A Lattice-Structured Proof Technique Applied to a Minimum Spanning Tree Algorithm (Extended Abstract)

Jennifer Lundelius Welch
Laboratory for Computer Science, Massachusetts Institute of Technology

Leslie Lamport
Digital Equipment Corporation, Systems Research Center

Nancy Lynch
Laboratory for Computer Science, Massachusetts Institute of Technology

Abstract: Highly-optimized concurrent algorithms are often hard to prove correct because they have no natural decomposition into separately provable parts. This paper presents a proof technique for the modular verification of such non-modular algorithms. It generalizes existing verification techniques based on a totally-ordered hierarchy of refinements to allow a partially-ordered hierarchy—that is, a lattice of different views of the algorithm. The technique is applied to the well-known distributed minimum spanning tree algorithm of Gallager, Humblet and Spira, which has until recently lacked a rigorous proof.

1. Introduction

The proliferation of distributed computer systems gives increasing importance to correctness proofs of distributed algorithms. Techniques for verifying sequential algorithms have been extended to handle concurrent and distributed ones—for example, by Owicki and Gries [OG], Manna and Pnueli [MP], Lamport and Schneider [LSc], and Alpern and Schneider [AS]. Practical algorithms are usually optimized for efficiency rather than simplicity, and proving them correct may be feasible only if the proofs can be structured. For a sequential algorithm, the proof is structured by developing a hierarchy of increasingly detailed versions of the algorithm and proving that each correctly implements the next higher-level version. This approach has been extended to concurrent algorithms by Lamport [L], Stark [S], Harel [H], Kurshan [K], and Lynch and Tuttle [LT], where a single action in a higher-level representation can represent a sequence of lower-level actions. The higher-level versions usually provide a global view of the algorithm, with progress made in large atomic steps and a large amount of nondeterminism allowed. At the lowest level is the original algorithm, which takes a purely local view, has more atomic steps, and usually has more constraints on the order of events.

With its totally ordered chain of versions, this hierarchical approach usually does not allow one to focus on a single task in the algorithm. The method described in this paper extends the hierarchical approach to a lattice of versions. At the bottom of the lattice is the original algorithm, which is a refinement of all other versions. However, two versions in the lattice may be incommeasurable, neither one being a refinement of the other.

Multiple higher-level versions of a communication protocol, each focusing on a different function, were considered by Lam and Shankar [LSh]. They called each higher-level version a "projection". If the original protocol is sufficiently modular, then it can be represented as the composition of the projections, and the correctness of the original algorithm follows immediately from the correctness of the projections. This approach was used by Fekete, Lynch, and Shrira [FLS] to prove the correctness of Awerbuch's synchronizer [A1].

The work of Lynch and Welch was supported in part by the Office of Naval Research under Contract N00014-85-K-0168 by the National Science Foundation under Grant CCR-8611442, and by the Defense Advanced Research Projects Agency under Contract N00014-83-K-0125.

Not all algorithms are modular. In practical algorithms, modularity is often destroyed by optimizations. The correctness of a non-modular algorithm is not an immediate consequence of the correctness of its higher-level versions. The method presented in this paper uses the correctness of higher-level versions of an algorithm to simplify its proof. The proofs of correctness of all the versions in the lattice (in which the original algorithm is the lowest-level version) constitute a structured proof of the algorithm.

Any path through our lattice of representations ending at the original algorithm is a totally-ordered hierarchy of versions that can be used in a conventional hierarchical proof. Why do we need the rest of the lattice? Each version in the lattice allows us to formulate and prove invariants about a separate task performed by the algorithm. These invariants will appear somewhere in any assertional proof of the original algorithm. Our method permits us to prove them at as high a level of abstraction as possible.

The method proceeds inductively, top-down through the lattice. First, the highest-level version is shown directly to have the original algorithm's desired property, which involves proving that it satisfies some invariant. Next, let A be any algorithm in the lattice, let $B_1, \ldots, B_i$ $(i \geq 1)$ be the algorithms immediately above A in the lattice, and let $Q_1, \ldots, Q_i$ be their invariants. We prove that A satisfies the same safety properties as each B_j, and that a particular predicate P is invariant for A. The invariant P has the form $Q \wedge Q_1 \wedge \cdots \wedge Q_i$ for some predicate Q. In this way, the invariants Q_j are carried down to the proof of lower-level algorithms, and Q introduces information that cannot appear any higher in the lattice—information about details of the algorithm that do not appear at higher levels, and relations between the B_j. We provide two sets of sufficient conditions for verifying these safety properties, one set for the case $i = 1$, and the other for $i > 1$. We also provide three techniques for verifying liveness properties; only one of them makes use of the lattice structure.

The technique is used to prove Gallager, Humblet and Spira's distributed minimum spanning tree algorithm [GHS]. This algorithm has been of great interest for some time. There appears in [GHS] an intuitive description of why the algorithm should work, but no rigorous proof. There are several reasons for giving a formal proof. First, the algorithm has important applications in distributed systems, so its correctness is of concern. Second, the algorithm often appears as part of other algorithms [A2,AG], and the correctness of these algorithms depends upon the correctness of the minimum spanning tree algorithm. Finally, many concepts and techniques have been taken from the algorithm, out of context, and used in other algorithms [A2,CT,G]. Yet the pieces of the algorithm interact in subtle ways, some of which are not explained in the original paper. A careful proof of the entire algorithm can indicate the dependencies between the pieces.

Our proof method helped us to find the correct invariants; it allowed us to describe the algorithm at a high level, yet precisely, and to use our intuition about the algorithm to reason at an appropriate level of abstraction. A by-product of our proof was a better understanding of the purpose and importance of certain parts of the algorithm, enabling us to discover a slight optimization.

The complete proof of the correctness of this minimum spanning tree algorithm is very long and can be found in [W]. One reason for its length is the intricacy of the algorithm. Another reason is the duplication inherent in the approach: the code in all the versions is repetitive, because of carry-over from a higher-level version to its refinement, and because the original algorithm cannot be presented as a true composition of its immediate projections; the repetition in the code leads to repetition in the proof. The full proof also includes extremely detailed arguments—detailed enough so we hope that, in the not too distant, future, they will be machine-checkable. This level of detail seems necessary to catch small bugs in the program and the proof.

Two other proofs of this algorithm have recently been developed. Stomp and de Roever [SdR] used the notion of communication-closed layers, introduced by Elrad and Francez [EF]. Chou and Gafni [CG] prove the correctness of a simpler, more sequential version of the algorithm and then prove that every execution of the original algorithm is equivalent to an execution of the more sequential version.

2. Lattice-Structured Proofs

This section contains the definitions and results that form the basis for our lattice-structured proof method. Proofs of the results may be found in [W].

Our method can be used with any state-based, assertional verification technique. In this paper, we formulate it in terms of the I/O automaton model of Lynch, Merritt, and Tuttle [LT,LM], which provides a convenient, ready-made "language" for our use. A summary of the I/O automaton model appears in Appendix A.

The first step is to design the lattice, using one's intuition about the algorithm. Each element in the lattice is a version of the algorithm, described as an I/O automaton, and has associated with it a predicate. The bottom element of the lattice is the original algorithm. Next, we must show that all the predicates in the lattice are invariants. The invariant for the top element of the lattice must be shown directly. Assuming that $Q_1, \ldots, Q_i$ are invariants for the versions $B_1, \ldots, B_i$ directly above A in the lattice, we verify that predicate $P = Q \wedge Q_1 \wedge \cdots \wedge Q_i$ is invariant for A, by demonstrating mappings that preserve Q and take executions of A to executions of $B_1, \ldots, B_i$ (thus preserve $Q_1 \wedge \cdots \wedge Q_i$). (Finding these mappings requires insight about the algorithm.) Finally, the lattice is used to show that the original algorithm solves the problem of interest by showing directly that the top element in the lattice solves the problem, and showing a path $A_1, \ldots, A_k$ in the lattice from top to bottom such that each version in the path *satisfies* its predecessor. To show that A_i satisfies A_{i-1}, we show that for every fair execution of A_i, there is a fair execution of A_{i-1} with the same sequence of external actions. The mapping used to verify the invariants takes executions to executions; by adding some additional constraints on the mapping, we can prove, using the invariants, that it takes fair executions to fair executions with the same sequence of external actions, i.e., that liveness properties are preserved.

Let A and B be automata, where B is offered as a "more abstract" version of A. We only consider automata such that each locally-controlled action is in a separate class of the action partition. (The results of this section can be generalized to avoid this assumption.) Let *alt-seq*(B) be the set of all finite sequences of alternating actions of B and states of B that begin and end with an action, including the empty sequence (and the sequence of a single action). An *abstraction mapping* $\mathcal{M}$ from A to B is a pair of functions, $\mathcal{S}$ and $\mathcal{A}$, where $\mathcal{S}$ maps *states*(A) to *states*(B) and $\mathcal{A}$ maps pairs (s, π), of states s of A and actions π of A enabled in s, to *alt-seq*(B). $\mathcal{S}$ tells how to view a low-level state at a higher level of abstraction; $\mathcal{A}$ tells what high-level actions, if any, correspond to the low-level action π executed in state s.

Given execution fragment $e = s_0 \pi_1 s_1 \ldots$ of A, define $\mathcal{M}(e)$ as follows.

- If $e = s_0$, then $\mathcal{M}(e) = \mathcal{S}(s_0)$.
- Let $e = s_0 \ldots s_{i-1} \pi_i s_i$, $i > 0$. If $\mathcal{A}(s_{i-1}, \pi_i)$ is empty, then $\mathcal{M}(e) = \mathcal{M}(s_0 \ldots s_{i-1})$. Otherwise, $\mathcal{M}(e) = \mathcal{M}(s_0 \ldots s_{i-1})\, \mathcal{A}(s_{i-1}, \pi_i)\, \mathcal{S}(s_i)$.
- If e is infinite, then $\mathcal{M}(e)$ is the limit as i increases without bound of $\mathcal{M}(s_0 \pi_1 s_1 \ldots s_i)$.

We first consider safety properties. We give two sets of conditions on abstraction mappings, both of which imply that executions map to executions, with the same sequence of external actions. The first set of conditions applies when there is a single higher-level automaton immediately above. As formalized in Lemma 1, the first condition ensures that the sequences of external actions are the same, and the second and third conditions ensure that executions map to executions, and that a certain predicate is an invariant for the lower-level algorithm. A key point about this predicate is that it includes the higher-level invariant. Condition (2) is the basis step. Condition (3) is the inductive step, in which the predicate, including the high-level invariant, may be used; part (a) shows the low-level predicate is invariant, while parts (b) and (c) show executions map to executions, by ensuring that if there is no corresponding high-level action, then the high-level state is unchanged, and if there is a corresponding high-level action, then it is enabled in the previous high-level state and its effects are mirrored in the subsequent high-level state. Since executions map to executions, the high-level invariant, when composed with the state mapping, is also invariant for A.

Definition: Let A and B be automata with the same external actions. Let $\mathcal{M} = (\mathcal{S}, \mathcal{A})$ be an abstraction mapping from A to B, P be a predicate on *states*(A), and Q be a predicate true of

all reachable states of B. We say A *simulates* B *via* $\mathcal{M}$, P, *and* Q if the following hold:
(1) If s is a state of A such that $Q(\mathcal{S}(s))$ and $P(s)$ are true, and π is any action of A enabled in s, then $\mathcal{A}(s,\pi)|ext(B) = \pi|ext(A)$.
(2) If s is in $start(A)$, then $P(s)$ is true, and $\mathcal{S}(s)$ is in $start(B)$.
(3) Let (s',π,s) be a step of A such that $Q(\mathcal{S}(s'))$ and $P(s')$ are true. Then
(a) $P(s)$ is true,
(b) if $\mathcal{A}(s',\pi)$ is empty, then $\mathcal{S}(s) = \mathcal{S}(s')$, and
(c) if $\mathcal{A}(s',\pi)$ is not empty, then $\mathcal{S}(s')\ \mathcal{A}(s',\pi)\ \mathcal{S}(s)$ is an execution fragment of B. □

Lemma 1: *If A simulates B via $\mathcal{M} = (\mathcal{S},\mathcal{A})$, P and Q, then the following are true for any execution e of A.*
(1) $\mathcal{M}(e)$ is an execution of B, and $\mathcal{M}(e)|ext(B) = e|ext(A)$.
(2) $(Q \circ \mathcal{S}) \wedge P$ is true in every state of e.

Next we suppose that there are several higher-level versions, say B_1 and B_2, of automaton A, each focusing on a different task. There are situations in which it is impossible to show that A simulates B_1 without using invariants about B_2's task, and it is impossible to show that A simulates B_2 without using invariants about B_1's task. One could cast the invariants about B_2's task as predicates of A, and use the previous definition to show A simulates B_1, but this violates the spirit of the lattice. Instead, we define a notion of *simultaneously simulates*, which allows invariants about both tasks to be used in showing that A simulates B_1 and B_2. The definition differs from simply requiring A to simulate B_1 and A to simulate B_2 in one important way: steps of A only need to be reflected properly in each higher-level algorithm when *all* the higher-level invariants are true (cf. condition (3)). Lemma 2 shows that this definition preserves safety properties similar to those in Lemma 1.

Definition: Let I be an index set. Let A and B_r, $r \in I$, be automata with the same external actions. For all $r \in I$, let $\mathcal{M}_r = (\mathcal{S}_r,\mathcal{A}_r)$ be an abstraction mapping from A to B_r, and let Q_r be a predicate true of all reachable states of B_r. Let P be a predicate on $states(A)$. We say A *simultaneously simulates* $\{B_r : r \in I\}$ *via* $\{\mathcal{M}_r : r \in I\}$, P, *and* $\{Q_r : r \in I\}$ if the following hold:
(1) If s is a state of A such that $\bigwedge_{r\in I} Q_r(\mathcal{S}_r(s))$ and $P(s)$ are true, and π is any action of A enabled in s, then $\mathcal{A}_r(s,\pi)|ext(B_r) = \pi|ext(A)$ for all $r \in I$.
(2) If s is in $start(A)$, then $P(s)$ is true, and $\mathcal{S}_r(s)$ is in $start(B_r)$ for all $r \in I$.
(3) Let (s',π,s) be a step of A such that $P(s')$ and $\bigwedge_{r\in I} Q_r(\mathcal{S}_r(s'))$ are true. Then
(a) $P(s)$ is true,
(b) if $\mathcal{A}_r(s',\pi)$ is empty, then $\mathcal{S}_r(s) = \mathcal{S}_r(s')$, for all $r \in I$, and
(c) if $\mathcal{A}_r(s',\pi)$ is not empty, then $\mathcal{S}_r(s')\mathcal{A}_r(s',\pi)\mathcal{S}_r(s)$ is an execution fragment of B_r, for all $r \in I$. □

Lemma 2: *Let I be an index set. If A simultaneously simulates $\{B_r : r \in I\}$ via $\{\mathcal{M}_r : r \in I\}$, P, and $\{Q_r : r \in I\}$, where $\mathcal{M}_r = (\mathcal{S}_r,\mathcal{A}_r)$ for all $r \in I$, then the following are true of any execution e of A.*
(1) $\mathcal{M}_r(e)$ is an execution of B_r, and $e|ext(A) = \mathcal{M}_r(e)|ext(B_r)$, for all $r \in I$.
(2) $\bigwedge_{r\in I}(Q_r \circ \mathcal{S}_r) \wedge P$ is true in every state of e.

We now consider liveness properties. Given automata A and B, and a locally-controlled action φ of B, a definition of A being *equitable* for φ is given. Lemmas 3 and 4 show that this definition implies that in the execution of B obtained from a fair execution of A by either of the simulation mappings, once φ becomes enabled, it either occurs or becomes disabled. If A is equitable for all locally-controlled actions of B, then the induced execution of B is fair. Thus, fair executions map to fair executions.

Definition: Suppose $\mathcal{M} = (\mathcal{S},\mathcal{A})$ is an abstraction mapping from A to B. Let φ be a locally-controlled action of B. A is *equitable for* φ *via* $\mathcal{M}$ if in every fair execution of A, whenever a state s occurs with φ enabled in $\mathcal{S}(s)$, then subsequently either a state s' occurs with φ disabled in $\mathcal{S}(s)$, or a state-action pair (s',π) occurs with φ in $\mathcal{A}(s',\pi)$. If A is equitable for φ via $\mathcal{M}$ for every locally-controlled action φ of B, then A is *equitable for* B. □

Given locally-controlled action φ of automaton B, define an execution of B to be *fair for* φ if whenever φ becomes enabled, subsequently φ either occurs or becomes disabled.

Lemma 3: *Suppose A simulates B via $\mathcal{M}$. Let φ be a locally-controlled action of B. If A is equitable for φ via $\mathcal{M}$, then $\mathcal{M}(e)$ is fair for φ, for every fair execution e of A.*

Lemma 4: *Suppose A simultaneously simulates $\{B_r : r \in I\}$ via $\{\mathcal{M}_r : r \in I\}$. Let φ be a locally-controlled action of B_r for some r. If A is equitable for φ via $\mathcal{M}_r$, then $\mathcal{M}_r(e)$ is fair for φ, for every fair execution e of A.*

Three methods of showing that A is equitable for locally-controlled action φ of B are described. The first method, presented in Lemma 5, is to show that there is an action ρ of A that is enabled whenever φ is, and whose occurrence implies φ's occurrence. This method is useful when ρ and φ are in some sense the same action, described at the same level of abstraction.

Lemma 5: *Suppose $\mathcal{M} = (\mathcal{S}, \mathcal{A})$ is an abstraction mapping from A to B, φ is a locally-controlled action of B, and ρ is a locally-controlled action of A such that, for all reachable states s of A,*
(1) ρ is enabled in s if and only if φ is enabled in state $\mathcal{S}(s)$ of B, and
(2) if ρ is enabled in s, then φ is in $\mathcal{A}(s, \rho)$.
Then A is equitable for φ via $\mathcal{M}$.

The second method uses a definition of A being *progressive* for φ, meaning there is a set of "helping" actions of A that are guaranteed to occur, and which make progress, measured with a variant function, toward an occurrence of φ in the induced execution of B. Lemma 6 shows that progressive implies equitable. This method is useful when φ is modeled in A at a lower level of abstraction as a series of actions. (A very similar technique can be used [LPS,F] to show that progress is made toward termination of a program, considered at a single level of abstraction.) For technical reasons, we actually need helping state-action pairs — there are situations in our proof of the [GHS] algorithm when a particular helping action only makes progress if it occurs in certain states.

Definition: Suppose $\mathcal{M} = (\mathcal{S}, \mathcal{A})$ is an abstraction mapping from A to B. If φ is a locally-controlled action of B, then we say A is *progressive* for φ via $\mathcal{M}$ if there is a set Ψ of pairs (s, ψ) of states s of A and locally-controlled actions ψ of A, and a function v from *states*(A) to a well-founded set such that the following are true.

(1) For any reachable state s with φ enabled in $\mathcal{S}(s)$, some action ψ is enabled in s such that (s, ψ) is in Ψ.

(2) For any step (s', π, s) of A, where s' is reachable, φ is enabled in $\mathcal{S}(s')$ and $\mathcal{S}(s)$, and φ is not in $\mathcal{A}(s', \pi)$,
(a) $v(s) \leq v(s')$,
(b) if $(s', \pi) \in \Psi$, then $v(s) < v(s')$, and
(c) if $(s', \pi) \notin \Psi$, ψ is enabled in s', and $(s', \psi) \in \Psi$, then ψ is enabled in s and $(s, \psi) \in \Psi$. □

Lemma 6: *If A is progressive for φ via $\mathcal{M}$, then A is equitable for φ via $\mathcal{M}$.*

The third method for checking the equitable condition can be useful when various automata are arranged in a lattice, as in Figure 1. (Cf. Lemma 7.) The main idea is to show that there is some action ρ of D that is essentially the same action as φ, described at the same level of abstraction, such that C is progressive for ρ using certain helping actions, and A is equitable for all the helping actions for ρ. In essence, the argument that φ either occurs or becomes disabled, once it is enabled, is made at a high level of abstraction, and then is pulled down to where it is needed. (For convenience, we define abstraction function $\mathcal{M}$ applied to the empty sequence to be the empty sequence.)

Lemma 7: *Let A, B, C and D be automata such that $\mathcal{M}_{AB} = (\mathcal{S}_{AB}, \mathcal{A}_{AB})$ is an abstraction function from A to B, and similarly for $\mathcal{M}_{AC}$ and $\mathcal{M}_{CD}$. Let φ be a locally-controlled action of B. Suppose the following conditions are true.*
(1) $\mathcal{M}_{AC}(e)$ is an execution of C for every execution e of A.
(2) There is a locally-controlled action ρ of D such that for any reachable state s of A, if φ is enabled in $\mathcal{S}_{AB}(s)$, then ρ is enabled in $\mathcal{S}_{CD}(\mathcal{S}_{AC}(s))$.
(3) If (s', π, s) is a step of A, s' is reachable, and ρ is in $\mathcal{M}_{CD}(\mathcal{A}_{AC}(s', \pi))$, then φ is in $\mathcal{A}_{AB}(s', \pi)$.
(4) C is progressive for ρ via $\mathcal{M}_{CD}$, using the set Ψ_ρ and the function v_ρ.
(5) A is equitable for ψ via $\mathcal{M}_{AC}$, for all actions ψ of C such that $(t, \psi) \in \Psi_\rho$ for state t of C.
Then A is equitable for φ via $\mathcal{M}_{AB}$.

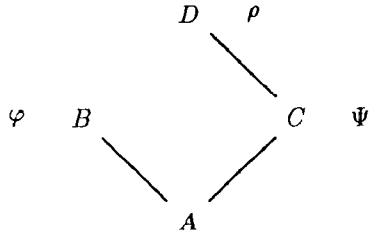

Figure 1

Theorems 8 and 9 show that our definitions of simulate, simultaneously simulate and equitable are sufficient for showing that A satisfies B.

Theorem 8: *If A simulates B via $\mathcal{M}$, P and Q and if A is equitable for B via $\mathcal{M}$, then A satisfies B.*

Theorem 9: *Let I be an index set. If A simultaneously simulates $\{B_r : r \in I\}$ via $\{\mathcal{M}_r : r \in I\}$, P and $\{Q_r : r \in I\}$, and if A is equitable for B_r via $\mathcal{M}_r$ for some $r \in I$, then A satisfies B_r.*

3. Minimum Spanning Trees

For the rest of this paper, let G be a connected undirected graph, with at least two nodes and a unique weight, chosen from a totally ordered set, associated with each edge. Nodes are $V(G)$ and edges are $E(G)$. For each edge (p,q) in $E(G)$, there are two *links* (i.e., directed edges), $\langle p,q\rangle$ and $\langle q,p\rangle$. The set of all links of G is denoted $L(G)$. It can be shown that the minimum spanning tree of G is unique; we denote it $T(G)$. Another important fact is:

Lemma 10: *If S is a subgraph of $T(G)$ and e is the minimum-weight external edge of S, then e is in $T(G)$.*

The $MST(G)$ problem is the following external schedule module (i.e., actions used to interact with the environment, and set of allowable behaviors; see Appendix A). Input actions are $\{Start(p) : p \in V(G)\}$. Output actions are $\{InTree(l), NotInTree(l) : l \in L(G)\}$. Schedules are all sequences of actions such that

- no output action occurs unless an input action occurs;
- if an input action occurs, then exactly one output action occurs for each $l \in L(G)$;
- if $InTree(\langle p,q\rangle)$ occurs, then (p,q) is in $T(G)$; and
- if $NotInTree(\langle p,q\rangle)$ occurs, then (p,q) is not in $T(G)$.

4. Proof of GHS Algorithm

In this section, we describe informally the structure of the lattice used to prove the correctness of Gallager, Humblet and Spira's minimum-spanning tree algorithm [GHS], by discussing each algorithm in the lattice and stating what relationships must be proven. The lattice is shown in Figure 2. The full proof may be found in [W]. Due to lack of space, in this paper we only present three of the automata in the lattice (see Appendix B), and only show satisfaction for one link in the chosen path (see Appendix C).

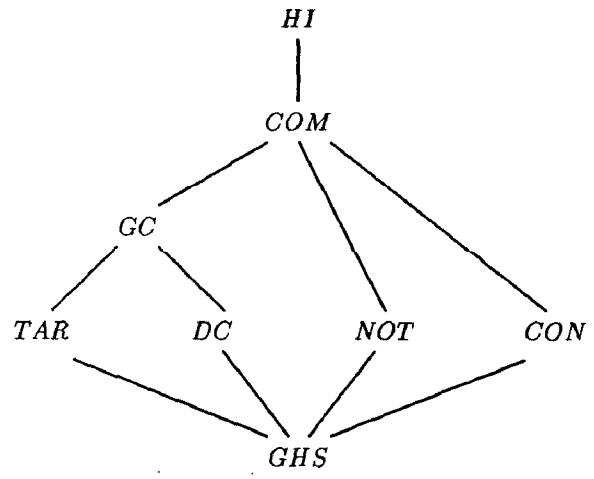

Figure 2: The Lattice

HI is a very high-level description of the algorithm, and is easily shown to solve the $MST(G)$ problem. GHS is the detailed algorithm from [GHS]. We show a path in the lattice from GHS to HI, where each automaton in the path satisfies the automaton above it. By transitivity of satisfaction, then GHS will have been shown to solve $MST(G)$.

Obviously, GHS must be shown to satisfy one of TAR, DC, NOT and CON. Showing that executions of GHS map to executions of the chosen automaton requires invariants about all four; thus, we show that GHS simultaneously simulates those four automata. To verify the invariants for the four, we show that TAR and DC (independently) simulate GC, and that NOT and CON (independently) simulate COM. Likewise, in order to show these facts, we need the invariants of GC and COM, which are obtained by showing that GC simulates COM, and that COM simulates HI. Thus, it is necessary to show safety relationships along every edge in the lattice.

The liveness relationships only need be shown along one path from GHS to HI. We decided on pragmatic grounds that it would be easiest to show that GHS is equitable for TAR. One consideration was that the output actions have exactly the same preconditions in GHS and in TAR, and thus showing GHS is equitable for those actions is trivial. Once TAR was chosen, the rest of the path was fixed.

HI: ("High") This automaton takes a totally global view of the graph. The essential feature of the state of HI is a set of subgraphs of G, initially the set of singleton nodes of G. The idea is that the subgraphs of G are connected subgraphs of the minimum spanning tree $T(G)$. Two subgraphs F and F' can combine along edge e, via the $Combine(F, F', e)$ action, if the minimum-weight external edge e of F' leads to F. $InTree(\langle p, q\rangle)$ can only occur if (p, q) is already in a subgraph, or is the minimum-weight external edge of a subgraph (i.e., is destined to be in a subgraph). By Lemma 10, these edges really are in $T(G)$. $NotInTree(\langle p, q\rangle)$ can only occur if p and q are in the same subgraph but the edge between them is not.

The obvious distributed implementation of this high-level algorithm, in which messages bearing the new subgraph identity must be broadcast throughout the new subgraph each time one is formed, has poor worst-case message complexity. GHS uses *levels* to reduce the number of messages; levels are introduced next.

COM: ("Common") This algorithm gives a good way to explain the main ideas of the [GHS] algorithm intuitively, yet precisely. The COM algorithm still takes a completely global view of the algorithm, but some intermediate steps leading to combining are identified, and the state is expanded to include extra information about the subgraphs. The COM state consists of a set of *fragments*, a data structure used throughout the rest of the lattice. Each fragment f has associated with it a subgraph of G, as well as other information: $level(f)$, $core(f)$, $minlink(f)$, and $rootchanged(f)$. Two milestones must be reached before a fragment can combine. First, the $ComputeMin(f)$ action causes the minimum-weight external link of fragment f to be identified as $minlink(f)$, and second, the $ChangeRoot(f)$ action indicates that fragment f is ready to combine, by setting the variable $rootchanged(f)$. There are two ways that fragments (and hence, their associated subgraphs) can combine. The $Merge(f, g)$ action causes two fragments, f and g, at the same level with the same minimum-weight external edge, to combine; the new fragment has level one higher than the level of f, and a new core (i.e., unique identifier), the combining edge. The $Absorb(f, g)$ action causes a fragment g to be engulfed by the fragment f at the other end of $minlink(g)$, provided f is at a higher level than g.

GC: ("Global ComputeMin") This version of the algorithm is still totally global in approach. The GC automaton expands on the process of finding the minimum-weight external link of a fragment. Each fragment f has a set $testset(f)$ of nodes that are participating in the search. A new action, $TestNode(p)$, is added, by which a node p in the *testset* atomically finds its minimum-weight external link and is removed from the *testset*, as long as the link does not lead to a lower-level fragment. $ComputeMin(f)$ can occur once $testset(f)$ is empty. After a merge, all the nodes in the new fragment are in the *testset*. When an $Absorb(f, g)$ action occurs, all the nodes formerly in g are added to $testset(f)$ if and only if the target of $minlink(g)$ is in $testset(f)$.

TAR: ("Test-Accept-Reject") TAR, as well as DC, NOT and CON, are partially global and partially local in approach. TAR expands on the method by which a node finds its local minimum-weight external link, using local variables and messages. TAR is unconcerned with how all this local information is collated to identify the fragment's global minimum-weight external link. (This problem is addressed by DC, which ignores the local protocol.)

Each link l is classified by the variable $lstatus(l)$ as branch, rejected, or unknown. Branch means the link will definitely be in the minimum spanning tree; rejected means it definitely will not be; and unknown means that the link's status is currently unknown. Initially, all the links are unknown.

The search for node p's minimum-weight external link is initiated by the action $SendTest(p)$, which causes p to identify its minimum-weight unknown link as $testlink(p)$, and to send a TEST message over its testlink together with information about the level and core (identity) of p's fragment. If the level of the recipient q's fragment is less than p's, the message is requeued at q, to be dealt with later (when q's level has increased sufficiently). Otherwise, a response is sent back. If the fragments are different, the response is an ACCEPT message, otherwise, it is a REJECT message. An optimization is that if q has already sent a TEST message over the same edge and is waiting for a response, and if p and q are in the same

fragment, then q does not respond — the TEST message that q already sent will inform p that the edge (p, q) is not external. When a REJECT message, or a TEST with the same fragment id, is received, the recipient marks that link as rejected, if it is unknown. (It is possible that the link is already marked as branch, in which case it should not be changed to rejected.)

When a *ChangeRoot*(f) occurs, *minlink*(f) is marked as branch; when an *Absorb*(f, g) occurs, the reverse link of *minlink*(g) is marked as branch. As soon as a link l is classified as branch, the *InTree*(l) output action can occur; as soon as a link l is classified as rejected, the *NotInTree*(l) output action can occur.

DC: ("Distributed ComputeMin") This automaton focuses on how the nodes of a fragment cooperate to find the minimum-weight external link of the fragment in a distributed fashion, using local variables and messages. It describes the flow of messages throughout the fragments: first a broadcast informs nodes that they should find their local minimum-weight external links, and then a convergecast reports the results back. However, the actual means by which a node finds its local minimum-weight external link are not of concern. The variable *minlink*(f) is now a derived variable, depending on variables local to each node, and the contents of message queues. There is no action *ComputeMin*(f).

The two nodes adjacent to the core send out FIND messages over the core. These messages are propagated throughout the fragment. When a node p receives a FIND message, it changes the variable *dcstatus*(p) from "unfind" to "find", relays FIND messages, and records the link from which the FIND was received as its *inbranch*(p). Then the node atomically finds its local minimum-weight external link using action *TestNode*(p) as in GC, and waits to receive REPORT(w) messages from all its "children" (the nodes to which it sent FIND). Then p takes the minimum over all the weights w reported by its children and the weight of its own local minimum-weight external link, and sends that weight to its "parent" in a REPORT message, along *inbranch*(p); the weight and the link associated with this minimum are recorded as *bestwt*(p) and *bestlink*(p), and *dcstatus*(p) is changed back to "unfind". A node p adjacent to the core waits until all its children have reported before processing any REPORT(w) message received over the core; when p processes such message with $w >$ *bestwt*(p), then the derived variable *minlink*(f) becomes defined, and is the link found by following *bestlinks* from p.

NOT: ("Notify") This automaton refines on COM by implementing the level and core of a fragment with local variables *nlevel*(p) and *nfrag*(p) for each node p in the fragment, and with NOTIFY messages. When two fragments merge, NOTIFY messages are sent over the new core, carrying the level and core of the newly created fragment. When a node p receives a NOTIFY message, it updates *nlevel*(p) and *nfrag*(p) using the information in the message. The level of a fragment is defined to be the maximum value, over all nodes p and links l in the fragment, of *nlevel*(p) and the level in a message in l. The core of a fragment is defined to be the value of *nfrag*(p), if *nlevel*(p) defines the level, otherwise it is the core information in the message defining the level.

CON: ("Connect") This automaton concentrates on what happens after *minlink*(f) is identified, until fragment f merges or is absorbed, i.e., the *ChangeRoot*(f, g), *Merge*(f, g) and *Absorb*(g, f) actions are broken down into a series of actions, involving message-passing. The variable *rootchanged*(f) is now derived. As soon as *ComputeMin*(f) occurs, the node adjacent to the core closest to *minlink*(f) sends a CHANGEROOT message on its outgoing link that leads to *minlink*(f). A chain of such messages makes its way to the source of *minlink*(f), which then sends a CONNECT(*level*(f)) message over *minlink*(f). The presence of a CONNECT message in *minlink*(f) means that *rootchanged*(f) is true. Thus, the *ChangeRoot*(f) action is only needed for fragments f consisting of a single node.

GHS: This automaton is essentially the fully distributed, original algorithm of [GHS]. (We have made some slight changes, which are discussed below.) The functions of TAR, DC, NOT and CON are united into one. All variables from those algorithms that are global (*fragments*, *minlink*, *testset*, etc.) are now derived variables, i.e., they are defined as functions of the explicit local variables and message queue contents. The messages sent in this automaton are all those sent in TAR, DC, NOT and CON, except that

NOTIFY messages are replaced by INITIATE messages, which have a parameter that is either "find" or "found", and FIND messages are replaced by INITIATE messages with the parameter equal to "find".

The bulk of the arguing done at this stage is showing that the derived variables have the proper values in the state mappings. In addition, a substantial argument is needed to show that the implementation of *level* and *core* by local variables interacts correctly with the test-accept-reject protocol. It would be ideal to do this argument in NOT, where the rest of the argument that *core* and *level* are implemented correctly is done, but reorganizing the lattice to allow this consolidation caused graver violations of modularity.

Some minor changes were made to the algorithm as presented in [GHS]. First, our version initializes all variables to convenient values. This change makes it easier to state the predicates. Second, the output actions $InTree(l)$ and $NotInTree(l)$ are added, to conform to the I/O automaton model. Third, when node p receives an INITIATE message, variables $inbranch(p)$, $bestlink(p)$ and $bestwt(p)$ are only changed if the parameter of the INITIATE message is "find". This change does not affect the performance or correctness of the algorithm. The values of these variables are not used until p subsequently receives an INITIATE-find message, at which time these variables are reset, in both the original and our version. The advantage of the change is that it greatly simplifies the state mapping from GHS to DC.

Our version of the algorithm is slightly more general than that in [GHS]. There, each node p has a single queue for incoming messages, whereas in our description, p has a separate queue of incoming messages for each of its neighbors. A node p in our algorithm could happen to process messages in the order, taken over all the neighbors, in which they arrive (modulo the requeueing), which would be consistent with the original algorithm. But p could also handle the messages in some other order (although, of course, still in order for each individual link). Thus, the set of executions of our version is a proper superset of the set of executions of the original.

A small optimization to the original algorithm was also found. (It does not affect the worst-case performance.) When a CONNECT message is received by p under circumstances that cause fragment g to be absorbed into fragment f, an INITIATE message with parameter "find" is only sent if $testlink(p) \neq nil$ in our version, instead of whenever $nstatus(p) =$ "find" as in the original. As a result of this change, if $nstatus(p) =$ "find" and $testlink(p) = nil$, p need not wait for the entire (former) fragment g to find its new minimum-weight external link before p can report to its parent, since this link can only have a larger weight than the minimum-weight external link of p already found.

Acknowledgments

We thank Yehuda Afek, Steve Garland, Michael Merritt, Liuba Shrira and members of the Theory of Distributed Systems research group at MIT for valuable discussions.

References

[A1] B. Awerbuch, "Complexity of Network Synchronization," *JACM* vol. 32, no. 4, pp. 804–823, 1985.

[A2] B. Awerbuch, "Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems," *Proc. 19th Ann. ACM Symp. on Theory of Computing,* pp. 230–240, 1987.

[AG] B. Awerbuch and R. Gallager, "Distributed BFS Algorithms," *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science,* pp. 250–256, 1985.

[AS] B. Alpern and F. Schneider, "Proving Boolean Combinations of Deterministic Properties," *Proc. 2nd Ann. Symp. on Logic in Computer Science,* pp. 131–137, 1987.

[CT] F. Chin and H. F. Ting, "An Almost Linear Time and $O(n \log n + e)$ Messages Distributed Algorithm for Minimum-Weight Spanning Trees," *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science,* pp. 257–266, 1985.

[EF] T. Elrad and N. Francez, "Decomposition of Distributed Programs into Communication-Closed Layers," *Science of Computer Programming*, vol. 2, no. 3, pp. 155–173, December 1982.

[F] N. Francez, *Fairness*, Springer-Verlag, New York, 1986, Chapter 2.

[FLS] A. Fekete, N. Lynch, L. Shrira, "A Modular Proof of Correctness for a Network Syn-

chronizer," *Proc. 2nd International Workshop on Distributed Algorithms*, 1987.

[G] E. Gafni, "Improvements in the Time Complexity of Two Message-Optimal Election Algorithms," *Proc. 4th Ann. ACM Symp. on Principles of Distributed Computing*, pp. 175–185, 1985.

[GHS] R. Gallager, P. Humblet and P. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. on Programming Languages and Systems*, vol. 5, no. 1, pp. 66–77, 1983.

[H] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[K] R. Kurshan, "Reducibility in Analysis of Coordination," *Proc. IIASA Workshop on Discrete Event Systems*, 1987.

[L] L. Lamport, "Specifying Concurrent Program Modules," *ACM Trans. on Programming Languages and Systems*, vol. 5, no. 2, pp. 190–222, April 1983.

[LM] N. Lynch and M. Merritt, "Introduction to the Theory of Nested Transactions," to appear in *Theoretical Computer Science*. (Also available as technical report MIT/LCS/TR-367, Laboratory for Computer Science, Massachusetts Institute of Technology, 1986.)

[LPS] D. Lehmann, A. Pnueli, and J. Stavi, "Impartiality, Justice and Fairness: The Ethics of Concurrent Termination," *Proc. 8th International Colloquium on Automata, Languages and Programming*, pp. 264–277, July 1981.

[LSc] L. Lamport and F. Schneider, "The 'Hoare Logic' and All That," *ACM Trans. on Programming Languages and Systems*, vol. 6, no. 2, pp. 281–296, April 1984.

[LSh] S. Lam and U. Shankar, "Protocol Verification via Projections," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 4, pp. 325–342, July 1984.

[LT] N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, pp. 137–151, 1987. (Also available as technical report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, 1987.)

[MP] Z. Manna and A. Pnueli, "Verification of Concurrent Programs: Temporal Proof Principles," in D. Kozen, editor, *Logic of Programs, Lecture Notes in Computer Science* 131, pp. 200–252, Springer-Verlag, Berlin, 1981.

[OG] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica*, vol. 6, no. 4, pp. 319–340, August 1976.

[S] E. Stark, "Foundations of a Theory of Specification for Distributed Systems," Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1984. (Available as technical report MIT/LCS/TR-342.)

[SR] F. Stomp and W. de Roever, "A Correctness Proof of a Distributed Minimum-Weight Spanning Tree Algorithm," *Proc. 7th International Conference on Distributed Computing Systems*, pp. 440–447, 1987.

[W] J. Welch, "Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms," Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1988. (To appear as MIT/LCS technical report.)

Appendix A: I/O Automata

This Appendix is a review of the aspects of the model from [LT] that are relevant to this paper.

An *input-output automaton* A is defined by the following four components. (1) There is a (possibly infinite) set of *states* with a subset of *start states*. (2) There is a set of *actions*, associated with the state transitions. The actions are divided into three classes, *input*, *output*, and *internal*. Input actions are presumed to originate in the automaton's environment; consequently the automaton must be able to react to them no matter what state it is in. Output and internal actions (or, *locally-controlled* actions) are under the local control of the automaton; internal actions model events not observable by the environment. The input and output actions are the *external* actions of A, denoted $ext(A)$. (3) The transition relation is a set of (state, action, state) triples, such that for any state s' and input action π, there is a transition (s', π, s) for some state s. (4) There is an equivalence relation $part(A)$ partitioning the output and internal actions of A. The partition is meant to reflect separate pieces of the system

being modeled by the automaton. Action π is *enabled* in state s' if there is a transition (s', π, s) for some state s; otherwise π is disabled.

An *execution* e of A is a finite or infinite sequence $s_0 \pi_1 s_1 \ldots$ of alternating states and actions such that s_0 is a start state, (s_{i-1}, π_i, s_i) is a transition of A for all i, and if e is finite then e ends with a state. The *schedule* of an execution e is the subsequence of actions appearing in e.

We often want to specify a desired behavior using a set of schedules. Thus we define an *external schedule module* S to consist of input and output actions, and a set of schedules $scheds(S)$. Each schedule of S is a finite or infinite sequence of the actions of S. Internal actions are excluded in order to focus on the behavior visible to the outside world. External schedule module S' is a *sub-schedule module* of external schedule module S if S and S' have the same actions and $scheds(S') \subseteq scheds(S)$.

An execution of a system is fair if each component is given a chance to make progress infinitely often. Of course, a process might not be able to take a step every time it is given a chance. Formally stated, execution e of automaton A is *fair* if for each class C of $part(A)$, the following two conditions hold. (1) If e is finite, then no action of C is enabled in the final state of e. (2) If e is infinite, then either actions from C appear infinitely often in e, or states in which no action of C is enabled appear infinitely often in e. Note that any finite execution of A is a prefix of some fair execution of A.

The *fair behavior* of automaton A, denoted $Fairbehs(A)$, is the external schedule module with the input and output actions of A, and with set of schedules $\{\alpha | ext(A) : \alpha$ is the schedule of a fair execution of $A\}$. $(\alpha | ext(A)$ is the subsequence of α consisting of exactly the external actions of A.) A *problem* is (specified by) an external schedule module. Automaton A *solves* the problem P if $Fairbehs(A)$ is a sub-schedule module of P, i.e., the behavior of A visible to the outside world is consistent with the behavior required in the problem specification. Automaton A *satisfies* automaton B if $Fairbehs(A)$ is a sub-schedule module of $Fairbehs(B)$.

Appendix B: Code for Automata

This Appendix contains the code for the automata HI, COM, and GHS. Each action is listed, together with its pre- and post-conditions. The preconditions specify the states in which the action is enabled. The postconditions describe the changes made to the state by the transition function.

HI: The state consists of a set FST of subgraphs of G, a Boolean variable *answered*(l) for each $l \in L(G)$, and a Boolean variable *awake*. In the start state of HI, FST is the set of single-node graphs, one for each $p \in V(G)$, every *answered*(l) is false, and *awake* is false.

INPUT ACTIONS

- *Start*(p), $p \in V(G)$
  Post: *awake* := true

OUTPUT ACTIONS

- *InTree*$(\langle p, q \rangle)$, $\langle p, q \rangle \in L(G)$
  Pre: *awake* = true
  $(p, q) \in F$ or (p, q) is the minimum-weight external edge of F, for some $F \in FST$
  answered$(\langle p, q \rangle)$ = false
  Post: *answered*$(\langle p, q \rangle)$:= true
- *NotInTree*$(\langle p, q \rangle)$, $\langle p, q \rangle \in L(G)$
  Pre: *awake* = true
  $p, q \in F$ and $(p, q) \notin F$, for some $F \in FST$
  answered$(\langle p, q \rangle)$ = false
  Post: *answered*$(\langle p, q \rangle)$:= true

INTERNAL ACTIONS

- *Combine*(F, F', e), $F, F' \in FST$, $e \in E(G)$
  Pre: *awake* = true
  $F \neq F'$
  e is an external edge of F
  e is the minimum-weight external edge of F'
  Post: $FST := FST - \{F, F'\} \cup \{F \cup F' \cup e\}$

COM: The state consists of a set *fragments*. Each element f of the set is called a *fragment*, and has the following components:

- *subtree*(f), a subgraph of G;
- *core*(f), an edge of G or *nil*;
- *level*(f), a nonnegative integer;
- *minlink*(f), a link of G or *nil*;
- *rootchanged*(f), a Boolean.

The state also contains Boolean variables, *answered*(l) one for each $l \in L(G)$, and Boolean variable *awake*. In the start state of COM, *fragments* has one element for each node in $V(G)$; for fragment f corresponding to node p, *subtree*(f) = $\{p\}$, *core*(f) = *nil*, *level*(f) = 0, *minlink*(f) is the minimum-weight link adjacent to p, and *rootchanged*(f) is false. Each *answered*(l) is false

and *awake* is false. Two fragments will be considered the same if either they have the same single-node subtree, or they have the same nonnil core.

We define the following derived variables.

- For node p, *fragment*(p) is the element f of *fragments* such that p is in *subtree*(f).
- A link $\langle p,q\rangle$ is an *external* link of p and of *fragment*(p) if *fragment*(p) $\neq$ *fragment*(q); otherwise the link is *internal*.
- If *minlink*(f) $= \langle p,q\rangle$, then *minedge*(f) is the edge (p,q).

INPUT ACTIONS

- *Start*(p), $p \in V(G)$
  Post: *awake* := true

OUTPUT ACTIONS

- *InTree*($\langle p,q\rangle$), $\langle p,q\rangle \in L(G)$
  Pre: *awake* = true
  $(p,q) \in$ *subtree*(*fragment*(p)) or
  $\langle p,q\rangle$ = *minlink*(*fragment*(p))
  answered($\langle p,q\rangle$) = false
  Post: *answered*($\langle p,q\rangle$) := true
- *NotInTree*($\langle p,q\rangle$), $\langle p,q\rangle \in L(G)$
  Pre: *awake* = true
  fragment(p) = *fragment*(q) and
  $(p,q) \notin$ *subtree*(*fragment*(p))
  answered($\langle p,q\rangle$) = false
  Post: *answered*($\langle p,q\rangle$) := true

INTERNAL ACTIONS

- *ComputeMin*(f), $f \in$ *fragments*
  Pre: *minlink*(f) = *nil*
  l is the minimum-weight external link of f
  level(f) $\leq$ *level*(*fragment*(*target*(l)))
  Post: *minlink*(f) := l
- *ChangeRoot*(f), $f \in$ *fragments*
  Pre: *awake* = true
  rootchanged(f) = false
  minlink(f) $\neq$ *nil*
  Post: *rootchanged*(f) := true
- *Merge*(f,g), $f,g \in$ *fragments*
  Pre: $f \neq g$
  rootchanged(f) = *rootchanged*(g) = true
  minedge(f) = *minedge*(g)
  Post: add a new element h to *fragments*
  subtree(h) := *subtree*(f) $\cup$ *subtree*(g)
  $\cup$ *minedge*(f)
  core(h) := *minedge*(f)
  level(h) := *level*(f) + 1
  minlink(h) := *nil*
  rootchanged(h) := false
  delete f and g from *fragments*
- *Absorb*(f,g), $f,g \in$ *fragments*
  Pre: *rootchanged*(g) = true
  level(g) < *level*(f)
  fragment(*target*(*minlink*(g))) = f
  Post: *subtree*(f) := *subtree*(f) $\cup$ *subtree*(g)
  $\cup$ *minedge*(g)
  delete g from *fragments*

GHS: This is the automaton modeling the fully distributed algorithm. The state has the following components for all $p \in V(G)$:

- *nstatus*(p), either sleeping, find, or found;
- *nfrag*(p), an edge of G or *nil*;
- *nlevel*(p), a nonnegative integer;
- *bestlink*(p), a link of G or *nil*;
- *bestwt*(p), a weight or ∞;
- *testlink*(p), a link of G or *nil*;
- *inbranch*(p), a link of G or *nil*; and
- *findcount*(p), a nonnegative integer.

The state has the following components for all $\langle p,q\rangle \in L(G)$:

- *lstatus*($\langle p,q\rangle$), either unknown, branch or rejected;
- $queue_p(\langle p,q\rangle)$, a FIFO queue of messages from p to q waiting at p to be sent;
- $queue_{pq}(\langle p,q\rangle)$, a FIFO queue of messages from p to q that are in the communication channel;
- $queue_q(\langle p,q\rangle)$, a FIFO queue of messages from p to q waiting at q to be processed; and
- *answered*($\langle p,q\rangle$), a Boolean.

The set of possible messages M is $\{\text{CONNECT}(l) : l \geq 0\} \cup \{\text{INITIATE}(l,c,st) : l \geq 0, c \in E(G), st$ is find or found$\} \cup \{\text{TEST}(l,c) : l \geq 0, c \in E(G)\} \cup \{\text{REPORT}(w) : w$ is a weight or $\infty\} \cup \{\text{ACCEPT}, \text{REJECT}, \text{CHANGEROOT}\}$.

In the start state, for all p: *nstatus*(p) = sleeping, *nfrag*(p) = *nil*, *nlevel*(p) = 0, *bestlink*(p) is arbitrary, *bestwt*(p) is arbitrary, *testlink*(p) = *nil*, *inbranch*(p) is arbitrary, *findcount*(p) = 0; for all l: *lstatus*(l) = unknown, *answered*(l) = false, and the three queues are empty.

INPUT ACTIONS

- *Start*(p), $p \in V(G)$
  Post: if *nstatus*(p) = sleeping then
  execute procedure *WakeUp*(p)

OUTPUT ACTIONS

- *InTree*(l), $l \in L(G)$
  Pre: *answered*(l) = false
  lstatus(l) = branch
  Post: *answered*(l) := true

- *NotInTree*(l), $l \in L(G)$
  Pre: *answered*(l) = false
  lstatus(l) = rejected
  Post: *answered*(l) := true

INTERNAL ACTIONS

- *ChannelSend*($\langle q,p\rangle, m$), $\langle q,p\rangle \in L(G)$, $m \in M$
  Pre: m at head of $queue_p(\langle p,q\rangle)$
  Post: dequeue($queue_p(\langle p,q\rangle)$)
  enqueue($queue_{pq}(\langle p,q\rangle)$)
- *ChannelRecv*($\langle q,p\rangle, m$), $\langle q,p\rangle \in L(G)$, $m \in M$
  Pre: m at head of $queue_{qp}(\langle q,p\rangle)$
  Post: dequeue($queue_{qp}(\langle q,p\rangle)$)
  enqueue($m, queue_p(\langle q,p\rangle)$)
- *ReceiveConnect*($\langle q,p\rangle, l$), $\langle q,p\rangle \in L(G)$
  Pre: CONNECT(l) at head of $queue_p(\langle q,p\rangle)$
  Post: dequeue($queue_p(\langle q,p\rangle)$)
  if *nstatus*(p) = sleeping then
  execute procedure *WakeUp*(p)
  if $l <$ *nlevel*(p) then [
  lstatus($\langle p,q\rangle$) := branch
  if *testlink*(p) $\neq$ *nil*, then [
  enqueue(INITIATE(*nlevel*(p), *nfrag*(p), find), $queue_p(\langle p,q\rangle)$)
  findcount(p) := *findcount*(p) + 1]
  else enqueue(INITIATE(*nlevel*(p), *nfrag*(p), found), $queue_p(\langle p,q\rangle)$)]
  else
  if *lstatus*($\langle p,q\rangle$) = unknown then
  enqueue(CONNECT(l), $queue_p(\langle q,p\rangle)$)
  else enqueue(INITIATE(*nlevel*(p) + 1, (p,q), find), $queue_p(\langle p,q\rangle)$)
- *ReceiveInitiate*($\langle q,p\rangle, l, c, st$), $\langle q,p\rangle \in L(G)$
  Pre: INITIATE(l, c, st) at head of $queue_p(\langle q,p\rangle)$
  Post: dequeue($queue_p(\langle q,p\rangle)$)
  nlevel(p) := l
  nfrag(p) := c
  nstatus(p) := st
  let $S = \{\langle p,r\rangle : lstatus(\langle p,r\rangle) = \text{branch}, r \neq q\}$
  enqueue(INITIATE(l, c, st), $queue_p(k)$)
  for all $k \in S$
  if st = find then [
  inbranch(p) := $\langle p,q\rangle$
  bestlink(p) := *nil*
  bestwt(p) := ∞
  execute procedure *Test*(p)
  findcount(p) := $|S|$]
- *ReceiveTest*($\langle q,p\rangle, l, c$), $\langle q,p\rangle \in L(G)$
  Pre: TEST(l, c) at head of $queue_p(\langle q,p\rangle)$
  Post: dequeue($queue_p(\langle q,p\rangle)$)
  if *nstatus*(p) = sleeping then
  execute procedure *WakeUp*(p)
  if $l >$ *nlevel*(p) then
  enqueue(TEST(l, c), $queue_p(\langle q,p\rangle)$)
  else
  if $c \neq$ *nfrag*(p) then
  enqueue(ACCEPT, $queue_p(\langle p,q\rangle)$)
  else [
  if *lstatus*($\langle p,q\rangle$) = unknown then
  lstatus($\langle p,q\rangle$) := rejected
  if *testlink*(p) $\neq \langle p,q\rangle$ then
  enqueue(REJECT, $queue_p(\langle p,q\rangle)$)
  else execute procedure *Test*(p)]
- *ReceiveAccept*($\langle q,p\rangle$), $\langle q,p\rangle \in L(G)$
  Pre: ACCEPT at head of $queue_p(\langle q,p\rangle)$
  Post: dequeue($queue_p(\langle q,p\rangle)$)
  testlink(p) := *nil*
  if $wt(p,q) <$ *bestwt*(p) then [
  bestlink(p) := $\langle p,q\rangle$
  bestwt(p) := $wt(p,q)$]
  execute procedure *Report*(p)
- *ReceiveReject*($\langle q,p\rangle$), $\langle q,p\rangle \in L(G)$
  Pre: REJECT at head of $queue_p(\langle q,p\rangle)$
  Post: dequeue($queue_p(\langle q,p\rangle)$)
  if *lstatus*($\langle p,q\rangle$) = unknown then
  lstatus($\langle p,q\rangle$) := rejected
  execute procedure *Test*(p)
- *ReceiveReport*($\langle q,p\rangle, w$), $\langle q,p\rangle \in L(G)$
  Pre: REPORT(w) at head of $queue_p(\langle q,p\rangle)$
  Post: dequeue($queue_p(\langle q,p\rangle)$)
  if $\langle p,q\rangle \neq$ *inbranch*(p) then [
  findcount(p) := *findcount*(p) − 1
  if $w <$ *bestwt*(p) then [
  bestwt(p) := w
  bestlink(p) := $\langle p,q\rangle$]
  execute procedure *Report*(p)]
  else
  if *nstatus*(p) = find then
  enqueue(REPORT(w), $queue_p(\langle q,p\rangle)$)
  else if $w >$ *bestwt*(p) then
  execute procedure *ChangeRoot*(p)
- *ReceiveChangeRoot*($\langle q,p\rangle$), $\langle q,p\rangle \in L(G)$
  Pre: CHANGEROOT at head of $queue_p(\langle q,p\rangle)$
  Post: dequeue($queue_p(\langle q,p\rangle)$)
  execute procedure *ChangeRoot*(p)

PROCEDURES

- *WakeUp*(p)
  let $\langle p,q\rangle$ be the minimum-weight link of p
  lstatus($\langle p,q\rangle$) := branch

nstatus(*p*) := found
enqueue(CONNECT(0), $queue_p(\langle p, q \rangle)$)

- *Test*(*p*)
  if *l*, the minimum-weight link of *p* with
  lstatus(*l*) = unknown, exists then [
  testlink(*p*) := *l*
  enqueue(TEST(*nlevel*(*p*), *nfrag*(*p*)),
  $queue_p(l)$)]
  else [
  testlink(*p*) := *nil*
  execute procedure *Report*(*p*)]
- *Report*(*p*)
  if *findcount*(*p*) = 0 and *testlink*(*p*) = *nil* then [
  nstatus(*p*) := found
  enqueue(REPORT(*bestwt*(*p*)), $queue_p$
  (*inbranch*(*p*)))]
- *ChangeRoot*(*p*)
  if *lstatus*(*bestlink*(*p*)) = branch then
  enqueue(CHANGEROOT, $queue_p$(*bestlink*(*p*)))
  else [
  enqueue(CONNECT(*nlevel*(*p*)),
  $queue_p$(*bestlink*(*p*)))
  lstatus(*bestlink*(*p*)) := branch]

Appendix C: COM Satisfies HI

In Lemma 11, we show that *COM* simulates *HI* via $\mathcal{M}$, *P*, and *Q*, and in Lemma 12 we show that *COM* is equitable for *HI* via $\mathcal{M}$, where $\mathcal{M}$, *Q* and *P* are defined below. Then Theorem 8 implies that *COM* satisfies *HI*.

Define *Q* to be the conjunction of the following predicates on *states*(*HI*).

- HI-A: Each *F* in *FST* is connected.
- HI-B: *FST* is a minimum spanning forest of *G* (i.e., a set of disjoint subgraphs of *G* that span *V*(*G*) and form a subgraph of a minimum spanning tree of *G*.)
- HI-C: If *awake* = false, then each *F* in *FST* is a singleton.

Define *P* to be the conjunction of the following predicates on states of *COM*. (*f* and *g* range over all fragments.)

- COM-A: If *minlink*(*f*) = *l*, then *l* is the minimum-weight external link of *f*, and *level*(*f*) ≤ *level* (*fragment*(*target*(*l*))).
- COM-B: If *rootchanged*(*f*) = true, then *minlink*(*f*) ≠ *nil*.
- COM-C: If $f \neq g$, then *subtree*(*f*) ≠ *subtree*(*g*).
- COM-D: If |*nodes*(*f*)| = 1, then *level*(*f*) = 0, *core*(*f*) = *nil*, *minlink*(*f*) ≠ *nil*, and *rootchanged*(*f*) = false.
- COM-E: If |*nodes*(*f*)| > 1, then *level*(*f*) > 0 and *core*(*f*) ∈ *subtree*(*f*).

Next we define the abstraction mapping $\mathcal{M} = (\mathcal{S}, \mathcal{A})$ from *COM* to *HI*. Define the function $\mathcal{S}$ from *states*(*COM*) to *states*(*HI*) as follows. For any *s* in *states*(*COM*), the values of *awake* and *answered*(*l*) (for all *l*) in $\mathcal{S}(s)$ are the same as in *s*, and the value of *FST* in $\mathcal{S}(s)$ is the multiset {*subtree*(*f*) : *f* ∈ *fragments*}.

Define the function $\mathcal{A}$ as follows. Let *s* be a state of *COM* and π an action of *COM* enabled in *s*.

- If π = *Start*(*p*), *InTree*(*l*), or *NotInTree*(*l*), then $\mathcal{A}(s, \pi) = \pi$.
- If π = *ComputeMin*(*f*) or *ChangeRoot*(*f*), then $\mathcal{A}(s, \pi)$ is empty.
- If π = *Merge*(*f*, *g*) or *Absorb*(*f*, *g*), then $\mathcal{A}(s, \pi)$ = *Combine*(*F*, *F'*, *e*), where *F* = *subtree*(*f*), *F'* = *subtree*(*g*), and *e* = *minedge*(*g*) in *s*.

The following predicates are true in every state of *COM* satisfying $(Q \circ \mathcal{S}) \wedge P$, i.e., they are deducible from *P* and *Q*. (See [W] for proofs.)

- COM-F: If *awake* = false, then |*nodes*(*f*)| = 1, *minlink*(*f*) ≠ *nil*, and *rootchanged*(*f*) = false.
- COM-G: The multiset {*subtree*(*f*) : *f* ∈ *fragments* } forms a partition of *V*(*G*), and *fragment*(*p*) is well-defined.

Lemma 11: *COM simulates HI via* $\mathcal{M}$, *P*, *Q*.

Proof: By inspection, the types of *COM*, *HI*, $\mathcal{M}$ and *P* are correct. In the full paper it is shown that *Q* is a predicate true in every reachable state of *HI*. We must check the three conditions in the definition of "simulates".

(1) Let *s* be in *start*(*COM*). Obviously, *P* is true in *s*, and $\mathcal{S}(s)$ is in *start*(*HI*).

(2) Obviously, $\mathcal{A}(s, \pi)|ext(HI) = \pi|ext(COM)$ for any state *s* of *A*.

(3) Let (s', π, s) be a step of *COM* such that *Q* is true of $\mathcal{S}(s')$ and *P* is true of *s'*. We consider each possible value of π.

i) π is Start(p), InTree(l), or NotInTree(l). $\mathcal{A}(s', \pi) = \pi$. Obviously, *P* is true in *s*, and $\mathcal{S}(s')\pi\mathcal{S}(s)$ is an execution fragment of *HI*.

ii) π is ComputeMin(f) or ChangeRoot(f). $\mathcal{A}(s', \pi)$ is empty. Obviously, $\mathcal{S}(s') = \mathcal{S}(s)$. It is straightforward to show that *P* is true in *s*.

iii) π is Merge(f,g).

(3a) It is straightforward to show *P*(*s*) is true.

(3c) $\mathcal{A}(s', \pi)$ = *Combine*(*F*, *F'*, *e*), where *F* = *subtree*(*f*) in *s'*, *F'* = *subtree*(*g*) in *s'*, and *e* =

minedge(g) in s'.
Claims about s':
1. $f \neq g$, by precondition.
2. *rootchanged*(f) = *rootchanged*(g) = true, by precondition.
3. *minedge*(f) = *minedge*(g), by precondition.
4. *awake* = true, by Claim 2 and COM-F.
5. *minedge*(f) $\neq$ *nil* and *minedge*(g) $\neq$ *nil*, by Claim 2 and COM-B
6. *minlink*(f) is an external link of f, by COM-A and Claim 5.
7. *minlink*(g) is the minimum-weight external link of g, by COM-A and Claim 5.
Claims about $\mathcal{S}(s')$: (All depend on the definition of $\mathcal{S}$.)
8. *awake* = true, by Claim 4.
9. $F \neq F'$, by Claim 1 and COM-C.
10. e is an external edge of F, by Claims 3, 6.
11. e is the minimum-weight external edge of F', by Claim 7.

By Claims 8 through 11, *Combine*(F, F', e) is enabled in $\mathcal{S}(s')$. Obviously, its effects are mirrored in $\mathcal{S}(s)$.

iv) π is Absorb(f,g).

(3a) We verify that COM-A is true in s (the rest of P is straightforward). If *minlink*(f) = *nil* in s', then the same is true in s, and COM-A is vacuously true for f. Let f' = *fragment*(*target*(*minlink*(f))).
Claims about s':
1. *level*(g) < *level*(f), by precondition.
2. *fragment*(*target*(*minlink*(g))) = f, by precondition.
3. *level*(f) $\leq$ *level*(f'), by COM-A.
4. $f' \neq g$, by Claims 1 and 3.
5. *minedge*(f) $\neq$ *minedge*(g), by Claims 2 and 4.
6. *wt*(*minedge*(f)) < *wt*(*minedge*(g)), by Claims 2 and 5 and COM-A.
7. *minlink*(g) is the minimum-weight external link of g, by COM-A.
8. *wt*(*minedge*(f)) < *wt*(e'), where e' is any external edge of g, by Claims 6 and 7.

In going from s' to s, *minlink*(f) is unchanged and *subtree*(f) changes by incorporating the old *subtree*(g). Thus, Claim 8 implies that in s, *minlink*(f) is the minimum-weight external link of f. The only fragment whose *level* changes in going from s' to s is g (since g disappears). Thus, Claim 3 implies that in s, *level*(f) $\leq$ *level*(f').

(3c) $\mathcal{A}(s', \pi)$ = *Combine*(F, F', e), where F = *subtree*(f) in s', F' = *subtree*(g) in s', and e = *minedge*(g) in s'. It is straightforward to show that *Combine*(F, F', e) is enabled in $\mathcal{S}(s')$ and that its effects are mirrored in $\mathcal{S}(s)$. □

We now show that *COM* is equitable for *HI* via $\mathcal{M}$. A significant argument is required to show that once the *HI* action *Combine*(F, F', e) becomes enabled, it eventually occurs or becomes disabled. The main idea is to show that as long as there exist two distinct subgraphs in *HI*, progress is made in *COM*; the heart of the argument is showing that some fragment at the lowest level can always take a step in *COM*. This requires a global argument that considers all the fragments. A similar argument is required to show that once the *HI* action *InTree*(l) becomes enabled, it eventually occurs, i.e., if l is the minimum-weight external link of a subgraph in *HI*, then eventually l becomes the *minlink* of some fragment *COM*.

Lemma 12: *COM is equitable for HI via* $\mathcal{M}$.
Proof: By Lemmas 1 and 11, $(Q \circ \mathcal{S}) \wedge P$ is true in every reachable state of P. Thus, in the sequel we will use the HI and COM predicates.

For each locally-controlled action φ of *HI*, we must show that *COM* is equitable for φ via $\mathcal{M}$.

i) φ is Start(p) or NotInTree(l). Lemma 5 gives the result.

ii) φ is Combine(F,F',e). We show *COM* is progressive for φ via $\mathcal{M}$; Lemma 6 implies *COM* is equitable for φ via $\mathcal{M}$.

Let Ψ_φ be the set of all pairs (s, ψ) of reachable states s of *COM* and internal actions ψ of *COM* enabled in s. For reachable state s, let $v_\varphi(s) = (x, y, z)$, where x is the number of fragments in s, y is the number of fragments f with *rootchanged*(f) = false in s, and z is the number of fragments f with *minlink*(f) = *nil* in s. (Two triples are compared lexicographically.)

(1) Let s be a reachable state of *COM* such that φ is enabled in $\mathcal{S}(s)$. We now show that some action ψ is enabled in s with $(s, \psi) \in \Psi_\varphi$.
Claims:
1. *awake* = true in $\mathcal{S}(s)$, by precondition of φ.
2. $F \neq F'$ in $\mathcal{S}(s)$, by precondition of φ.
3. *awake* = true in s, by Claim 1 and definition of $\mathcal{S}$.
4. There exist f and g in *fragments* such that *subtree*(f) = F and *subtree*(g) = F' in s, by Claim 2 and definition of $\mathcal{S}$.

5. $f \neq g$ in s, by Claims 2 and 4.

Let $l = \min\{level(f') : f' \in fragments\}$ in s. (By Claim 4, *fragments* is not empty, so l is defined.) Let $L = \{f' \in fragments : level(f') = l\}$.

Case 1: There exists $f' \in L$ with $minlink(f') = nil$. Let $\psi = ComputeMin(f')$. We now show ψ is enabled in s. By Claim 5, the minimum-weight external link $\langle p, q\rangle$ of f' exists. By choice of l, $level(f') \leq level(fragment(q))$. Obviously $(s, \psi) \in \Psi_\varphi$.

Case 2: For all $f' \in L$, $minlink(f') \neq nil$.

Case 2.1: There exists $f' \in L$ with $rootchanged(f') =$ false. Let $\psi = ChangeRoot(f')$. ψ is enabled in s by Claim 3 and the assumption for Case 2. Obviously $(s, \psi) \in \Psi_\varphi$.

Case 2.2: For all $f' \in L$, $rootchanged(f') =$ true.

Case 2.2.1: There exists fragment $g' \in L$ with $level(f') > l$, where $f' = fragment(target(minlink(g')))$. (By COM-G, f' is uniquely defined.) Let $\psi = Absorb(f', g')$. Obviously ψ is enabled in s, and $(s, \psi) \in \Psi_\varphi$.

Case 2.2.2: There is no fragment $g' \in L$ such that $level(f') > l$, where $f' = fragment(target(minlink(g')))$. Pick any fragment f_1 such that $level(f_1) = l$. For $i > 1$, define f_i to be $fragment(target(minlink(f_{i-1})))$.

More claims about s:

6. f_i is uniquely defined, for all $i \geq 1$. *Proof:* If $i = 1$, by definition. Suppose it is true for $i - 1 \geq 1$. Then it is true for i by COM-G, since $minlink(f_i)$ is well-defined and non-nil.

7. $minlink(f_i)$ is the minimum-weight external link of f_i, for all $i \geq 1$, by COM-A.

8. $f_i \neq f_{i-1}$, for all $i > 1$, by Claims 6 and 7 and definition of f_i.

9. If $minedge(f_i) \neq minedge(f_{i-1})$ for some $i > 1$, then f_{i+1} is not among $f_1, \ldots, f_i$, by Claims 7 and 8, and since the edge-weights are totally ordered.

10. There are only a finite number of fragments, by COM-C and the fact that $V(G)$ is finite.

By Claims 9 and 10, there is an $i > 1$ such that $minedge(f_i) = minedge(f_{i-1})$. Let $\psi = Merge(f_i, f_{i-1})$. Obviously ψ is enabled in s, and $(s, \psi) \in \Psi_\varphi$.

(2) Consider a step (s', π, s) of COM, where s' is reachable, φ is enabled in both $\mathcal{S}(s')$ and $\mathcal{S}(s)$, and φ is not in $\mathcal{A}(s', \pi)$.

(a) $v_\varphi(s) \leq v_\varphi(s')$, because there is no action of COM that increases the number of fragments; only a *Merge* action increases the number of fragments with *minlink* equal to *nil* or *rootchanged* equal to false, and it simultaneously causes the number of fragments to decrease.

(b) Suppose $(s', \pi) \in \Psi_\varphi$. Then $v_\varphi(s) < v_\varphi(s')$, since *Absorb* and *Merge* decrease the number of fragments, *ComputeMin* maintains the number of fragments and the number of fragments with *rootchanged* = false and decreases the number with *minlink* = *nil*, and *ChangeRoot* maintains the number of fragments and decreases the number with *rootchanged* = false.

(c) Suppose $(s', \pi) \notin \Psi_\varphi$, ψ is enabled in s', and $(s', \psi) \in \Psi_\varphi$. Then ψ is still enabled in s, since the only possible values of π are $Start(p)$, $InTree(l)$ and $NotInTree(l)$, none of which disables ψ. By definition, $(s, \psi) \in \Psi_\varphi$.

iii) φ is InTree($\langle$p,q$\rangle$). We show COM is progressive for φ via $\mathcal{M}$; Lemma 6 implies that COM is equitable for φ via $\mathcal{M}$.

Let Ψ_φ be the set of all pairs (s, ψ) of reachable states s of COM and actions ψ of COM enabled in s such that ψ is either an internal action or is φ. For reachable state s, let $v_\varphi(s) = v_{Combine(F,F',e)}(s)$. The argument is very similar to that for the case $\varphi = Combine(F, F', e)$. □