Chicago Journal of Theoretical Computer Science

The MIT Press

Volume 1997, Article 3

4 September 1997

ISSN 1073–0486. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493 USA; (617)253-2889; *journals-orders@mit.edu, journals-info@mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://mitpress.mit.edu/CJTCS/*
- *http://www.cs.uchicago.edu/publications/cjtcs/*
- *ftp://mitpress.mit.edu/pub/CJTCS*
- *ftp://cs.uchicago.edu/pub/publications/cjtcs*

The *Chicago Journal of Theoretical Computer Science* is abstracted or indexed in *Research Alert,*® *SciSearch,*® *Current Contents*®*/Engineering Computing & Technology*, and *CompuMath Citation Index.*®

©1997 The Massachusetts Institute of Technology. Subscribers are licensed to use journal articles in a variety of ways, limited only as required to insure fair attribution to authors and the journal, and to prohibit use in a competing commercial product. See the journal's World Wide Web site for further details. Address inquiries to the Subsidiary Rights Manager, MIT Press Journals; (617)253-2864; *journals-rights@mit.edu*.

The *Chicago Journal of Theoretical Computer Science* is a peer-reviewed scholarly journal in theoretical computer science. The journal is committed to providing a forum for significant results on theoretical aspects of all topics in computer science.

Editor in chief: Janos Simon

Consulting editors: Joseph Halpern, Stuart A. Kurtz, Raimund Seidel

Editors:	Martin Abadi	Greg Frederickson	John Mitchell
	Pankaj Agarwal	Andrew Goldberg	Ketan Mulmuley
	Eric Allender	Georg Gottlob	Gil Neiger
	Tetsuo Asano	Vassos Hadzilacos	David Peleg
	Laszló Babai	Juris Hartmanis	Andrew Pitts
	Eric Bach	Maurice Herlihy	James Royer
	Stephen Brookes	Ted Herman	Alan Selman
	Jin-Yi Cai	Stephen Homer	Nir Shavit
	Anne Condon	Neil Immerman	Eva Tardos
	Cynthia Dwork	Howard Karloff	Sam Toueg
	David Eppstein	Philip Klein	Moshe Vardi
	Ronald Fagin	Phokion Kolaitis	Jennifer Welch
	Lance Fortnow	Stephen Mahaney	Pierre Wolper
	Steven Fortune	Michael Merritt	

Managing editor: Michael J. O'Donnell

Electronic mail: *chicago-journal@cs.uchicago.edu*

This article is included in the *Special Issue on Self-Stabilization*, edited by Shlomi Dolev and Jennifer Welch. The article presents a technique for proving stabilization of algorithms for systems with a rooted tree communication graph. Proving correctness of a distributed algorithm is sometimes a subtle task. Proving correctness of self-stabilizing distributed algorithms is even a more subtle task, since self-stabilizing algorithms can be started in any possible state. Therefore, techniques like the one presented in this article may serve the algorithm designer in proving the stabilization property.

The editor for this article is Shlomi Dolev.

Self-Stabilization by Tree Correction*

Goerge Varghese Anish Arora Mohamed Gouda

4 September, 1997

Abstract

Abstract-1 We describe a simple tree-correction theorem that states that any locally checkable protocol that works on a tree can be efficiently stabilized in time proportional to the height of the tree. We show how new protocols can be designed, and how existing work can be easily understood using this theorem.

1 Introduction

1-1 A protocol is *self-stabilizing* if, when started from an arbitrary global state, it begins to execute "correctly" after finite time. Typical protocols are designed to cope with a specified set of failure modes such as packet loss and link failures. A self-stabilizing protocol handles a set of failures that subsumes most previous categories, and is robust against transient errors. Transient errors include memory corruption, as well as malfunctioning devices that send out incorrect packets. Transient errors do occur in real networks and cause systems to fail unpredictably. Thus stabilizing protocols are attractive because they offer *increased robustness* as well as *potential simplicity.* Self-stabilizing algorithms can be simpler, because they use uniform mechanisms to deal with many different kinds of failures.

1-2 Self-stabilizing protocols were introduced by Dijkstra [Dij74], and have been studied by various researchers (see [Sch93] for a survey). There have been few general techniques for self-stabilization. This paper introduces one such general technique which we call *tree correction.* We will show that,

*A preliminary version of this paper appeared in [VAG95].

whenever applicable, tree correction is a simple and effective technique for stabilization.

1.1 General Techniques for Self-Stabilization

1.1-1 We begin by reviewing two earlier general techniques for self-stabilization that are most directly relevant to tree correction. Work on general techniques for self-stabilization began with a seminal paper by Katz and Perry [KP90] which showed how to compile an arbitrary asynchronous protocol into a stabilizing equivalent. In Katz and Perry's method, there is a leader that periodically checks the network. If the leader discovers that the network is in an illegal state, the leader corrects the network by resetting it to a good state. Intuitively, *centralized* checking and correction is slow, and has high message complexity. Thus the general transformation is expensive; hence more efficient (and possibly less general) techniques are worth investigating.

1.1-2 One such direction [APV91, Var93] is to divide the network into a number of overlapping *link subsystems*. A link subsystem consists of a pair of neighboring nodes and the channels, if any, between them.[1] The idea is to replace the *global, centralized* checking of [KP90] with *local, decentralized* checking. The intent, of course, is to allow each link subsystem to be checked in parallel. This results in faster stabilization.

1.1-3 A network protocol is *locally checkable* if whenever the protocol is in a bad state, some link subsystem is also in a bad state. Thus if the protocol is in a bad state, some link subsystem will be able to detect this fact locally. Intuitively, a network protocol is *locally correctable* if the network can be corrected to a good state by each link subsystem independently correcting itself to a good state. Clearly, local correction is non trivial, because link subsystems overlap at nodes that have more than two neighbors.

1.1-4 Techniques that transform any *locally checkable and correctable* protocol into a stabilizing equivalent are given in [APV91, Var93]. The idea is simple: a leader is elected for each link subsystem. Each leader periodically checks the invariants of its link subsystem and corrects the subsystem if it finds an error. Because checking and correction proceed in parallel for each subsystem, this method yields fast stabilization times for several protocols [APV91, Var93].

1.1-5 While the centralized checking and correction of Katz and Perry [KP90]

[1]In simple shared-memory models, such as the ones considered in this paper, we consider a link subsystem just to be a pair of neighboring nodes.

can be too slow, the local correctability requirement of [APV91, Var93] can be too restrictive. There are several protocols (including some described in this paper) that are not locally correctable using the definitions of [APV91, Var93]. The tree-correction technique described in this paper provides some middle ground. We show how to efficiently stabilize protocols that are locally checkable and work over a tree topology. Thus we have removed the local correctability requirement, but added a requirement for tree topologies. In this paper, we make a case for the applicability of tree correction, and show two examples where it can be used effectively.

1.1-6 It is also important to compare tree correction to other, related work. However, these comparisons are rather technical. For this reason, we defer these more technical comparisons to Section 7. The nuances in Section 7 can be better understood after the tree-correction technique is presented.

1.2 Dijkstra's Shared-Memory Model

1.2-1 In this paper, we use Dijkstra's [Dij74] shared-memory model to model network protocols. In this model, a network protocol is modeled using a graph of finite-state machines. In a single move, a *single* node is allowed to read the state of its neighbors, compute, and then possibly change its state. In a real distributed system, such atomic communication is impossible. Typically communication has to proceed through channels. Such channels must be modeled explicitly as state machines that can store messages sent from one node to another.

1.2-2 While Dijkstra's model is not very realistic, it captures the essential features of network protocols in a simple way. This allows us to state a very simple version of our main tree-correction theorem. Second, it allows us to apply our theorem to succinctly understand earlier protocols (e.g., [Dij74], [AG90]) that were described using Dijkstra's model. Third, many protocols that work in Dijkstra's model have been easily extended to work in message-passing models.[2]

1.2-3 In summary, we use Dijkstra's model because it allows a simple exposition of tree correction; tree correction can be extended to more realistic message-passing models [Var93] with some additional complexity in the theorem statement and proof.

[2]The first two protocols in [Dij74] are extended to message-passing models in [Var93].

1.3 Paper Organization

1.3-1 The main result of the paper is a theorem (Theorem 1) that states that any locally checkable protocol on a rooted tree can be efficiently stabilized. In Section 3 we state Theorem 1, and provide a proof in Section 4. In Section 5, we show how a reset protocol [AG90] due to Arora and Gouda can also be simply understood in this framework. In Section 6, we describe how a mutual exclusion protocol, due to Dijkstra, can also be easily understood using our theorem. Finally, we compare our work in more detail with related work on local checking and correction and distributed constraint satisfaction, and state our conclusions. The appendix contains a formal summary of the timed I/O automaton model used in this paper.

2 Modeling Shared-Memory Protocols

2-1 We will use a version of the timed I/O automaton model [MMT91]. The appendix contains a formal summary of this model. It is essentially a state-machine model described using automata. The state transitions of the automata are called actions. The general model has actions called *input* and *output* actions that allow an automaton to interact with the environment and to facilitate the composition of automata. We only use the simplest type of action called an *internal* action, which changes the state of the automaton without any externally visible effects.

2-2 When the automaton runs, it produces executions. Each execution is a sequence of alternating timed states and actions. A *timed state* is a tuple consisting of a state and a time value. Timed states allow us to model the passage of time and state precisely intuitive statements such as "action a occurs within time t in all executions of the automaton." The correctness of an automaton (or more generally, the protocol that is being modeled by the automaton) is specified by giving a set of "correct executions." The automaton is said to be correct if its executions fall within the specified set. The reader who is unfamiliar with this model is referred to the appendix.

2-3 How can we map Dijkstra's model into this model? Suppose each node in Dijkstra's model is a separate automaton. Then in the I/O automaton model, it is not possible to model the simultaneous reading of the state of neighboring nodes. Our solution is to dispense with modularity, and model *the entire network as a single automaton.* All actions, such as reading the state of neighbors and computing, are *internal actions.* The asynchrony in the system, which Dijkstra modeled using a "demon," is naturally a part of the I/O automaton model. Also, we describe the correctness of Dijkstra's systems in terms of *executions* of the automaton.

2-4 The major reason for using the timed I/O automaton model is that it allows us to give precise definitions of stabilization times. However, in terms of asynchronous executions, our model behaves exactly like Dijkstra's model, and hence our results apply to Dijkstra's original model. The I/O automaton model also provides standard notation (e.g., fairness using classes) that we find helpful.

2-5 Thus we model a network as a single automaton in which a node can read the state of all its neighbors (and write its own state) in a single move using an internal action. We define this special kind of I/O automaton formally as follows:

2-6 A *shared memory network automaton* $\mathcal{N}$ for graph $G = (E, V)$ is an automaton in which:

- The state set of $\mathcal{N}$ is the cross-product of a set of node states, $S_u(\mathcal{N})$, one for each node $u \in V$. The set of start states of $\mathcal{N}$ is the cross-product of a set of start states, $I_u(\mathcal{N})$, one for each node $u \in V$. For any state s of $\mathcal{N}$, we use $s|u$ to denote s projected onto S_u. This is also read as the state of node u in global state s.

- All actions of $\mathcal{N}$ are internal actions, and are partitioned into sets, $A_u(\mathcal{N})$, one for each node $u \in V$.

- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Consider any state s' of $\mathcal{N}$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u. Then there is some transition $(s', \pi, \tilde{s}')$ of $\mathcal{N}$ such that $\tilde{s}'|v = \tilde{s}|v$ for u and all u's neighbors in G.

- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Then $s|v = \tilde{s}|v$ for all nodes $v \neq u$.

2-7 Informally, the third condition requires that the transitions of a node $u \in V$ only depend on the state of node u and the states of the neighbors of u in G. The fourth condition requires that the effect of a transition assigned to node $u \in V$ can only be to change the state of u.

2-8 A shared-memory *tree automaton* is a shared-memory network automaton where G is a rooted tree. Thus for any node i in a tree automaton, we assume there is a value *parent*(i) that points to the parent of node i in the tree. There is also a unique root-node r that has *parent*$(r) = nil$. For our purposes, it is convenient to model the *parent* values as part of the code at each node. More generally, the parent pointers could be variables that are set by a stabilizing spanning-tree protocol as shown in [AG90]. We often use the phrase "tree automaton" to mean a "shared-memory tree automaton," and the phrase "network automaton" to mean a "shared-memory network automaton."

2-9 In general, shared-memory automata have start states (see first condition in definition). However, to model stabilization, we use a special kind of automaton called an *unrestricted automaton*. We say that an I/O automaton A is an *unrestricted automaton* if all possible states in the state set of A are also start states of A.

2-10 Intuitively, an unrestricted automaton models a system that can start in an arbitrary state. Thus to show stabilization results, we will show that the executions of some unrestricted automaton A will eventually begin to "look like" the executions of some other correct automaton B (B need not be unrestricted).

3 Tree Correction for Shared-Memory Systems

3-1 We start with some definitions. A *closed predicate* of an automaton A is a predicate L such that for any transition $(s, \pi, \tilde{s})$ of A, if $s \in L$, then $\tilde{s} \in L$.

3-2 A *link subsystem* of a tree automaton is an ordered pair (u, v), such that u and v are neighbors in the tree. To distinguish states of the entire automaton from the states of its subsystems, we sometimes use the word *global state* to denote a state of the entire automaton. For any global state s of a network automaton, we define $(s|u, s|v)$ to be the state of the (u, v) link subsystem. Thus the state of the (v, u) link subsystem in global state s is $(s|v, s|u)$.

3-3 A *local predicate* $L_{u,v}$ of a tree automaton is a subset of the states of a (u, v) link subsystem. A *link predicate set* $\mathcal{L}$ for a tree automaton is a set that contains exactly one local predicate $L_{u,v}$ for every link subsystem in the tree, and which satisfies the following conditions:

- Symmetry: For each pair of neighbors u and v, if $(a, b) \in L_{u,v}$, then $(b, a) \in L_{v,u}$ (i.e., while a link predicate set has two link predicates for each pair of neighbors, these two predicates are identical except for the order in which the states are written down).

- Non-triviality: For all link subsystems (u, v) in the tree, there is at least one global state s such that $(s|u, s|v) \in L_{u,v}$.
- Closed Intersection: Define the intersection predicate $L' = \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.$\}$. Then L' is a closed predicate.

3-4 These are not very restrictive conditions. One can always rewrite any link predicate set to satisfy symmetry. In the sequel, we will design an automaton that will stabilize to states satisfying the intersection predicate. Thus if the two link predicates $L_{u,v}$ and $L_{v,u}$ were unequal (except for the order of states), we could always rewrite them both as a new link predicate that contains the intersection of the states in the two original predicates, and still have the same result. Next, if some local predicate is the empty set, then it is impossible to find any global state that satisfies the intersection predicate L'. Finally, requiring that the intersection predicate be closed is essential to stabilization; if L' were not closed, the protocol might stabilize to some state satisfying L' and then transition to some state that does not satisfy L'. This would make our definition of stabilization meaningless. Note that we do not require that each of the local predicates $L_{u,v}$ be closed, a much stronger condition required in [APV91].

3-5 A tree automaton is *locally checkable* for predicate L if there is some link predicate set $\mathcal{L} = \{L_{u,v}\}$ such that: $L \supseteq \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.$\}$ In other words, the global state of the automaton satisfies L if every link subsystem (u, v) satisfies $L_{u,v}$.

3-6 Recall that an execution of an automaton is an alternating sequence of timed states and actions satisfying the transition rules and timing conditions specified by the automaton definition. Refer to the appendix for a formal definition. Since we use timed states in this definition, we can talk about some state s starting at time t after the time of the first state in the execution. With this machinery, we are ready to define stabilization.

3-7 We say that automaton A stabilizes to the executions of automaton B in time t if for every execution α of A there is some suffix of execution α, whose first state starts no less than t time units after the first state of α, that is an execution of B. Intuitively, this says that after time t, every execution of A begins to "look like" an execution of B.

3-8 Recall the definition of an unrestricted automaton as an automaton whose set of start states is identical to its set of states (i.e., any state can be a start state). We also use $A|L$ to denote the automaton that is identical to automaton A except that the start states of $A|L$ are the states in set L. For any rooted tree T, we let *height*(T) denote the maximum length of a path between the root and a leaf in T. We can now state a simple theorem.

Theorem 1 (Tree-Correction in Shared Memory Systems) *Consider any tree automaton $\mathcal{T}$ for tree T that is locally checkable for predicate L. Then there exists an unrestricted tree automaton $\mathcal{T}^+$ for T such that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}|L$ in time proportional to height(T).*

3-9 Thus after a time proportional to the height of the tree, any execution of the new automaton $\mathcal{T}^+$ will "look like" an execution of $\mathcal{T}$ that starts with a state in which L holds.

4 Proof of Tree-Correction Theorem

4-1 We start by giving an overview of the proof. We then show how to automatically construct the stabilizing automaton $\mathcal{T}^+$ from the original automaton $\mathcal{T}$, and show that $\mathcal{T}^+$ satisfies the requirements of the theorem.

4.1 Proof Overview

4.1-1 The essence of the construction is to augment the original automaton with periodic correction actions that are added to the action set of each node other than the root. The correction action is simple: node u reads the state $s|v$ of its parent and checks whether $(s|u, s|v) \in L_{u,v}$ (i.e., u checks whether the link subsystem state satisfies the link predicate for link (u, v)).

4.1-2 Let us informally say that (u, v) is in a *bad* state in global state s if $(s|u, s|v) \notin L_{u,v}$. Let us informally call a global state s a *bad* state if some link (u, v) is in a bad state. In other words, a *good* state is a state that satisfies the intersection predicate $L' = \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree$\}$. The goal of the correction actions is to move toward a good global state from a bad global state by correcting links that are in bad states.

4.1-3 However, to make this work, we need to ensure two conditions. First, we need to ensure that a node can always find some local state to correct itself into after reading its parent's state. Second, we need to ensure that once a link (u, v) gets into a good state and all links "above" the link in the tree are also in good states, then (u, v) remains in a good state. If these two conditions are met, we can argue that links directly connected to the root will eventually get into good states and stay that way. Once this is true, we can argue that links connected to nodes that are distance 1 from the root eventually get into good states and stay that way. We can apply this argument inductively to argue that links connected to nodes that are distance i from the root eventually get into good states and stay that way.

4.1-4 As a result, the automaton will eventually get into a good global state regardless of what state it starts in, and hence it is a stabilizing automaton. Notice that in the formal description, a "good" state is a state satisfying L' (and not L). But since L' is a subset of L, this is sufficient for our purposes.[3]

4.1-5 It only remains to satisfy the two conditions alluded to earlier. The first condition may not be satisfiable if the state set of a node contains states that never occur in L'. For example, consider the two-node system (u, v) where $L_{u,v} = \{(1, 1)\}$ and the state set of node $v = \{1, 3\}$. If node v is in state 3 and node v is the parent of u, then node u cannot find any state compatible with $L_{u,v}$ that it can correct itself to.

4.1-6 The solution to this problem is to banish useless states (that never occur in L') from the state of set of each node in the augmented automaton. Thus in the example above, we redefine the state set of v to be $v = \{1\}$, ruling out the "useless" state 3. We call this step-normalization. Notice that normalization is *not* a way of adding start states to our new automaton (the theorem requires the constructed automaton to allow all possible states to be start states!), but a way of refining the state space of the new automaton.[4]

4.1-7 The second condition we need to ensure is that once a link (u, v) gets into a good state and all links "above" the link in the tree are also in good

[3] In practice, this is a spurious generality, because we might as well have defined our goal state L to be equal to L', since that is all the theorem guarantees.

[4] The reader should convince himself or herself that state-space refinement cannot be used to trivialize stabilization problems. Consider doing token passing between two distributed processes. Even after state-space refinement, each process must have at least one flag to indicate when it has the token. The two flags result in four possible global states, of which two are "good states" and two (i.e., two tokens or no token) are "bad" states. On the other hand, if the two processes were on a single computer, state-space refinement could trivially solve the stabilization problem by using a single two-valued *turn* variable! The bottom line is that state-space refinement is often useful, but does not by itself solve stabilization problems for distributed protocols.

states, then (u, v) remains in a good state. We solve this by adding extra guards to the normal (i.e., other than correction) actions at each node u of the automaton such that these actions are not taken unless all links adjacent to node u are in good states. We can then prove that any normal action taken under these circumstances at node u cannot affect the "goodness" of the states of links adjacent to u.

4.1-8 Intuitively, this is because we can always construct a good global state $s \in L'$ in which u has the same local view. If the normal action violates the "goodness" of any link adjacent to node u, then it will do so for some valid transition from global state s as well, because of the locality properties of the automaton. But this contradicts the assumption L' is a closed predicate. The formal proof below (see Lemma 2 and Lemma 4) follows this intuition.

4.2 Main Construction

4.2-1 We now show how to construct the stabilizing automaton $\mathcal{T}^+$ from $\mathcal{T}$. Assume that $\mathcal{T}$ is *locally checkable* for predicate L using link predicate set $\mathcal{L} = \{L_{u,v}\}$. Recall that we defined the intersection predicate (i.e., the set of global states that satisfy all local predicates). Thus we defined $L' = \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree$\}$. Clearly $L \supseteq L'$. Also because of the nontriviality of the link predicate set, L' is not the empty set. To construct $\mathcal{T}^+$ from $\mathcal{T}$ we do the following:

- First *normalize* all node states in $\mathcal{T}$. Intuitively, we remove all states in the state set of a node u that are not part of a global state that satisfies L'. Thus $S_u(\mathcal{T}^+) := \{s|u : s \in L'\}$. The state set of $\mathcal{T}^+$ is just the cross-product of the normalized state sets of all nodes. This rules out useless node states that never occur in global states that satisfy all local predicates.

- Retain all the actions of $\mathcal{T}$, but add an extra precondition (i.e., an extra guard) to each action $a_u \in A_u$ of $\mathcal{T}$ as shown in Figure 1. This extra guard ensures that a normal action of $\mathcal{T}$ is not taken at node u unless all links adjacent to u are in "good states." All actions of $\mathcal{T}$ remain in the same classes in $\mathcal{T}^+$. (Recall that in the timed I/O automaton model, timing guarantees for internal actions are expressed by grouping these actions into classes. Each class c has an associated time bound t_c; if some action in class c is enabled for t_c time, then some action in class c must occur in t_c time.)

The state of $\mathcal{T}^+$ is identical to $\mathcal{T}$ except that the state set of each node is normalized to $\{s|u : s \in L'\}$

Modified Action $a_u, a_u \in A_u$ (*modification of action a_u in $\mathcal{T}$*)
Preconditions:
Exactly as in a_u, except for the additional condition:
For all neighbors v of u: $(s|u, s|v) \in L_{u,v}$
Effects:
Exactly as in a_u

Correct$_u$ (*extra correction action for all nodes except the root*)
Preconditions: $(parent(u) = v)$ and $((s|u, s|v) \notin L_{u,v})$
Effects:
Let a be any state in $S_u(\mathcal{T}^+)$ such that $(a, s|v) \in L_{u,v}$
Change the state of node u to a

Each Correct$_u$ action is in a separate class with upper bound t_{node}

Figure 1: Augmenting $\mathcal{T}$ to create $\mathcal{T}^+$

- Add an extra correction action, Correct$_u$, for every node u in the tree that is not the root. Correct$_u$ is also described in Figure 1. This extra action "corrects" the link between node u and its parent if this link is not in a "good" state. Each Correct$_u$ action is put in a separate class with upper bound equal to t_{node}; t_{node} is just a parameter that expresses an upper bound on the time for any node to perform a correction action.

4.3 Proof of Validity of Construction

4.3-1 We outline a proof of the theorem by a series of lemmas. The first thing a reader needs to be convinced of is that the construction in Figure 1 is realizable. The careful reader will notice that we made two assumptions. First, in the Correct$_u$ action, we assumed that for any link subsystem (u, v) of $\mathcal{T}^+$ and any state b of node v there is some a such that $(a, b) \in L_{u,v}$. Second, we assumed that when a modified action a_u is taken at node u, the resulting state of node u has not been removed as part of the normalization step.

4.3-2 We will begin with a lemma showing that the first assumption is a safe one. Later, we show that the second assumption is also safe.

Lemma 1 *For any link subsystem (u, v) of $\mathcal{T}^+$ and for any state b of node v there is some a such that $(a, b) \in L_{u,v}$.*

Proof of Lemma 1 We know that for any state b of v there is some state $s \in L'$ such that $s|v = b$. This follows because all node states have been normalized and because L' is not empty. Then we choose $a = s|u$.

Proof of Lemma 1 □

4.3-3 The next lemma shows a *local extensibility* property. It says that if any node u and its neighbors have node states such that the links between u and its neighbors are in good states, then this set of node states can be extended to form a good global state.

Lemma 2 *Consider a node u and some global state s of $\mathcal{T}^+$ such that for all neighbors v of u, $(s|u, s|v) \in L_{u,v}$. Then there is some global state $s' \in L'$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u.*

Proof of Lemma 2 We create a global state s' by assigning node states to each node in the tree such that for every link subsystem (u, v), the state of the subsystem is in $L_{u,v}$. Start by assigning node state $s|u$ to u and $s|v$ to all neighbors v of u. At every stage of the iteration, we label a node x that has not been assigned a state and is a neighbor of a node y that has been assigned a state. But, by Lemma 1, we can do this such that the state of the subsystem containing x and y is in $L_{x,y}$. Eventually we label all nodes in the tree and the resulting global state is in L'. Once again, this is because for every link subsystem (u, v), the state of the (u, v) subsystem is in $L_{u,v}$.

Proof of Lemma 2 □

4.3-4 The labeling procedure used in the lemma depends crucially on the fact that the topology is a tree.

4.3-5 To prove the main theorem, we use an execution convergence lemma first stated and proved in [Var93]. To state the theorem, we need the following definition. Note that the definition below uses what are called *predicate sets* as opposed to the link predicate sets used so far; the only difference is that a predicate set contains sets of global states, as opposed to a link predicate set, which contains sets of states of link subsystems.

4.3-6 We say that an I/O automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t if:

1. $\mathcal{L} = \{L_i, i \in I\}$ of sets of states of A, where $(I, <)$ is a finite, partially ordered index set. We let *height*$(\mathcal{L})$ denote the maximum length chain in the partial order.

2. $\cap_{i \in I} L_i = L$.

3. For all $i \in I$ and for all steps $(s, \pi, \tilde{s})$ of A, if s belongs to $\cap_{j \leq i} L_j$, then $\tilde{s}$ belongs to L_i.

4. For every $i \in I$, every execution α of A, and every state s in α, the following is true. Suppose that either $s \in \cap_{j<i} L_j$ or there is no $L_j < L_i$. Then there is some state $\tilde{s} \in L_i$ that occurs within time t of s in α.

4.3-7 The first condition says there is a partial order on the predicates in $\mathcal{L}$. The second says that L becomes "true" when all the predicates in $\mathcal{L}$ become true. The third is a stability condition. It says that any transition of A leaves a predicate L_i true if all the predicates less than or equal to L_i are true in the previous state. Finally the last item is a liveness condition. It says that if all the predicates *strictly* less than L_i are true in a state, then within time t after this state, L_i will become true.

4.3-8 We define *height*(L_i), the height of a predicate $L_i \in \mathcal{L}$, to be the maximum length of a chain that ends with L_i in the partial order. The value of *height*$(\mathcal{L})$ is, of course, the maximum height of any predicate $L_i \in \mathcal{L}$. By the liveness condition, within time t all predicates with height 1 become true; these predicates stay true for the rest of the execution because of the third stability condition. In general, we can prove by and stay true. This leads to a simple but useful lemma:

Lemma 3 (Execution Convergence) *Suppose that I/O automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t. Then, A stabilizes to the executions of $A|L$ in time height$(\mathcal{L}) \cdot t$.*

Proof of Lemma 3 By induction on $h, 0 \leq h \leq \mathit{height}(\mathcal{L})$, in the following inductive hypothesis.

Inductive Hypothesis: There is some state s that occurs within time $h \cdot t$ of the start of α such that $s \in L_i$ for all $L_i \in \mathcal{L}$ with $\mathit{height}(L_i) \leq h$.

4.3-9 The inductive hypothesis implies that there is some state s that occurs within time $h \cdot \mathit{height}(\mathcal{L})$ of the start of α and such that $s \in L$. The lemma follows from this fact. It also uses the fact that any suffix of an execution that starts with a timed start state is also a valid execution (recall that we have no lower bounds on the time between actions).

4.3-10 Base Case, $h = 0$: Follows trivially since there is no $L_i \in \mathcal{L}$ with $\mathit{height}(L_i) = 0$.

4.3-11 Inductive Step, $0 \leq h \leq \mathit{height}(\mathcal{L}) - 1$: Assume it is true for h. Then there is some state s_i that occurs within time $h \cdot t$ of the start of α and such that for all $L_i \in \mathcal{L}$ with $\mathit{height}(L_i) \leq h$, $s_i \in L_i$. Consider any $L_j \in \mathcal{L}$ with $\mathit{height}(L_j) = h + 1$. By the fourth condition in the definition of being stabilized using a predicate set, we know there must be some state $s_{f(j)} \in L_j$ that occurs within time t of s_i in α. Let $k = Max\{f(j) : \mathit{height}(L_j) = h+1\}$. Then, from the third and fourth conditions in the definition of being stabilized using a predicate set, we see that s_k occurs within time $(h+1) \cdot t$ of the start of α and such that for all $L_i \in \mathcal{L}$ with $\mathit{height}(L_i) \leq h + 1$, $s_i \in L_i$. (The third condition ensures that predicates L_j and other established predicates remain true.)

Proof of Lemma 3 □

4.3-12 However, to apply the previous lemma, we have to work with predicates of $\mathcal{T}^+$ (i.e., sets of states of $\mathcal{T}^+$) and not local predicates of $\mathcal{T}^+$ (i.e., sets of states of link subsystems of $\mathcal{T}^+$). This is just a technicality that we deal with as follows. For each link subsystem (u, v), we define the predicate $L'_{u,v} = \{s : (s|u, s|v) \in L_{u,v}\}$. Clearly, $L' = \cap L'_{u,v}$.

4.3-13 Next, consider some u, v, w such that $v = \mathit{parent}(u)$ and $w = \mathit{parent}(v)$. We assume that $v \neq \mathit{nil}$. (But v may be the root in which case w is *nil*.) The next lemma states an important stability property. It states that if $L'_{u,v}$ holds in some global state s of $\mathcal{T}^+$, it will remain true in any successor state of s if either:

- v is the root, or
- $L'_{v,w}$ is also true in s.

Lemma 4 *Consider some u, v, w such that $v = parent(u) \neq nil$ and $w = parent(v)$. Suppose there is some global state s of $\mathcal{T}^+$ such that $s \in L'_{u,v}$ and $(w \neq nil) \rightarrow s \in L'_{v,w}$. Then for any transition $(s, \pi, \tilde{s})$, $\tilde{s} \in L'_{u,v}$.*

Proof of Lemma 4-1 **Proof of Lemma 4** It suffices to consider all possible actions π that can be taken at either u or v in state s. It is easy to see that we don't have to consider correction actions because, by assumption, neither the $\textsc{Correct}_u$ or the $\textsc{Correct}_v$ action is enabled in state s.

Proof of Lemma 4-2 Consider a modified action a_u of $\mathcal{T}^+$ that is taken at node u. Suppose action a_u occurs in state s and results in a state $\tilde{s}$. By the preconditions of modified action a_u, for all children x of u, $(s|x, s|u) \in L_{x,u}$. But in that case by Lemma 2 there is some other global state $s' \in L'$ such that: $s'|u = s|u$, $s'|v = s'|v$, and $s'|x = s|x$ for all children x of u. Thus by the third property of a network automaton, the action a_u is also enabled in s' and, if taken in s', will result in some state, say $\tilde{s}'$. But since L' is closed, $\tilde{s}' \in L'$ and hence $(\tilde{s}'|u, \tilde{s}'|v) \in L_{u,v}$. But by the third property of a network automaton, $\tilde{s}|u = \tilde{s}'|u$ and $\tilde{s}|v = \tilde{s}'|v$. Thus $\tilde{s} \in L'_{u,v}$. The case of a modified action at v is similar.

Proof of Lemma 4 □

4.3-14 The previous lemma also shows that our second assumption is safe. If a modified action is taken at a node u, resulting in state $\tilde{s}$, then $\tilde{s} \in L'_{u,v}$ for some v. Thus by Lemma 2 there is some other state $\tilde{s}' \in L'$ such that $\tilde{s}'|u = \tilde{s}|u$. Thus $\tilde{s}|u$ cannot have been removed as part of the normalization step.

4.3-15 The next lemma states an obvious liveness property. If $L'_{u,v}$ does not hold in some global state of $\mathcal{T}^+$, then after at most t_{node} time units, we will eventually reach some global state $\tilde{s}$ in which $L'_{u,v}$ holds. Clearly this is guaranteed by the correction actions (either $\textsc{Correct}_u$ or $\textsc{Correct}_v$, depending on whether u is the child of v or vice versa) and by the timing guarantees.

Lemma 5 *For any (u, v) link subsystem, any execution α of $\mathcal{T}^+$, and any state s_i of α, if $s_i \notin L'_{u,v}$, then there is some later state $s_j \in L'_{u,v}$ that occurs within t_{node} time units of s_i.*

Proof of Lemma 5 Suppose not. Then either $\textsc{Correct}_u$ (if u is the child of v) or $\textsc{Correct}_v$ (if v is the child of u) is continuously enabled for t_{node}

time units after s_i. But then, by the timing guarantees, either $\textsc{Correct}_u$ or $\textsc{Correct}_v$ must occur within t_{node} time after s_i, resulting in a state in which $L'_{u,v}$ (and, of course, $L'_{v,u}$) holds.

Proof of Lemma 5 □

4.3-16 We now return to the proof of the main theorem. First we define a natural partial order on the predicates $L'_{u,v}$. For any link subsystem (u, v), define the *child node* of the subsystem to be u if $parent(u) = v$ and v otherwise. Define the ordering $<$ such that $L'_{u,v} < L'_{w,x}$ iff the child node of the (u, v) subsystem is an ancestor (in the tree T) of the child node of the (w, x) subsystem.

4.3-17 Using this partial order and Lemmas Lemma 4 and Lemma 5, we can apply Lemma 3 to show that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}^+|L'$ in time $height(T) \cdot t_{node}$. But any execution α of $\mathcal{T}^+|L'$ is also an execution of $\mathcal{T}|L$. This follows from three observations. First, since L' is closed for $\mathcal{T}$, L' is closed for $\mathcal{T}^+$. Second, if L' holds in all states of an execution α of $\mathcal{T}^+|L'$, then no correction actions can occur in α. Third, any execution of $\mathcal{T}|L'$ is also an execution of $\mathcal{T}|L$ because $L \supseteq L'$. Thus we conclude that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}|L$ in time $height(T) \cdot t_{node}$.

4.3-18 This theorem can be used as the basis of a design technique. We start by designing a tree automaton T that is locally checkable for some L. Next we use the construction in the theorem to convert T into $\mathcal{T}^+$. $\mathcal{T}^+$ stabilizes to the executions of $T|L$ even when started from an arbitrary state. The theorem also provides the worst-case time to stabilization, which is an important benefit.

4.3-19 Weakening the Fairness Requirement In the previous construction, we assigned each $\textsc{Correct}_u$ action to a separate class. Actually the theorem only requires a property we call *eventual correction*: if a $\textsc{Correct}_u$ action is continuously enabled, then a $\textsc{Correct}_u$ action occurs within bounded time. This property can be established quite easily for the protocols in [Dij74] and [AG90], allowing all the actions in the entire automaton to be placed in a single class!

5 A Reset Protocol on a Tree

5-1 Before describing the reset protocol due to Arora and Gouda [AG90], we first describe the network reset problem. Recall that we have a collection of nodes that communicate by reading the state of their neighbors. The interconnection topology is described by an arbitrary graph. Assume that we are given some application protocol that is being executed by the nodes. We wish to superimpose a reset protocol over this application such that when the reset protocol is executed, the application protocol is "reset" to some "legal" global state. A "legal" global state is allowed to be any global state that is reachable by the application protocol after correct initialization. The problem is called *distributed reset*, because reset requests may arrive at any node.

5-2 A simple and elegant network-reset protocol is due to Finn [Fin79]. In this protocol each node i running the application protocol has a session number. When the reset protocol is not running, the session numbers at every node are the same. When a node receives a reset request, it resets the local state of the application (to some prespecified initial state), and increments its session number by 1. When a node sees that a neighbor has a higher session number, it changes its session number to the higher number and resets the application. Finally, the application protocol is modified so that a node cannot make a move until its session number is the same as that of its neighbors. This check prevents older instances of the application protocol from "communicating" with newer instances of the protocol. This protocol is shown to be correct [Fin79] if all the session numbers are initially zero and the session numbers are allowed to grow without bound.[5]

5-3 We rule out the use of unbounded session numbers as unrealistic. Also, in a stabilizing setting, having a "large enough" size for a session number does not work. This is because the reset protocol can be initialized with all session numbers at their maximum values. Thus, we are motivated to search for a reset protocol that uses bounded session numbers. Suppose we could design a reset protocol with unbounded numbers in which *the difference between the session numbers at any two nodes is at most one in any state.* Suppose also that for any pair of neighboring nodes u and v that compare session numbers, the session number of one of the nodes (say u) is always no less than the session number of the other node. Then, since the session numbers are only used for comparisons, it suffices to replace the session numbers by a single bit that we call $sbit_i$. This is the first idea in Arora and Gouda's reset protocol [AG90].

[5] Finn also considered the problem of using bounded sequence numbers. His solution was shown to be incorrect by Humblett and Soloway, who proposed a fix in [SH87]. However, neither paper addresses the problem of designing a self-stabilizing reset protocol.

5-4 To realize this idea, we cannot allow a node to increment its session number as soon as it gets a reset request. Otherwise, multiple reset requests at the same node would cause the difference in session numbers to grow without bound. Thus nodes must coordinate before they increment session numbers.

5-5 In Arora and Gouda's reset protocol [AG90], the coordination is done over a rooted tree. Arora and Gouda first show how to build a rooted tree in a stabilizing fashion. In what follows, we assume that the tree has already been built. Thus every node i has a pointer called $parent(i)$ that points to its parent in the tree, and the parent of the root is a special value *nil*.

5-6 Given a tree, an immediate idea is to funnel all reset requests to the root. On receipt of a request, the root could send reset grants down the tree. A node could increment its session number on receiving a grant. Unfortunately, this does not work either, because a node A in the tree may send a reset request and receive a grant before some other node B in the tree receives a grant. After getting its first grant, A may send another request and receive a second grant before B gets its first grant. Assuming that the session numbers are unbounded, the difference in the session numbers of A and B can grow without bound.

5-7 Instead, the reset task is broken into three phases. In the first phase, a node sends a reset request up the tree toward the root. In the second phase, the root sends a reset wave down the tree. In the third phase, the root waits until the reset wave has reached every node in the tree before starting a new reset phase. This ensures that after the system stabilizes, the use of three phases will guarantee that a single bit $sbit_i$ is sufficient to distinguish instances of the application protocol.

5-8 The three phases are implemented by a mode variable $mode_i$ at each node i. The mode at node i has one of three possible values: *init*, *reset*, and *normal*. All nodes are in the *normal* mode when no reset is in progress. To initiate a reset, a node i sets $mode_i$ to *init* (this can be done only if both i and its parent are in *normal* mode). A reset request is propagated upward by the action PROPAGATE_REQUEST$_i$, which sets the mode of the parent to *init* when the mode of the child is *init*. A reset wave is begun by the root by the action START_RESET which sets the mode of the root to *reset*. The reset wave propagates downward by PROPAGATE_RESET$_i$, which sets the mode of a child to *reset* if the mode of the parent is *reset*. When a node changes its mode to *reset*, it flips its session number bit, and resets the application protocol. Finally, the completion wave is propagated by the action PROPAGATE_COMPLETION$_i$, which sets a node's mode to *normal* when *all* the node's children have *normal* mode.

5-9 The I/O automaton code for this implementation is shown in Figures 2 and 3. Notice that besides the actions we have already described, there is a CORRECT$_i$ action in Figure 3. This action was used in an earlier version [AG90] to ensure that the reset protocol was stabilizing.

5-10 Informally, the reset protocol is stabilizing if after bounded time, any reset requests will cause the application protocol to be properly reset. The correction action in Figure 3 [AG90] ensures stabilization in a very ingenious way. However, the proof of stabilization is somewhat difficult, and not as intuitive as one might like. The reader is referred to [AG90] for details. Instead, we will use local checking and tree correction to describe another correction procedure that is very intuitive. As a result, the proof of stabilization becomes transparent.

5-11 We start by writing down the "good" states of the reset system in terms of link predicates $L_{i,j}$. We say that the system is in a good state if for all neighboring nodes i and j, the predicate $L_{i,j}$ holds, where $L_{i,j}$ is the conjunction of the two predicates:

- If $(parent(i) = j)$ and $(mode_j \neq reset)$, then $(mode_i \neq reset)$ and $(sbit_i = sbit_j)$.
- If $(parent(i) = j)$ and $(mode_j = reset)$, then either:
  - $(mode_i \neq reset)$ and $(sbit_i \neq sbit_j)$, or
  - $(sbit_i = sbit_j)$.

5-12 The predicates can be understood intuitively as describing states that occur when the reset system is working correctly. The first predicate says that if the parent's mode is not *reset*, then the child's mode is not *reset* and the two session bits are the same. This is true when the system is working correctly, because of two reasons: first, the child enters *reset* mode only when its parent is in that mode, and the parent does not leave *reset* mode until the child has left *reset* mode; second, if the parent changes its session bit, the parent also goes into *reset* mode, and the child only changes its session bit when the parent's mode is *reset*.

5-13 The second predicate describes the correct states during the second and third phases of the reset until the instant that the completion wave reaches j. It says that if the parent's mode is *reset*, then there are two possibilities.

The state of the system consists of two variables for every process in the tree:
$mode_i \in \{init, normal, reset\}$
$sbit_i$, a bit

PROPAGATE_REQUEST$_i$ (*internal action to propagate a reset request upward *)
Preconditions:
$mode_i = normal$
$i = parent(j)$ and $mode_j = init$
Effects: $mode_i := init$

START_RESET$_i$ (*internal action at root to start a reset wave*)
Preconditions: $mode_i = init$ and $parent(i) = nil$
Effects:
$mode_i := reset$; (*also reset application state at this node*)
$sbit_i :=\sim sbit_i$; (*flip bit*)

PROPAGATE_RESET$_i$ (*internal action to propagate reset downwards*)
Preconditions:
$mode_i \neq reset$
$j = parent(i)$ and $mode_j = reset$ and $sbit_j \neq sbit_i$
Effects:
$mode_i := reset$; (*also reset application state at this node*)
$sbit_i :=\sim sbit_i$; (*flip bit*)

PROPAGATE_COMPLETION$_i$ (*internal action to propagate completion wave upwards *)
Preconditions:
$mode_i = reset$
For all children j of i: $mode_j = normal$ and $sbit_i = sbit_j$
Effects: $mode_i := normal$

Every action is in a separate class with upper bound equal to
t_{node}

Figure 2: Normal Actions at node i in Arora and Gouda's Reset Protocol[AG90]

CORRECT$_i$ (*extra internal action for correction at node i*)
  Preconditions:
    $j = parent(i) \neq nil$
    $(mode_j = mode_i)$ and $(sbit_i \neq sbit_j)$
  Effects: $sbit_i := sbit_j$

Every action is in a separate class with upper bound equal to t_{node}

Figure 3: Original correction action in Arora and Gouda's reset protocol [AG90]

If the child has not "noticed" that the parent's state is *reset*, then the child's bit is not equal to the parent's bit. (This follows because when the parent changes its mode to *reset*, the parent also changes its bit; and just before such an action the second predicate assures us that the two bits are the same.) On the other hand, if the child has noticed that the parent's state is *reset*, then the two bits are the same. (This follows because when the child notices that the parent's mode is *reset*, the child sets its bit equal to the parent's bit and does not change its bit until the parent changes its mode.)

5-14 Suppose that in some state s these link predicates hold for all links in the tree. Then [AG90] shows that the system will execute reset requests correctly in any state starting with s. This is not very hard to believe, but it means that all we have to do is add correction actions so that all link predicates will become true in bounded time. But this can easily be done using the transformation underlying the tree correction theorem that we just stated. An even simpler correction strategy is described below.

5-15 The tree topology once again suggests a simple strategy. We remove the old action CORRECT$_i$ in Figure 3 and add a new action CORRECT_CHILD$_i$ as shown in Figure 4. Basically, CORRECT_CHILD$_i$ checks whether the link predicate on the link between i and its parent is true. If not, i changes its state such that $L_{i,j}$ becomes true. Notice that CORRECT_CHILD$_i$ leaves the state of i's parent unchanged. Suppose j is the parent of i and k is the parent of j. Then CORRECT_CHILD$_i$ will leave $L_{j,k}$ true if $L_{j,k}$ was true in

CORRECT_CHILD$_i$ (*modified correction action at nodes*)
  Preconditions:
    $j = parent(i) \neq nil$
    $L_{i,j}$ does not hold
  Effects:
    $sbit_i := sbit_j$
    $mode_i := mode_j$

All actions are in a separate class with upper bound t_{node}

Figure 4: Modified correction action for Arora and Gouda's reset protocol. All other actions are as in Figure 2.

the previous state.

5-16 Thus we have an important stability property: correcting a link does not affect the correctness of links above it in the tree. Using this, we can apply Theorem 1 to show that all links will be in good states (and so the system is in a good state) in time proportional to the height of the tree.

6 Dijkstra's Mutual Exclusion Protocol

6-1 In this section, we show how to rederive an existing stabilizing protocol using our theorem. The advantage of our presentation is that it provides a simple and transparent proof. We reconsider the second example in [Dij74]. This protocol is essentially a token-passing protocol on a line of nodes with process indices ranging from 0 to $n-1$. Imagine that the line is drawn vertically, so that process 0 is at the bottom of the line (and hence is called "bottom") and process $n-1$ is at the top of the line (and called "top"). This is shown in Figure 5. The down neighbor of process i is process $i-1$, and the up neighbor is process $i+1$. Process $n-1$ and process 0 are not connected.

6-3 Dijkstra observed that it is impossible (without randomization) to solve mutual exclusion in a stabilizing fashion if all processes have identical code. To break the symmetry, he made the code for the "top" and "bottom" pro-

Top (Process n-1)

0 up = false
0 up = false
0 up = false —— *Token*
1 up = true
1 up = true
. .
. .
. .
. .
1 up = true

Bottom (Process 0)

Figure 5: Dijktra's protocol for token passing on a line

cesses different from the code for the others.

6-4 Dijkstra's second example is modeled by the automaton $D2$, shown in Figure 6. Each process i has a Boolean variable up_i and a bit x_i. Roughly, up_i is a pointer at node i that points in the direction of the token, and x_i is a bit that is used to implement token passing. Figure 5 shows a state of this protocol when it is working correctly. First, there can be at most two consecutive nodes whose *up* pointers differ in value, and the token is at one of these two nodes. If the two bits at the two nodes are different (as in the figure), then the token is at the upper node; otherwise the token is at the lower node.

6-5 For the present, assume that all processes start with $x_i = 0$. Also, initially assume that $up_i = \mathit{false}$ for all processes other than process 0. We do so only so that the reader can first understand the correct executions of this protocol after it has stabilized (or after it has been correctly initialized). We will remove the need for such initialization when we describe the final stabilizing automaton.

6-6 A process i is said to have the token when any action at process i is enabled. As usual, the system is correct when there is at most one token in the system. Now, it is easy to see that in the initial state, only MOVE_UP_0

The state of the system consists of a Boolean variable
up_i, and a bit x_i, one for every process in the line.
We will assume that $up_0 = \mathit{true}$ and $up_{n-1} = \mathit{false}$ by definition

In the initial state $x_i = 0$ for $i = 0 \dots n-1$ and $up_i = \mathit{false}$ for $i = 1 \dots n-1$

MOVE_UP$_0$ (*action for the bottom process only to move token up*)
Precondition: $x_0 = x_1$ and $up_1 = \mathit{false}$
Effect: $x_0 :=\sim x_0$

MOVE_DOWN$_{n-1}$ (*action for top process only to move token down*)
Precondition: $x_{n-2} \neq x_{n-1}$
Effects:
$x_{n-1} := x_{n-2}$

MOVE_UP$_i, 1 \leq i \leq n-2$ (*action for other processes to move token up*)
Precondition: $x_i \neq x_{i-1}$
Effects:
$x_i := x_{i-1}$
$up_i := \mathit{true}$ (*point upward in direction token was passed*)

MOVE_DOWN$_i, 1 \leq i \leq n-2$ (*action for other processes to move token down*)
Precondition: $x_i = x_{i+1}$ and $up_i = \mathit{true}$ and $up_{i+1} = \mathit{false}$
Effect: $up_i := \mathit{false}$ (*point downward in direction token was passed*)

All actions are in a single class with upper bound t_{node}

Figure 6: Automaton $D2$: a version of Dijkstra's second example with initial states. The protocol does token passing on a line using nodes with at most four states.

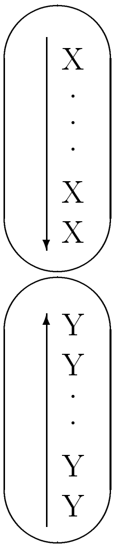

Figure 7: In the good states for Dijktra's second example, the line can be partitioned into two bands with the token at the boundary.

is enabled. Once node 0 makes a move, then Move_Up_1 is enabled, followed by Move_Up_2, and so on, as the "token" travels up the line. Finally, the token reaches node $n-1$, and we reach a state s in which $x_i = x_{i+1}$ for $i = 0 \ldots n-3$ and $x_{n-1} \neq x_{n-2}$. Also in state s, $up_i = true$ for $i = 0 \ldots n-2$ and $up_{n-1} = false$. Thus in state s, Move_Down_{n-1} is enabled, and the token begins to move down the line by executing Move_Down_{n-2}, followed by Move_Down_{n-3}, and so on, until we reach the initial state again. Then the cycle continues. Thus in correct executions, the "token" is passed up and down the line.

6-7 In the good states for Dijktra's second example, the line can be partitioned into two bands, as shown in Figure 7. All bit and pointer values within a band are equal. (If a node within a band has $up_i = false$, we sketch it as a pointer that points downward). All nodes within the upper band point downward and all nodes within the lower band point upward. The token is at the boundary between the two bands. If the bit value X of the upper band is equal to the bit value Y of the lower band, the token is moving downward; if the two bit values are unequal, the token is moving downward.

6-8 We describe these "good states" of $D2$ (that occur in correct executions) in terms of local predicates. In the shared-memory model, a local predicate is any predicate that only refers to the state variables of a pair of neighbors. We see that if two neighboring nodes have the same pointer value, then their bits are equal; also, if a node is pointing upward, then so is its lower neighbor. Thus in a good state of $D2$, two properties are true for any process i other than 0:

- If $up_{i-1} = up_i$, then $x_{i-1} = x_i$.
- If $up_i = true$, then $up_{i-1} = true$.

6-9 First, we prove that if these two local predicates hold for all $i = 1 \ldots n-1$, then there is exactly one action enabled. Intuitively, since $up_{n-1} = false$ and $up_0 = true$, we can start with process $n-1$ and go down the line until we find a pair of nodes i and $i-1$ such that $up_i = false$ and $up_{i-1} = true$. Consider the first such pair. Then the second predicate guarantees that there is exactly one such pair. The first predicate then guarantees that all nodes $j < i-1$ have $x_j = x_{i-1}$, and all nodes $k > i$ have $x_k = x_i$. Thus only one action is enabled. If $x_i = x_{i-1}$ and $i-1 \neq 0$, then only MOVE_DOWN_{i-1} is enabled. If $x_i = x_{i-1}$ and $i-1 = 0$, then only MOVE_UP_0 is enabled. If $x_i \neq x_{i-1}$ and $i \neq n-1$, then only MOVE_UP_i is enabled. If $x_i \neq x_{i-1}$ and $i = n-1$, then only MOVE_DOWN_{n-1} is enabled.

6-10 A similar argument shows that if there is exactly one action enabled, then both local predicates hold for all $i = 1 \ldots n-1$.

6-11 Let L be the predicate of $D2$ consisting of the states of $D2$ in which exactly one action is enabled. It is easy to see that $D2$ is a tree automaton that is locally checkable for L. Then we can use Theorem 1 to convert $D2$ into a new automaton $D2^+$, which stabilizes to executions where there is exactly one token in each state. The correction actions we add are once again different from the original actions in [Dij74]. However, the correction actions we add (and consequently, the proofs) are much more transparent than the original version.

7 Detailed Discussion of Related Work

7-1 In this section, we discuss some subtle differences between the work in this paper and closely related works.

7-2 The original paper on local checking and correction [APV91] proved a *local correction theorem*: any locally checkable protocol that met certain other conditions (referred to as *local correctability*) can be stabilized. However, that theorem *does not* imply our tree-correction theorem. In order to apply the local correction theorem, one must exhibit a correction function with the appropriate properties; this is not needed in tree correction. Also, the *explicit* correction action used in local correction depends only on the previous state of a node, whereas the *implicit* correction action used in the proof of tree correction depends on the state of a node's parent as well as the previous state of the node. Finally, the definition of local checkability used in [APV91] is stronger than ours. The previous definitions require that each local predicate be a closed predicate. This is a strong assumption that we do not require.

7-3 Both local correction and tree correction are useful techniques that apply to different problems. For example, local correction has been used to provide stabilizing solutions to many problems on general graphs (e.g., synchronizers [Var93] and end-to-end protocols [APV91]) for which tree correction is inapplicable. On the other hand, local correction does not seem applicable to the reset protocol described in Section 5. This is because the correction action (Figure 4) depends on the state of both the parent and the child. There are still other problems (e.g., mutual exclusion on a tree [Var93]) for which both techniques are applicable.

7-4 There has also been significant work in the Artificial Intelligence literature on *distributed constraint satisfaction.* A constraint can be considered to be a local predicate between two neighboring nodes, and the goal is to find a solution that meets all constraints. A seminal paper[6] by Collin, Dechter, and Katz [CDK91] considers the feasibility of self-stabilizing, distributed constraint satisfaction, and shows that the problem is impossible in general graphs without some form of symmetry breaking. However, the authors also show that the problem is solvable in tree networks. Their basic procedure is similar to ours: they use what they call an *arc consistency step* (similar to our normalization step), and then each node chooses a value in its domain that is consistent with its parent's value.

7-5 However, their tree protocol is limited to finding *static* solutions to constraint satisfaction. Thus constraint satisfaction can be used to solve a static problem such as coloring, but not a *dynamic* problem such as mutual exclusion or network reset. More precisely, constraint satisfaction seeks solutions that converge to a *fixed point*; our formulation seeks solutions that converge to a *closed predicate.* Even after convergence to a closed predicate, protocol

[6] We are grateful to one of the referees for pointing out this reference.

actions continue to occur.

7-6 To allow this, we added an extra modification beyond the normalization and local correction modifications needed by [CDK91]: we modified the original protocol actions to add an extra guard to check whether all local predicates are satisfied before the action is executed. This modification is crucial to ensure stability of the local predicates (Lemma 4), which in turn prevents oscillatory behavior (in which predicates are corrected for and then become falsified by protocol actions). The proof of stability requires the notion of local extensibility (Lemma 2). None of these notions are required for the result in [CDK91].

7-7 Thus our theorem does not follow from the result in [CDK91]. However, the result in [CDK91] applies to uniform tree networks—i.e., without symmetry breaking in the form of parent pointers, as we have used. This is done by a protocol that finds tree centers and directs each node's parent pointer toward its closest center [KPBG94]. We conjecture that their method could be applied to make our theorem applicable to uniform tree networks. Also, [CDK91] has a number of other results on general networks, including a self-stabilizing procedure to find a feasible solution using distributed backtracking.

7-8 Finally, [AGV94] also expresses local checkability conditions using *constraint graphs*. The formulation in [AGV94] applies to more general protocols than the constraint satisfaction protocols of [CDK91]. However, none of the theorems in [AGV94] imply our tree-correction theorem.

8 Conclusions

8-1 It may seem obvious that any locally checkable protocol on a tree can be stabilized; however, it is a different matter to state and prove a precise result that embodies this intuition. The proof requires some subtle notions such as normalizing each automaton, the notion of local extensibility, and the need to defend against unexpected transitions.

8-2 Much of the initial work in self-stabilization was done in the context of Dijkstra's shared-memory model of networks. Later, the work on local checking and correction was introduced [APV91] in a message-passing model. A contribution of this paper is to show that existing work in the shared-memory model can be understood crisply in terms of local checking and correction. Protocols that appeared to be somewhat *ad hoc* are shown to have a common underlying principle.

8-3 As argued at the beginning of this paper, we believe that message-passing models are more useful and realistic. The definitions of network automata, local predicates, local checkability, local correctability, and link subsystems that we used in this paper are specific to shared-memory systems. The main theorem in this paper states that any locally checkable protocol that uses a tree topology can be efficiently stabilized. As the reader might expect, there is a corresponding tree correction theorem for message-passing systems. This theorem is described in [Var93]. The theorem requires a lot more notation to state. While the proof is similar, the reasoning is more tedious; thus, we believe that the theorem stated in this paper conveys the essential ideas in a much simpler fashion.

A Formal Summary of the I/O Automaton Model

A-1 In this paper, we use the following model, which is a special case of the timed I/O automaton model in [MMT91]. However, our terminology is slightly different from that of [MMT91].

A-2 An *I/O automaton* A consists of five components:

- A finite set of actions, $actions(A)$, that is partitioned into three sets called the set of *input, output, and internal* actions. The union of the set of input actions and the set of output actions is called the set of *external* actions. The union of the set of output and internal actions is called the set of *locally controlled* actions.
- A finite set of states, called $states(A)$.
- A nonempty set of start states, $start(A) \subseteq states(A)$.
- A transition relation $R(A) \subseteq states(A) \times actions(A) \times states(A)$, with the property that for every state s and input action a, there is a transition $(s, a, \tilde{s}) \in R(A)$.
- An equivalence relation, $part(A)$, partitioning the set of locally controlled actions into equivalence classes, such that for each class c in $part(A)$ we have a positive real upper bound t_c. (Intuitively, t_c denotes an upper bound on the time to perform some action in class c.)

A-3 An action a is said to be *enabled* in state s of automaton A if there exist some $\tilde{s} \in states(A)$ such that $(s, a, \tilde{s}) \in R(A)$. An action a is *disabled* in state s of automaton A if it is not enabled in that state. Since one action may occur multiple times in a sequence, we often use the word *event* to denote a particular occurrence of an action in a sequence.

A-4 To model the passage of time, we use a time sequence. A *time sequence* $t_0, t_1, t_2, \ldots$ is a nondecreasing sequence of non-negative real numbers; also the numbers grow without bound if the sequence is infinite. A *timed element* is a tuple (x, t) where t is a non-negative real and x is an element drawn from an arbitrary domain. A *timed state* for automaton A is a timed element (s, t) where s is a state of A. A *timed action* for automaton A is a timed element (a, t) where a is an action of A.

A-5 Let $X = (x_0, t_0), (x_1, t_1), \ldots$ be a sequence of timed elements. We will also use $x_j.time$ (which is read as the time associated with element x_j) to denote t_j.

A-6 We say that element x_j occurs within time t of element x_i if $j > i$ and $x_j.time \leq x_i.time + t$. We will use $X.start$ (which is read as the start time of X) to denote t_0.

Definition A.1 *An* execution α *of automaton A is an alternating sequence of timed states and actions of A of the form* $(s_0, t_0), (a_1, t_1), (s_1, t_1), (a_2, t_2), (s_2, t_2), \ldots$ *such that the following conditions hold:*

1. $s_0 \in start$ *and* $(s_i, a_{i+1}, s_{i+1}) \in R$ *for all* $i \geq 0$.

2. *The sequence can either be finite or infinite, but if finite it must end with a timed state.*

3. *The sequence* $t_0, t_1, t_2, \ldots$ *is a time sequence.*

4. *If any action in any class c is enabled in any state s_i of α, then within time $s_i.time + t_c$, either some action in c occurs or some state s_j occurs in which every action in c is disabled.*

A-7 Notice that the time assigned to any event a_i in α (i.e., $a_i.time$) is equal to the time assigned to the next state (i.e., $s_i.time$). Notice also that we have ruled out the possibility of so-called "Zeno executions," in which the execution is infinite but time stays within some bound.

Acknowledgement of support: George Varghese's work is supported in part by an ONR Young Investigator Award and NSF Grant NCR-9405444.

References

[AG90] Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science, volume 472 of Lecture Notes in Computer Science*, pages 316–331, Berlin, 1990. Springer-Verlag.

[AGV94] Anish Arora, Mohamed G. Gouda, and George Varghese. Distributed constraint satisfaction. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, February 1994.

[APV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, October 1991.

[CDK91] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of the 12th IJCAI*, August 1991.

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[Fin79] Steven G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Transactions on Communications*, COM-27(6):840–845, June 1979.

[KP90] Shmuel Katz and Kenneth Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, ACM*, Quebec City, Canada, August 1990.

[KPBG94] M. Karaata, S. Pemmaraju, S. Bruell, and S. Ghosh. Self-stabilizing algorithms for finding centers and medians of trees. In *PODC 94*, August 1994.

[MMT91] M. Merritt, F. Modugno, and M. R. Tuttle. Time constrained automata. In *CONCUR 91*, pages 408–423, 1991.

[Sch93] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25, March 1993.

[SH87] Stuart R. Soloway and Pierre A. Humblet. On distributed network protocols for changing topologies. Technical Report LIDS-P-1564, May 1987. Also have version from May 1986.

[VAG95] George Varghese, Anish Arora, and Mohamed G. Gouda. Self-stabilization by tree correction. In *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, Las Vegas, May 1995.

[Var93] George Varghese. Self-stabilization by local checking and correction. Phd thesis, also technical report mit/lcs/tr-583, Massachussetts Institute of Technology, October 1993.