100:1–77, 1992.

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

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

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

[29] C. Newport. Consensus and collision detectors in wireless ad hoc networks. Master's thesis, Massachusetts Institute of Technology, 2006.

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

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

[32] 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.

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

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

[35] 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.

[36] 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.