28

A Theorem on Atomicity in Distributed Algorithms

Leslie Lamport

May 1, 1988

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

A Theorem on Atomicity in Distributed Algorithms

Leslie Lamport

May 1, 1988

Author's Abstract

Reasoning about a distributed algorithm is simplified if we can ignore the time needed to send and deliver messages and can instead pretend that a process sends a collection of messages as a single atomic action, with the messages delivered instantaneously as part of the action. A theorem is derived that proves the validity of such reasoning for a large class of algorithms. It generalizes and corrects a well-known folk theorem about when an operation in a multiprocess program can be considered atomic.

Capsule Review

In executing a distributed algorithm, process actions and message-delivery actions can be interleaved in numerous ways. The algorithm is correct if it works properly no matter how actions are interleaved. In general, all possible interleavings need to be considered.

This paper presents a theorem that allows some interleavings to be ignored when reasoning about many practical distributed algorithms. Enough of the underlying formalism is described to make it clear that the theorem can be stated precisely and proved. The formal discussions can be skipped by more trusting readers.

Martín Abadi

Contents

1 Introduction

Consider a finite, connected network of processes, where a process can send messages to its neighbors. The following algorithm causes each process i eventually to wind up with its local variable $d[i]$ equal to the distance (number of links in the minimum-length path) from i to a distinguished *root* process. We assume that initially $d[i] = \infty$ for every process i, and all message buffers are empty except for the root's buffer, which contains the single message "0".

Distance-Finding Algorithm

for each process i **do**
  while *true* **do**
    wait until input buffer nonempty;
    remove some message "m" from buffer;
    if $d[i] > m$
      then $d[i] := m$;
        for each neighbor j **do** send "$m + 1$" to j

To prove the correctness of this algorithm, one needs a more precise description of it. We adopt the common approach of formally defining an execution of a concurrent algorithm to be a sequence of atomic actions; concurrent actions of separate processes are assumed to be "interleaved" in an arbitrary manner. A formal description of the Distance-Finding Algorithm requires specifying which of the algorithm's operations are atomic. Consider a single iteration of process i's **while** loop that removes a message "m" from the input buffer, where $d[i] > m$. In a naive representation of the algorithm, each of the following actions might be separate atomic operations.

- Remove message "m" from buffer.
- Test if $d[i] > m$.
- Set $d[i]$ to m.
- Send "$m + 1$" to a neighbor j.

In addition, there would be separate message-delivery actions, performed by the communication network, that put messages into the processes' input buffers.

Reducing the number of atomic actions makes reasoning about a concurrent program easier because there are fewer interleavings to consider. For assertional reasoning, it leads to a simpler invariant and fewer actions to consider in the proof of invariance. The number of atomic actions in the Distance-Finding Algorithm can be reduced by appealing to the following popular observation.

Folk Theorem: When reasoning about a multiprocess program, we can combine into one atomic action any sequence of operations that contains only a single access to a single shared variable.

Although this theorem is usually asserted for shared-variable programs, it applies as well to other kinds of multiprocess program because any form of interprocess communication can be modeled with shared variables.

Since $d[i]$ is local to process i, the Folk Theorem allows us to combine the first three operations—removing the message, evaluating the expression $d[i] > m$, and setting $d[i]$—into a single atomic action. Depending upon how message passing is modeled, the Folk Theorem might also allow the sending of messages to process i's neighbors to be part of the same atomic action. However, the network actions that put the messages into the neighbors' buffers would still be separate actions.

In this paper, we derive a Reduction Theorem that allows one to consider an iteration of process i's **while** loop and the delivery of any messages generated by it to be a single atomic action. Thus, not only are all the operations listed above considered to comprise one atomic action, but the *send* operations put the messages directly into the recipients' input buffers. There are no separate message-delivery actions. Our Reduction Theorem is a generalization of the Folk Theorem. Furthermore, it includes some essential, subtle hypotheses missing from the Folk Theorem.

In general, we consider a distributed algorithm $\mathcal{A}$ in which each process performs a sequence of nonatomic operations, where an operation removes a (possibly empty) set of messages from the process's input buffers, performs some computation, and sends a (possibly empty) set of messages to other processes. Let the *reduced* version $\widehat{\mathcal{A}}$ of algorithm $\mathcal{A}$ be one in which an entire operation is a single atomic action and message transmission is instantaneous—a message appears in the receiver's input buffer when the message is sent. (Any loss or corruption of messages occurs when they are sent.) Algorithm $\widehat{\mathcal{A}}$ is simpler than the original algorithm $\mathcal{A}$, since it has no computation states in which a process is in the middle of an operation or a message is in transit. Hence, it is easier to reason about $\widehat{\mathcal{A}}$ than about $\mathcal{A}$. In this paper, we prove the following:

Reduction Theorem: If conditions C1–C6 (given below) are satisfied, then $\mathcal{A}$ satisfies a correctness property P if and only if $\widehat{\mathcal{A}}$ satisfies P.

The major part of this paper consists of the development of conditions C1–C6. A state-based approach is taken, in which the execution of an algorithm produces a sequence of states, and a property is an assertion about the sequence produced by each individual execution.

The derivation of conditions C1–C6 is perhaps more interesting than the conditions themselves, which are not hard to obtain once one understands why each of them is needed. To prevent simple concepts from being obscured by formalism, the exposition is informal. A sequence of notes indicates how the arguments can be made rigorous, but they do not attempt to give a complete formal exposition. The formalism is at the semantic level, and is independent of language issues. A list of notations appears at the end.

2 The Conditions and Proof of the Theorem

2.1 C1: The Restriction on P

An *execution* of $\mathcal{A}$ consists of a finite or infinite sequence of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots$$

where the s_i are states, the α_i are atomic actions, and $s_{i-1} \xrightarrow{\alpha_i} s_i$ denotes an execution of action α_i that takes the algorithm from state s_{i-1} to state s_i. A state consists of the following:

- The values of a set of *externally visible* variables. An externally visible variable is either *local* to a process, meaning that it is accessed (read or written) only by that process, or *global*, meaning that it is accessed by more than one process.

- The *internal state* of each process, consisting of the state of its input buffers, the values of its *local internal variables*, and its program control state. A process cannot access the internal state of another process.

- The state of the communication network, which describes the status of all messages in transit.

In the Distance-Finding Algorithm, each $d[i]$ is an externally visible variable that is local to process i. Each process has a local internal variable m that holds the value of the message removed from the buffer. A process's control state indicates where the process is in its execution—that is, what statement it will execute next. The state of the communication network could simply be a multi-set of message, source, destination triples, or it could contain additional structure describing the order in which messages may be delivered.

We allow a global externally visible variable to be read and written by any process. Thus, our Reduction Theorem can be applied to algorithms in which processes communicate with shared variables, as well as to distributed algorithms. For programs that communicate only through shared variables, our theorem provides a rigorous formulation of the Folk Theorem. Since the Folk Theorem is so well-known, we will not discuss the application of our theorem to shared-variable programs.

Formalism: We provisionally define an algorithm to be a quadruple (C, $\{S_c : c \in C\}, \mathbf{S}_0, \mathbf{A}$), where C is a set of *state components*, the S_c are sets of *values*, the set of *initial states* $\mathbf{S}_0$ is a subset of the set of *states* $\mathbf{S}$, which is the Cartesian product $\prod\{S_c : c \in C\}$, and $\mathbf{A}$ is a set of *actions*, where an action is defined to be a subset of $\mathbf{S} \times \mathbf{S}$. (The definition is extended later to include liveness conditions.)

An *execution* is a (finite or infinite) sequence $s_0, s_1, \ldots$ of states such that $s_0 \in \mathbf{S}_0$ and, for each s_i with $i > 0$, there is an $\alpha_i \in \mathbf{A}$ such that $(s_{i-1}, s_i) \in \alpha_i$.

For $s \in \mathbf{S}$ and $c \in C$, we let $s.c$ denote the c-component of state s, so $s.c \in S_c$, and let s_v^c denote the state s' such that $s'.c = v$ and $s'.c' = s.c'$ for all $c' \neq c$. An action α *modifies* component c if there exists $(s,t) \in \alpha$ with $s.c \neq t.c$; action α *accesses* component c if it modifies c or if there exist $(s,t) \in \alpha$ and $v \in S_c$ such that $s_v^c \in \mathbf{S}$ and $(s_v^c, t_v^c) \notin \alpha$. (The latter condition is a language-independent definition of what it means for α to read the value of c.)

We assume that the set of actions $\mathbf{A}$ is partitioned into a set of communication actions and a collection of *processes*. (Formally, a process is the set of actions belonging to the process.) We also assume that state components are classified as input buffers, local internal variables, etc. One state component represents the state of the communication network. We assume the existence of a *set of messages in transit* that depends only on the communication network's state.

The first condition for the Reduction Theorem characterizes the class of properties P. We assume that P is a property of executions, and we say that it holds for algorithm $\mathcal{A}$ if it is true for all executions of $\mathcal{A}$. We require that P satisfy the following condition:

C1. P depends only on the sequence of different values assumed by the externally visible variables.

In the Distance-Finding Algorithm, the correctness property P asserts that there exists some n such that, for all $l > n$, state s_l is one in which each $d[i]$ equals the distance of process i to the root. This property satisfies C1 because it depends only upon the sequence of values assigned to the $d[i]$, which are externally visible variables.

Condition C1 requires that P depend on the sequence of values assumed by externally visible variables; not on when (at which step of the execution) those values are assumed. In the physical world, the notion of when an event occurs can be defined only relative to the occurrence of other events—for example, relative to the ticking of a clock or counter. Condition C1 permits the specification of when values are assumed only if the relevant clock or counter is an externally visible variable.

Formalism: Let E denote the set of externally visible state components, and let $\mathcal{E} : \mathbf{S} \rightarrow \prod\{S_c : c \in E\}$ denote the projection mapping. We extend any mapping whose domain is $\mathbf{S}$ to a mapping on the set of sequences of states in the obvious way, so $\mathcal{E}(s_0, s_1, \ldots) = \mathcal{E}(s_0), \mathcal{E}(s_1), \ldots$. For any sequence Σ, let $\natural\Sigma$ denote the sequence obtained by removing repeated elements from Σ—for example, $\natural 1, 2, 2, 2, 3, 3 = 1, 2, 3$ and $\natural 1, 1, 1, \ldots = 1$. Condition C1 asserts that P is a Boolean-valued function on sequences of states such that $\natural\mathcal{E}(\Sigma) = \natural\mathcal{E}(\Sigma')$ implies $P(\Sigma) = P(\Sigma')$.

Even if the desired correctness property depends upon parts of the state that are not externally visible, adding dummy variables[1] to the algorithm usually allows the correctness property to be restated in a form satisfying C1. For example, one might want to prove that the Distance-Finding Algorithm eventually *terminates*, meaning that it reaches a state in which there are no more messages in any input buffer or in transit. As stated, this termination property does not satisfy C1 because it depends upon the state of the communication network and of the processes' input buffers, which are not externally visible variables. (Making them externally visible would violate other hypotheses of the Reduction Theorem.) However, we can add a global externally visible dummy variable x whose value equals the number of unprocessed messages, and we can modify the algorithm so that after process i removes a message from its input buffer, it increments x by the number of messages it is going to send in response minus one. The termination

[1] A dummy variable is one that does not affect the execution of the algorithm and need not be implemented [9].

property is expressed by the assertion that x eventually equals zero—an assertion that satisfies condition C1. Similarly, by adding a dummy variable to count the total number of messages sent, P can express message-complexity properties.

Formalism: Let $\mathcal{A} = (C, \{S_c : c \in C\}, \mathbf{S}_0, \mathbf{A})$, and $\mathcal{A}' = (C', \{S_c : c \in C'\}, \mathbf{S}'_0, \mathbf{A}')$ be algorithms such that $C = C \cup \{y\}$, $\mathcal{Y}(\mathbf{S}') = \mathbf{S}$, and $\mathcal{Y}(\mathbf{S}'_0) = \mathbf{S}_0$, where $\mathbf{S}$ and $\mathbf{S}'$ are the state spaces of $\mathcal{A}$ and $\mathcal{A}'$, respectively, and $\mathcal{Y}$ is the obvious projection mapping. We say that $\mathcal{A}'$ is obtained from $\mathcal{A}$ by adding the dummy component y if there is a one-to-one correspondence $\alpha \leftrightarrow \alpha'$ between $\mathbf{A}$ and $\mathbf{A}'$ such that (i) if $(s', t') \in \alpha'$ then $(\mathcal{Y}(s'), \mathcal{Y}(t')) \in \alpha$ and (ii) if $(s, t) \in \alpha$, $s' \in \mathbf{S}'$, and $\mathcal{Y}(s') = s$, then there exists $t' \in \mathbf{S}'$ such that $(s', t') \in \alpha'$. If $\mathcal{A}'$ is obtained from $\mathcal{A}$ in this way, then Σ is an execution of $\mathcal{A}$ if and only if there is an execution Σ' of $\mathcal{A}'$ such that $\Sigma = \mathcal{Y}(\Sigma')$.

2.2 C2–C5: Actions and Commutativity

An atomic action executed by a process is assumed to be one of the following.

- An *internal* action that may access the process's local internal variables and control state, and may read (but not modify) externally visible variables that are local to the process.
- A *receive* action that removes a message from the process's input buffer; it may read the contents of the buffers, it may access the process's internal state, and it may read the process's local externally visible variables. (The action may be executed only if the input buffer is nonempty.)
- A *send* action that changes the state of the communication network to indicate that an additional message is in transit from this process to another process. (The message's destination is determined when it is sent.) The action may also access the process's local variables and control state and may read the process's local externally visible variables.
- An *externally visible* action that may (but need not) access externally visible variables, variables local to the process, and the process's control state.

In addition to these process actions, we assume that the communication network executes *deliver* actions, which put a message (sent by a previous *send* action) into a process's input buffer. We allow a *deliver* action to corrupt the message or simply destroy it without delivering it, so faulty communication can be modeled. Delivery of multiple copies of a message can be modeled by allowing multiple *send* actions, each sending a copy of the same message. (The program can nondeterministically choose how many copies to send.) Thus, we can model a network that loses, corrupts, or duplicates messages.

Process i of the Distance-Finding Algorithm executes the following actions:

- A *receive* action that waits for the buffer to be nonempty and removes a message from it, storing the message's value in a local internal variable and changing the control state.
- An *internal* action that evaluates the expression $d[i] > m$ and modifies the control state accordingly.
- An *externally visible* action that sets $d[i]$, accessing the local internal variable m and modifying the control state.
- For each neighbor j, a *send* action that initiates the transmission of a message from i to j.

The first condition on $\mathcal{A}$ is

C2. In $\mathcal{A}$, each process's algorithm executes a sequence of operations of the form $R; \langle\, X \,\rangle; L$, where

- R consists only of *receive* or *internal* actions.
- L consists only of *send* or *internal* actions.
- $\langle\, X \,\rangle$ is a single externally visible action.
- If control has reached L, then there exists a terminating execution of L.

The only other actions in $\mathcal{A}$ are *deliver* actions performed by the communication network. It is always possible for all messages in transit to be delivered (or lost) by *deliver* actions without any further process actions.

The requirement that there exists a terminating execution of L rules out, for example, a communication network in which a message cannot be sent until the previous message was delivered—since there would be no terminating execution of L if the previous message had not been delivered.

In the Distance-Finding Algorithm, each iteration of a process's **while** loop is an operation that executes a receive action followed by an internal action (evaluating $d[i] > m$) and then either does nothing or else executes an externally visible action followed by a sequence of send actions. An operation that does not execute an externally visible action can be considered to be part of the "R" of the next iteration's operation. Thus, Condition C2 is satisfied.

Alternatively, we can pretend that when process i finds $d[i] \leq m$, it executes an external action that does not change the value of any externally visible variable. By C1, adding such an action does not affect the truth of property P. Adding this dummy action makes each iteration of a process's loop have the form $R; \langle\, X \,\rangle; L$ of Condition C2. (In the condition, R or L may be null.)

In general, we could extend C2 to allow operations of the form $R; L$, but adding this extra case would complicate our discussion.

For C2 to be satisfied by the modified version of the Distance-Finding Algorithm, where the variable x has been added to detect termination, the same atomic action that changes $d[i]$ must also change x. Since x is a dummy variable added only for the proof, we are free to choose which action modifies it.

Formalism: We assume that the actions in $\mathbf{A}$ are disjoint (sets of pairs of states). This implies that if $\Sigma = s_0, s_1, \ldots$ is an execution, then for each $i > 0$ there is a unique action α_i such that $(s_{i-1}, s_i) \in \alpha_i$, so we can consider Σ to be the sequence $s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \stackrel{\alpha_2}{\rightarrow} s_2 \stackrel{\alpha_3}{\rightarrow} \cdots$. (This representation of Σ is used throughout the proof of the Reduction Theorem. Making the actions in $\mathbf{A}$ disjoint could, but seldom will, require adding dummy variables.)

The internal state of each process contains program control information for that process. This information can be expressed by a function $\mathcal{N}_p$ such that $\mathcal{N}_p(s)$ is the set of possible next actions of process p. For any action α in process p, if there exists a state t with $(s, t) \in \alpha$, then $\alpha \in \mathcal{N}_p(s)$; but the converse need not be true. If an action β in $\mathbf{A}$ is not an action of process p, and $(s, t) \in \beta$, then $\mathcal{N}_p(s) = \mathcal{N}_p(t)$.

A set of actions all belonging to the same process is called an *operation* of that process. A *terminating execution* of an operation A of a process p is a finite sequence $s_0, \ldots, s_n$ such that each (s_{i-1}, s_i) belongs to an element of A and $\mathcal{N}_p(s_n)$ is disjoint from A. An operation A *can terminate from* state s if there exists a terminating execution of A starting with s.

We define ";" by saying that, if A and B are operations of process p, then the operation $A \cup B$ is of the form $A; B$ if the following conditions hold: (i) A and B

are disjoint, (ii) for all $\alpha \in A$, if $(s,t) \in \alpha$ then $\mathcal{N}_p(t)$ is a subset either of A or of B, and (iii) for all $\beta \in B$, if $(s,t) \in \beta$ then $\mathcal{N}_p(s)$ is a subset either of A or of B and $\mathcal{N}_p(t)$ is either a subset of or disjoint from B. It follows that $A \cup B \cup C$ is of the form $(A;B);C$ if and only if it is of the form $A;(B;C)$, in which case we say that it is of the form $A;B;C$.

Condition C2 asserts that the set of actions of each process is the disjoint union of operations of the form $R;\langle X \rangle;L$ for sets of actions R, $\langle X \rangle$, and L, where: (i) $\langle X \rangle$ contains a single action, (ii) the actions in R, $\langle X \rangle$, and L can modify and access the appropriate state components, (iii) if $\mathcal{N}_p(s)$ contains an action in L, then L can terminate from state s, and (iv) for any initial state s in $\mathbf{S}_0$, $\mathcal{N}_p(s)$ contains actions only from the sets R. We assume that *send* and *deliver* actions have the obvious effects on the set of messages in transit, and that *deliver* and *receive* actions are the only ones that access a process's input buffer.

If algorithm $\mathcal{A}$ satisfies C2, then an atomic action of $\widehat{\mathcal{A}}$ has the form $\langle R;\langle X \rangle;\widehat{L} \rangle$, where $R;\langle X \rangle;L$ is an operation of a process p in $\mathcal{A}$, and $\widehat{L}$ consists of the actions of L together with the *deliver* actions that deliver (or lose) messages sent by the *send* actions in L. Given any execution Σ of $\widehat{\mathcal{A}}$, we obtain an execution $\overline{\Sigma}$ of $\mathcal{A}$ by expanding each action $\langle R;\langle X \rangle;\widehat{L} \rangle$ of $\widehat{\mathcal{A}}$ into the sequence $R;\langle X \rangle;\widehat{L}$ of actions of $\mathcal{A}$. The externally visible variables are changed only by $\langle X \rangle$, so it follows from C1 that Σ satisfies property P if and only if $\overline{\Sigma}$ does. Since an algorithm satisfies a property if and only if all its executions do, this implies that if $\mathcal{A}$ satisfies P, then $\widehat{\mathcal{A}}$ also satisfies P.

For convenience, we identify the execution Σ of $\widehat{\mathcal{A}}$ with the corresponding execution $\overline{\Sigma}$ of $\mathcal{A}$. Thus, the set of executions of $\widehat{\mathcal{A}}$ is a subset of the set of executions of $\mathcal{A}$.

Formalism: Let $\circ$ denote the usual composition operator on relations, defined by $(s,u) \in \alpha \circ \beta$ if and only if there exists t such that $(s,t) \in \alpha$ and $(t,u) \in \beta$. For any *send* action σ and *deliver* action δ, let σ^δ be the (possibly empty) subaction of $\sigma \circ \delta$ consisting of all pairs (s,t) for which $s \stackrel{\sigma \circ \delta}{\rightarrow} t$ represents the action of sending a message and then immediately delivering that message. (If the state of the communication network contains unordered multisets of messages, it may be necessary to add a dummy variable for σ^δ to be defined.) Let $\widehat{\sigma}$ be the union of the actions σ^δ for all *deliver* actions δ.

For any operation A, define $\langle A \rangle$ to be the action consisting of the set of all pairs (s,t) such that there exists a terminating execution $s = s_0, s_1, \ldots, s_n = t$ of A with $n > 0$. Condition C2 asserts of $\mathcal{A}$ that the set of actions of each process p is the disjoint union of actions of the form $R;\langle X \rangle;L$. The algorithm $\widehat{\mathcal{A}}$ is defined to have the same components, states, and initial states as $\mathcal{A}$, and to have a set of actions consisting of all the actions $\langle R;\langle X \rangle;\widehat{L} \rangle$, where $\widehat{L}$ is obtained from L by replacing each *send* action σ with $\widehat{\sigma}$.

To complete the proof of the Reduction Theorem, we must prove that if $\widehat{\mathcal{A}}$ satisfies property P then $\mathcal{A}$ does too. We do this by constructing, for every execution Σ of $\mathcal{A}$, a corresponding execution $\widehat{\Sigma}$ of $\widehat{\mathcal{A}}$ such that P is true of Σ if and only if it is true of $\widehat{\Sigma}$. We first consider the case in which Σ is finite—more precisely, when Σ is a finite initial segment of an execution. (Σ may be a complete execution if the execution is finite.) The extension to complete infinite executions is given in Section 2.3.

In an execution of $\mathcal{A}$, actions of other processes and of the communication network may be interleaved between the actions of a single operation $R; \langle\, X \,\rangle; L$ and between the *send* actions in L and their corresponding *deliver* actions. We construct $\widehat{\Sigma}$ from Σ by permuting the order in which actions are executed so that there are no other actions interleaved between the actions in a single operation $R; \langle\, X \,\rangle; \widehat{L}$. We do this by moving actions of R to the right and actions of $\widehat{L}$ to the left. In constructing $\widehat{\Sigma}$, we first delete any action from a partially completed operation in which the $\langle\, X \,\rangle$ action has not been executed (which we can do because actions in R affect only the process's internal state) and complete any unfinished operation in which $\langle\, X \,\rangle$ has been executed (which we can do because condition C2 guarantees the existence of a terminating execution of L) and add actions to deliver any outstanding messages (which C2 allows us to do).

We say that an atomic action ρ *right commutes* with an atomic action λ, or that λ *left commutes* with ρ, if and only if, whenever $\rho; \lambda$ (a ρ action followed by a λ action) can be executed, it is also possible to produce the same result by executing $\lambda; \rho$. In other words, if $s \xrightarrow{\rho} t \xrightarrow{\lambda} u$ is possible then $s \xrightarrow{\lambda} t' \xrightarrow{\rho} u$ is possible for some state t'. Two actions are said to *commute* if and only if each right commutes with the other. Commutativity of two actions means that executing them in either order has the same effect.

Formalism: Action ρ right commutes with action λ if and only if $\rho \circ \lambda \subseteq \lambda \circ \rho$. If neither action accesses any component modified by the other action, then $\rho \circ \lambda = \lambda \circ \rho$, so the actions commute.

We will construct $\widehat{\Sigma}$ from Σ by a series of interchanges, replacing a sequence of the form $\cdots \xrightarrow{\rho} s \xrightarrow{\lambda} \cdots$ by $\cdots \xrightarrow{\lambda} s' \xrightarrow{\rho} \cdots$. We can do this if ρ right commutes with λ.

To construct $\widehat{\Sigma}$ from Σ, actions in R must be moved to the right, while actions in L and *deliver* actions must be moved to the left. Actions belonging to the same process do not have to be interchanged, so commutativity relations between actions from the same process are not needed. Two actions obviously commute if they do not both access the same variable or state component, so we have the following commutativity relations.

- An *internal* action commutes with every action not belonging to the same process.
- An "$\langle\, X \,\rangle$" action commutes with every *deliver* action and every action of another process except another "$\langle\, X \,\rangle$" action.
- A *receive* action commutes with all actions in other processes, and with *deliver* actions delivering messages to other processes.

By C2, R contains only *receive* and *internal* actions, and L contains only *send* and *internal* actions. Therefore, $\widehat{\Sigma}$ can be constructed by commuting the actions of Σ if the following commutativity relations are satisfied.

- A *send* action must commute with
  - *send* actions of other processes.
  - *deliver* actions.
- A *receive* action in a process p must right commute with actions that deliver a message to p.
- A *deliver* action delivering a message to process p must
  - commute with other *deliver* actions.
  - commute with *send* actions.
  - left commute with *receive* actions process p.

These commutativity relations are sufficient to allow the construction of $\widehat{\Sigma}$, but they are not all necessary. A *send* action need not commute with the corresponding *deliver* action—the one that delivers the message that the *send* had sent. Also, two *deliver* actions need not commute if they occur in the same order as their corresponding *send* actions. The remaining commutativity relations are implied by the following three conditions, where $\Delta(p, q)$ denotes the set of *deliver* actions that deliver to process q a message sent by process p.

C3. A *send* action σ commutes with every *send* action in another process and with every *deliver* action except the one that delivers the message sent by σ.

C4. A *receive* action of process p right commutes with every *deliver* action that delivers a message to p.

C5. For every pair of processes p, q: if messages from p to q are delivered in the order in which they are sent, then every action in $\Delta(p, q)$ commutes with every *deliver* action not in $\Delta(p, q)$; otherwise, if messages may be delivered out of order, then every action in $\Delta(p, q)$ commutes with every other *deliver* action (including ones in $\Delta(p, q)$).

The following are two examples of communication schemes that satisfy these conditions.

(a) The state of the communication system consists of an unordered set of message, source, destination triples; and each process's input buffer is an unordered set of message, source pairs. A process can receive any message in its input buffer.

(b) The state of the communication system contains a FIFO (first-in-first-out) message queue for each sender, receiver pair; and each process has a separate FIFO input buffer for each sender process. A process can receive a message at the head of any queue.

Condition C3 is not satisfied if a process that tries to send a message can be suspended because other processes have filled the network's message buffers, so the condition essentially requires unbounded buffering by the communication network. Although communication schemes can be devised that fail to satisfy C3 despite having unbounded buffering, they don't seem to arise in practice.

Condition C4 states that if a *receive* action can be performed before a message is delivered, then that same action can be performed after the delivery. We can restate this condition somewhat more informally as:

C4$'$. A process's operation cannot depend upon the absence of a message.

For example, the algorithm cannot require that a certain action be taken only if a process's input buffer is empty. In example (b) above, C4$'$ implies that a process cannot query its input queues in a fixed order, since there would then be states in which the absence of a message in one queue is necessary for the process to receive a message from the following queue.

There appears to be no simple, intuitive restatement of condition C5. However, the two examples above are common enough that they are worth stating as the following condition, which implies C5.

C5′. For each process p, either

(a) p has an input buffer consisting of an unordered set of messages, or

(b) p has a separate input queue for each process from which it receives messages, and messages from any single process are delivered in the order that they are sent.

For example, process p cannot maintain a single FIFO input queue in which it puts messages from all processes. If it did, two *deliver* actions that deliver messages from different processes would not commute because reversing their order of execution reverses the order of the messages in the queue.

Do C3–C5 hold for the Distance-Finding Algorithm? C3 is a condition on the communication network, which we haven't specified. It is implied by the assumption of unbounded buffering usually made when studying this type of algorithm. Condition C4 asserts that receipt of a message cannot prevent a process from performing an action that it could have performed had the message not arrived—an assertion that holds for this algorithm. Condition C5 depends upon the queueing discipline employed by the algorithm. By not specifying which message is to be removed from the buffer, we have allowed each process to maintain a single buffer containing an unordered set of messages—an implementation for which C5′(a) holds.

Since no queueing policy is specified, the Distance-Finding Algorithm can be implemented by any policy. The most general queueing policy is represented by a single, unordered buffer. Any other policy is a special case, whose executions are the same as possible executions with the unordered buffer. The correctness of the more general algorithm implies the correctness of the special case. For example, the buffer could be implemented as a single FIFO queue. However, C5 does not hold for this queueing discipline, so if the algorithm were to specify a single FIFO buffer, then our Reduction Theorem would not apply. We would then have to generalize the algorithm to allow an unordered buffer in order to simplify the proof.

Formalism: The formal statement of Conditions C3 and C4 is straightforward, since they simply express commutativity relations among the actions of $\mathbf{A}$. In C3, the fact that commutativity is not required between the actions of sending and delivering the same message is expressed by requiring for any *send* action σ and *deliver* action δ only that $\sigma \circ \delta = \delta \circ \sigma \cup \sigma^{\delta}$, rather than full commutativity.

Condition C5 assumes that the set of communication network actions can be partitioned into the sets $\Delta(p, q)$. To make this partition possible, one might have to modify **A** by partitioning a single action α into subactions $\alpha_1, \ldots, \alpha_m$. Such a change does not alter the set of executions.

2.3 Safety, Liveness, and C6

Conditions C2–C5 guarantee that, for any finite initial segment Σ of an execution of $\mathcal{A}$, we can construct an execution $\widehat{\Sigma}$ in which the actions in any process's operation and the corresponding *deliver* actions are contiguous. Moreover, P holds for Σ if and only if it holds for $\widehat{\Sigma}$. Before considering arbitrary executions, we must return to the question of how one specifies an algorithm.

The specification of an algorithm is the conjunction of two parts: a *safety specification* that describes what the actions *may* do, and a *liveness specification* that describes what actions *must* eventually be performed.[2] Consider an algorithm containing the program statement $\langle\, x := x + 1 \,\rangle$. The algorithm's safety specification implies that executing this statement may change the value of x only by adding one to it, but it does not imply that the statement is ever executed. A requirement that the statement must eventually be executed when control reaches it would be part of the liveness specification, which is usually implicit in the semantics of the programming language.

In general, the safety specification may be any safety property, which is one that holds for an execution if and only if it holds for all finite initial segments of the execution. Mutual exclusion, FIFO service, and partial correctness are all safety properties.

The liveness specification must be a liveness property, which is one for which any finite sequence of states and actions can be extended to a sequence that satisfies the property [1]. This definition is independent of any algorithm. A liveness specification may not be an arbitrary liveness property, but must satisfy the stronger requirement that any finite sequence of states and actions that satisfy the algorithm's safety specification can be extended to a sequence that satisfies both its liveness and safety properties. This stronger requirement essentially means that the liveness specification does not specify any additional safety properties; it is satisfied by all commonly used liveness specifications.

An arbitrary property P holds for an algorithm if and only if it is implied by the conjunction of the algorithm's safety and liveness specifications. But a safety property holds for an execution if and only if it holds for every

[2]The term "fairness" is sometimes used in place of "liveness".

finite initial segment of the execution, and every such segment that satisfies the safety specification can be extended to an execution that satisfies both the safety and the liveness specifications. Therefore, a safety property is satisfied by the algorithm if and only if it is implied by the algorithm's safety specification alone, which is true if and only if the property holds for every finite initial segment of every execution.

Conditions C2–C5 were chosen to guarantee that the execution $\widehat{\Sigma}$ constructed from the finite initial segment Σ of an execution of $\mathcal{A}$ satisfies the safety specification of $\widehat{\mathcal{A}}$. Hence, $\widehat{\Sigma}$ is a finite initial segment of an execution of $\widehat{\mathcal{A}}$. Moreover, C1 implies that P holds for $\widehat{\Sigma}$ if and only if it holds for Σ. Hence, our construction of $\widehat{\Sigma}$ from Σ proves that if P is a safety property, then $\mathcal{A}$ satisfies P if and only if $\widehat{\mathcal{A}}$ does. We have therefore proved the Reduction Theorem for a safety property P without using C6. Condition C6 need apply only when P is not a safety property.

Formalism: Let Σ be any finite portion of an execution of $\mathcal{A}$. Let Σ' be obtained from Σ by appending to it L actions and *deliver* actions so that, in the last state, there are no undelivered messages and control in every process is either not inside its operation or inside its R operation. (Condition C2 implies the existence of Σ'.) Since no actions have been added that affect the externally visible state, C1 implies that Σ' satisfies P if and only if Σ does. By commuting actions as allowed by C2–C5 and the assumptions about which actions can access and modify which state components, we can transform Σ' to a sequence $\widehat{\Sigma}$ of the form $\Upsilon_1, \ldots \Upsilon_i, \Phi$, where each Υ_j is a subsequence consisting of a complete execution of the operation $R; \langle\, X \,\rangle; \widehat{L}$ of some process and Φ consists only of R actions. (Each *deliver* action δ is moved left until reaching a position $\cdots s \xrightarrow{\sigma} t \xrightarrow{\delta} u$ for a *send* action σ with $(s, u) \in \sigma^{\delta}$.) Moreover, the states immediately before and after each $\langle\, X \,\rangle$ action are the same in Σ' and in $\widehat{\Sigma}$, so C1 implies that Σ' satisfies P if and only if $\widehat{\Sigma}$ does. But $\widehat{\Sigma}$ is an execution of $\widehat{\mathcal{A}}$, so we have proved that, for every finite execution Σ of $\mathcal{A}$, there exists an execution $\widehat{\Sigma}$ of $\widehat{\mathcal{A}}$ that satisfies P if and only if Σ does. This proves the Reduction Theorem if P is a safety property.

To prove the Reduction Theorem for any arbitrary property P, we need to construct $\widehat{\Sigma}$ when Σ is an infinite execution of $\mathcal{A}$. Conditions C2–C5 are not enough to make this construction possible. In $\widehat{\Sigma}$, every process operation $R; \langle\, X \,\rangle; L$ is completed and every message sent by L is delivered. In the finite case, we could complete unfinished operations by adding actions to the end of Σ. We cannot do this in the infinite case; the actions must already be in Σ. To construct $\widehat{\Sigma}$, in the execution Σ every process operation must be completed and every message delivered. This can be guaranteed by requiring that these conditions be part of $\mathcal{A}$'s liveness specification. ("Delivery" of a message includes the possibility that the message is destroyed, so requiring eventual delivery does not rule out the possibility of losing messages.) With this requirement, we can construct $\widehat{\Sigma}$ as the limit of the sequences $\widehat{\Sigma^n}$, where Σ^n consists of the first n steps of Σ. (The required liveness conditions implies that each operation of $\widehat{\Sigma}$ consists of actions from Σ.)

Requiring these liveness conditions to be part of $\mathcal{A}$'s liveness specification ensures that $\widehat{\Sigma}$ can be constructed, but it does not guarantee the validity of the Reduction Theorem if the specification contains other liveness conditions as well. The problem is that $\widehat{\Sigma}$ need not satisfy these other liveness properties, so it need not be an execution of $\widehat{\mathcal{A}}$. Thus, P can hold for $\widehat{\mathcal{A}}$ without holding for $\mathcal{A}$. As an example, consider the following algorithm $\mathcal{A}$ with two processes, p and q. Process p repeatedly performs an operation that sends two messages to q; process q repeatedly performs an operation that removes one message from its input queue and then nondeterministically sets the externally visible variable x to either 0 or 1. To this safety specification we add the liveness requirement that if q's input buffer ever contains two messages, then some later action of q (not necessarily the next one) must set x to 1. Let property P assert that x must equal 1 at some point in the execution. In algorithm $\widehat{\mathcal{A}}$, the two messages that p's operation sends to q are put into the buffer simultaneously, so the liveness requirement implies that P holds for every execution of $\widehat{\mathcal{A}}$. However, $\mathcal{A}$ has a possible execution Σ in which process q removes messages from its buffer as fast as they arrive, so its buffer never contains two messages, and it always sets x equal to 0. (For this Σ, the sequence $\widehat{\Sigma}$ is not an execution of $\widehat{\mathcal{A}}$.) Then P holds for $\widehat{\mathcal{A}}$ but not for $\mathcal{A}$.

The simplest statement of the precise condition C6 needed to complete the Reduction Theorem is that, when P is not a safety condition, if Σ satisfies the liveness specification of $\mathcal{A}$ then the sequence $\widehat{\mathcal{A}}$ can be constructed and satisfies the liveness specification. However, such a condition is not very convenient because verifying it requires reasoning about executions. Instead, we give the following more restrictive condition that seems to handle most cases of interest. An action α is said to be *enabled* in a state if it is possible to execute α starting in that state—that is, if the safety specification allows such an execution of α.

C6. If P is not a safety property, then the liveness specification for $\mathcal{A}$ *must* include the following conditions:

- Every process operation (which by C2 has the form $R; \langle X \rangle; L$) that is begun is eventually completed.

- For every execution of a *send* action there is a corresponding execution of a *deliver* action that delivers (or destroys) the message that was sent.

The liveness specification also *may* include any of the following types of conditions:

- For the entire algorithm: $\mathcal{A}$ does not halt if some action is enabled.
- For an individual process p:
  - If there is a message in p's input buffer, then some action of p is eventually executed.
  - If there is a message from a particular process q in p's input buffer, then p eventually removes some message from q from its input buffer.
- For the communication network: if infinitely many messages are sent from process p to process q, then infinitely many of them eventually arrive at their destination.

Condition C6 has two parts. The first part describes the conditions that the liveness specification must contain; it guarantees that the sequence $\widehat{\Sigma}$ can be constructed for any execution Σ of $\mathcal{A}$. The sequence $\widehat{\Sigma}$ obviously also satisfies these conditions. The second part describes the only other conditions that the liveness specification may (but need not) contain. To complete the proof of the Reduction Theorem, we need only show that if Σ satisfies any such condition, then $\widehat{\Sigma}$ does as well. It is easy to check that this is the case. For example, if Σ satisfies the last kind of allowed condition, then $\widehat{\Sigma}$ also satisfies it because every message that is sent from p to q in execution Σ, or that arrives at its destination in execution Σ, also does so in execution $\widehat{\Sigma}$.

In the Distance-Finding Algorithm, we have tacitly assumed a liveness specification with the following conditions:

1. If there is a message in process p's input buffer, then (a) some message is removed from the buffer and (b) the entire operation of reading the message and reacting to it is eventually completed.

2. Every message that is sent eventually arrives at its destination.

Condition 1(a) is a type of condition allowed by C6, and 1(b) is the first of the two conditions required by C6. Condition 2 is the conjunction of two conditions: (a) every *send* action has a corresponding *deliver* action, which is the second of C6's required conditions, and (b) no *deliver* action destroys a message, which is part of the safety specification. Therefore, the Distance-Finding Algorithm satisfies C6.

Formalism: We must extend our original definition of an algorithm as a quadruple $(C, \{S_c : c \in C\}, \mathbf{S}_0, \mathbf{A})$, to include a liveness specification. The liveness conditions used in specifying most algorithms can be expressed by adding a set of *weak fairness* conditions and a set of *strong fairness* conditions. A fairness condition is a pair (L, Γ) where L is a Boolean-valued function on the set of states and Γ is a subset of the set of actions.

An infinite sequence $s_0, s_1, \ldots$ satisfies the weak fairness condition (L, Γ) if and only if the following condition is satisfied (where $\in\in$ means "is an element of an element of")

$$\forall i\ \exists j \geq i :\ (s_j, s_{j+1}) \in\in \Gamma \text{ or } \neg L(s_j)$$

The sequence satisfies the strong fairness condition (L, Γ) if and only if the following condition is satisfied

$$\forall i\ \exists j \geq i :\ (s_j, s_{j+1}) \in\in \Gamma \text{ or } \forall k \geq j : \neg L(s_k)$$

A finite sequence $s_0, \ldots, s_n$ is considered to be equivalent to the infinite one $s_0, \ldots, s_n, s_n, s_n, \ldots$. An execution of the algorithm is now required to satisfy the fairness conditions.

The liveness conditions allowed by C6 for the entire algorithm and for an individual process are weak fairness conditions. The condition allowed for the communication network is a strong fairness condition (L, Γ), where L asserts that a message has been sent from p to q and Γ is the set of actions that successfully deliver such a message.

The required condition that each *send* has a corresponding *deliver* implies that for any portion of an execution $s_i \xrightarrow{\sigma} s_{i+1} \cdots s_j$ where σ is a *send* action, we can determine if the message sent by σ has already been delivered when state s_j is reached. If this can be determined by just examining state s_j, then the condition can be expressed by weak fairness conditions. Otherwise, it is a more complicated type of condition and must be added separately to the liveness specification.

C6's required liveness conditions allow us to extend to infinite executions the method given above for constructing the execution $\widehat{\Sigma}$ of $\widehat{\mathcal{A}}$ from the finite execution Σ of $\mathcal{A}$. As before, Σ satisfies P if and only if $\widehat{\Sigma}$ does. To prove the Reduction Theorem, we must show that if the execution Σ satisfies any of the liveness conditions allowed by C6, then $\widehat{\Sigma}$ also satisfies these conditions.

C6's entire-algorithm condition is maintained because, if Σ does not halt, then neither does $\widehat{\Sigma}$. An individual-process condition allowed by C6 is a weak fairness condition of the form (L, Γ) where Γ is a set of *receive* actions. Moreover, L is initially false; it is made true by executing a *deliver* action; and it is made false again only by executing a corresponding action of Γ. This weak fairness condition asserts that an execution contains either an infinite number of Γ actions, or else L is false infinitely often. If Σ has an infinite number of Γ actions, then so does $\widehat{\Sigma}$. If Σ has only a finite number of Γ actions, then L false infinitely often implies that there are only a finite number of *deliver* actions that make L true, each of which has a *receive* action that makes L false again. If this latter condition holds for Σ, then it must also hold for $\widehat{\Sigma}$, which is obtained from Σ by commuting *receive* actions to the right and *deliver* actions to the left.

A communication-network condition allowed by C6 is a strong fairness condition (L, Γ) where L is made true by executing a *send* action and is made false only by executing a corresponding *deliver* action in Γ. In constructing $\widehat{\Sigma}$, a *deliver* action is never moved to the right of its corresponding *send* action, so $\widehat{\Sigma}$ satisfies the condition if Σ does.

3 Discussion

The six hypotheses of the Reduction Theorem may seem like a formidable array of conditions that would prevent the theorem from being of much practical value. However, the Distance-Finding Algorithm is not a fluke, but rather an example of a broad class of distributed algorithms to which the theorem can be applied. Condition C3 implies unbounded buffering, which is assumed of most distributed algorithms considered in the literature. The only condition that eliminates a large class of algorithms is C4. By requiring that the receipt of a message not disable an action, C4 rules out real-time algorithms in which a process does something when it has not received a message within a certain length of time.

C4 may also be violated because of unnecessary overspecification of the input buffer. The well-known minimum spanning tree algorithm of Gallager, Humblet, and Spira, as described in [5], does not satisfy C4 because it specifies that each process maintain a single FIFO input queue. The algorithm does not require the single queue; it can be generalized by having a process maintain a separate queue for each neighboring process.[3] This is still not sufficient, because the algorithm moves certain messages that cannot be processed immediately to the end of the input queue. C4 is not satisfied because the action of moving a message to the end of the queue does not right commute with the action of delivering a new message to the queue; the order of messages in the queue depends upon the order in which the actions

[3] Multiple input queues are a generalization because they can be implemented by a single queue.

are executed. However, the algorithm can just as well be implemented by not moving a message to the back of the queue, but allowing messages later in the queue to be processed before it. With this additional modification, the minimum spanning tree algorithm satisfies C1–C6, and the reduction theorem can be applied.

Our Reduction Theorem can be applied to a multiprocess algorithm in which there is no message passing, so all interprocess communication is performed with global, externally visible shared variables. In this case, C3–C5 are vacuous, and condition C2 is just the hypothesis of the Folk Theorem. However, conditions C1 and C6, which are not mentioned by the Folk Theorem, are not vacuous. These or similar conditions are necessary for the Folk Theorem to be valid.

The Folk Theorem asserts that two programs—the original and the reduced version—are equivalent. Equivalence means that they satisfy the same properties, and it can be valid only if one specifies the class of properties under consideration. Condition C1 rectifies this omission from the Folk Theorem.

Condition C6, which is needed to apply the Reduction Theorem to liveness properties, is a more insidious omission from the hypotheses of the Folk Theorem. The Folk Theorem is not valid for arbitrary liveness properties without some additional hypothesis such as C6. Counterexamples are easily obtained by using liveness specifications that determine under what conditions a process is guaranteed eventually to execute its next action. For example, consider a multiprocess program with the following process

$$
\begin{array}{l}
\langle\, x := 2 \,\rangle; \\
\textbf{while } \mathit{true} \textbf{ do } \langle\, x := 1 \,\rangle; \\
\qquad\qquad\qquad \langle\, n := n + 1 \,\rangle; \\
\qquad\qquad\qquad \langle\, x := 2 \,\rangle \\
\qquad\quad\ \ \textbf{od}
\end{array}
$$

where x is local to the process. The Folk Theorem would allow us to make the entire loop body a single atomic action. However, suppose that the program contained the liveness specification that the process is only guaranteed to take a next step when $x \neq 1$. The reduced program satisfies the liveness property that n must get arbitrarily large, but the original program does not, since it permits an execution in which this process does nothing after the first time it sets x to 1.

Acknowledgments

In [8], Lipton proved a reduction theorem similar to ours for reasoning about partial correctness and deadlock-freedom properties of nondistributed programs, concentrating on programs that use semaphores. His result was extended by Doeppner [4] to a somewhat larger class of safety properties. In [2] and [3], Dijkstra proved a restricted version of our reduction theorem for reasoning about partial correctness properties of two-process distributed programs. Using a formalism based upon event traces instead of states, Jonsson proved in [6] what is essentially a special case of our reduction theorem for a system with FIFO buffers, and he cited a related result by Pachl for reachability and deadlock properties. There have undoubtedly been many other variations on the same theme that we are unaware of. The observation that the Folk Theorem is not valid for liveness properties was made by Reino Kurki-Suonio and Ralph Back, and reported to us by Kurki-Suonio.

Discussions with Fred Schneider led to the writing of this paper, which in turn led to our generalizing Lipton's and Doeppner's results in [7]. I wish to thank Martín Abadi, Eike Best, Richard Koo, Michael Merritt, Gil Neiger, Van Nguyen, Prasad Sistla, and Sam Toueg for their comments on earlier drafts.

References

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[2] Edsger W. Dijkstra. When messages may crawl. 1979. EWD708.

[3] Edsger W. Dijkstra. When messages may crawl, ii. 1979. EWD710.

[4] Thomas W. Doeppner, Jr. Parallel program correctness through refinement. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 155–169, ACM, January 1977.

[5] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[6] Bengt Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, Uppsala University, 1987.

[7] Leslie Lamport and Fred B. Schneider. *Pretending Atomicity.* research report 44, Digital Equipment Corporation, Systems Research Center, May 1989.

[8] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[9] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

List of Notations

$\mathbf{A}$	The set of program actions.
$\mathcal{A}$	The algorithm under consideration.
$\widehat{\mathcal{A}}$	The reduced version of algorithm $\mathcal{A}$.
$\langle A \rangle$	The action obtained by executing the operation A as an atomic action.
C	The set of state components.
$d[i]$	A variable of the Distance-Finding Algorithm.
L	An operation of $\mathcal{A}$, as in C2.
$\widehat{L}$	The operation obtained by adding to L the actions that deliver messages sent by L.
$\mathcal{N}_p(s)$	The set of possible next actions of process p from state s.
P	The correctness property.
R	An operation of $\mathcal{A}$, as in C2.
$\mathbf{S}$	The set of states.
$\mathbf{S}_0$	The set of initial states.
S_c	The range of values of state component c.
$\langle X \rangle$	An action of $\mathcal{A}$, as in C2.
Σ	Usually denotes an execution of $\mathcal{A}$.
$\widehat{\Sigma}$	The execution of $\widehat{\mathcal{A}}$ that corresponds to an execution Σ of $\mathcal{A}$.