Self-Stabilization by Tree Correction

George Varghese*and Anish Arora†and Mohamed Gouda‡

May 11, 1995

Abstract

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

In Dijkstra's [Dij74] 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. Also, in message passing models, the channel state machine is essentially fixed (with actions to send and deliver packets) but the node state machines can be arbitrarily specified by the protocol designer. However, in Dijkstra's model *all* state machines are node state machines and can be arbitrarily specified by the protocol designer.

While Dijkstra's original model is not very realistic, it is probably the simplest model of an asynchronous distributed system. This simple model provided an ideal vehicle for *introducing* [Dij74] the concept of stabilization without undue complexity. We will use this simple model to describe a simple technique for self-stabilization that we call *tree correction*. We use this to show that existing work in [Dij74] and [AG90] can be understood very succinctly using the framework of local checking and tree correction. We will also state (but not prove) that this theorem can be extended to message passing systems.

The main result of the paper is a theorem (Theorem 3.1) that states that any locally checkable protocol on a rooted tree can be efficiently stabilized. In Section 3 we obtain Theorem 3.1. In Section 4, we show briefly how a reset protocol [AG90] due to Arora and Gouda can also be simply understood in this framework. The tree correction theorem can be generalized to message passing systems [Var93] but is most simply stated and proved in the simple shared memory model. Finally we state our conclusions and

*Washington University in St. Louis
†Ohio State University
‡University of Texas, Austin

compare our work with related work on local checking and correction [APV91b] and distributed constraint satisfaction [AGV94, CDK91].

2 Modeling Shared Memory Protocols

We will use a version of the timed I/O Automaton model [MMT91]. 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. The solution we use 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 our model. Also, we will describe the correctness of Dijkstra's systems in terms of *executions* of the automaton.

The major reason for using the timed I/O automaton model is that it allows us to model time, and hence to provide precise definitions of stabilization time. 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.

Thus we model a network as a single automaton in which a node can read and write the state of its neighbors in a single move using an internal action. Formally:

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

- The state of $\mathcal{N}$ is the cross-product of a set of node states, $S_u(\mathcal{N})$, 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 $v \neq u$.

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 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.

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 being 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 will 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".

3 Tree Correction for Shared Memory Systems

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

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 will 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 $(s|v, s|u)$.

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 predicate for every link subsystem in the tree and which satisfies the following symmetry condition: 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.) We will also assume that every link predicate set is *non-trivial* in that there is at least one global state s such that $(s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.

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 $L_{u,v}$.

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.

For any automaton A we define $U(A)$ (which can be read as the unrestricted version of A) to be the automaton that is identical to A except that any state of A can be a start state of $U(A)$. Conversely, we use $A|L$ to denote the automaton that is identical to 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 3.1 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).*

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. To prove this theorem we first describe how to construct $\mathcal{T}^+$ from $\mathcal{T}$ and then show that $\mathcal{T}^+$ satisfies the requirements of the theorem.

Assume that $\mathcal{T}$ is *locally checkable* for predicate L using link predicate set $\mathcal{L} = \{L_{u,v}\}$. We start by defining the set of global states that satisfy all local predicates. Let $L' = \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.$\}$. Clearly $L \supseteq L'$. Also because of the non-triviality of the link predicate set, L' is not the empty set. To construct $\mathcal{T}^+$ from $\mathcal{T}$ we do the following:

[1] This is often called a stable predicate. We avoid this phrase because of potential confusion with stabilization.

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_n

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

- We 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. Intuitively, this rules out useless node states that never occur in global states that satisfy all local predicates.

- We retain all the actions of $\mathcal{T}$ but we 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. Intuitively, 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 IOA model, timing guarantees for internal actions are expressed by grouping these actions into classes. Each class c has an associated time bound t_c; intuitively, if some action is class c is enabled for t_c time, then some action in class c must occur in t_c time.)

- We 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. Intuitively, 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_n.

We outline a proof of the theorem by a series of lemmas. The first thing a careful reader needs to be convinced about is that the code in Figure 1 is realizable. The careful reader will have noticed 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.

We will begin with a lemma showing that the first assumption is a safe one. We show that the second assumption is safe later.

Lemma 3.2 *For any link subsystem (u, v) of $\mathcal{T}^+$ and for any state a of node u there is some b such that $(a, b) \in L_{u,v}$.*

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

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 3.3 *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: 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 will 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 3.2, 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 $L_{u,v}$. The labeling procedure depends crucially on the fact that the topology is a tree. □

To prove the theorem, we will use an Execution Convergence Theorem stated and proved in [Var93]. To state the theorem we need the following definition.

We say that an 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. $\cap_{i \in I} L_i \subseteq L$.
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 $i \in I$ and 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.

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.

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 induction that within time $i \cdot t$ all predicates with height i become and stay true. This leads to a simple but useful theorem:

Theorem 3.4 Execution Convergence: *Suppose that 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 $height(\mathcal{L}) \cdot t$.*

However, to apply that theorem we have to work with predicates of $\mathcal{T}^+$ (i.e., sets of states of $\mathcal{T}^+$) and not link 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, $L' = \cap L'_{u,v}$.

Next consider some u, v, w such that $v = parent(u)$ and $w = parent(v)$. We assume that $v \neq nil$. (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 3.5 *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: 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 CORRECT_u or the CORRECT_v action is enabled in state s.

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 action a_u, for all children x of u, $(s|x, s|u) \in L_{x,u}$. But in that case by Lemma 3.3 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. □

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 3.3 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.

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_n 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 CORRECT_u or CORRECT_v depending on whether u is the child of v or vice versa) and by the timing guarantees.

Lemma 3.6 *For any (u, v) link subsystem and any 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_n time units of s_i.*

Proof: Suppose not. Then either CORRECT_u (if u is the child of v) or CORRECT_v (if v is the child of u) is continuously enabled for t_n time units after s_i. But then by the timing guarantees, either CORRECT_u or CORRECT_v must occur within t_n time after s_i, resulting in a state in which $L'_{u,v}$ (and, of course, $L'_{v,u}$) holds. □

We now return to the proof of the 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.

Using this partial order and Lemmas 3.5 and 3.6, we can show by induction that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}^+|L'$ in time $height(T) \cdot t_n$. 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 $height(T) \cdot t_n$.

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.

Weakening the Fairness Requirement In the previous construction, we assigned each CORRECT_u action to a separate class. Actually the theorem only requires a property we call *eventual correction*: if a CORRECT_u action is continuously enabled, then a 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!

4 A Reset Protocol on a Tree

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.

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.[2]

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 value. 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].

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 will cause the difference in session numbers to grow without bound. Thus nodes must coordinate before they increment session numbers.

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 will 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*.

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. Nodes could increment their 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

[2] 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 [HS91]. However, neither paper addresses the problem of designing a self-stabilizing reset protocol.

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.

Instead, the reset task is broken into three phases. In the first phase, a node sends a reset request up the tree towards 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.

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 upwards 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 downwards 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.

The automaton code for this implementation is shown in Figure 2 and Figure 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.

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.

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
  - $(sbit_i = sbit_j)$

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

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 upwards *)
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_n

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_n

Figure 3: Original correction action in Arora and Gouda's Reset Protocol [AG90].

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*.

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. 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.)

Suppose that in some state s these link predicates hold for all links in the tree. Then [AG90] show 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 to 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.

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 the previous state.

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 show that in bounded time, all links will be in a good state and so the system is in a good state.

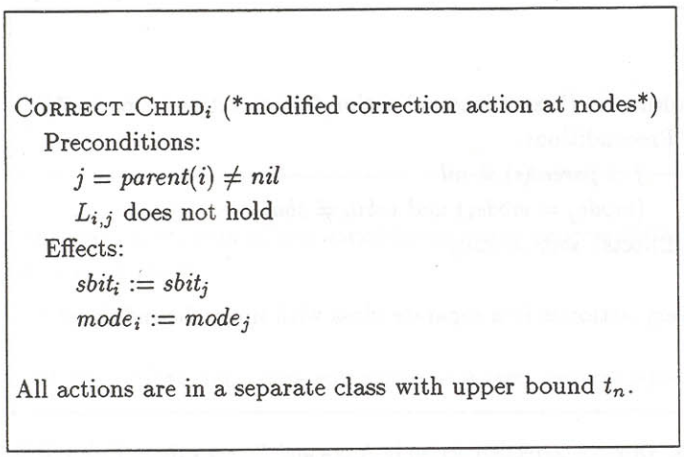

Figure 4: Modified Correction action for Arora and Gouda's Reset Protocol. All other actions are as in Figure 2.

5 Related Work

The original paper on local checking and correction [APV91b] proved a *local correction theorem*: any locally checkable protocol that met certain other constraints (referred to as *local correctability*) could be stabilized. However, that theorem *does not* imply our tree correction theorem. In order to apply the Local Correction Theorem, one has to explicitly exhibit a correction function with the appropriate properties; this is not needed in Tree Correction. Also, the explicit correction action used in local correction only depends on the previous state of a node; however, the implicit correction action used in the proof of Tree Correction depends on the state of a node's parent as well as the node's previous state. Finally, the definition of local checkability used in [APV91b] and [AGV94] 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.*

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], end-to-end protocols [APV91b]) for which tree correction is inapplicable. On the other hand, local correction does not seem applicable to the reset protocol described in Section 4. This is because the correction action (Figure 4) depends on both the state of the parent and the child. There are still other problems (e.g., mutual exclusion on a tree [Var93]) for which both techniques are applicable.

There has also been considerable 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[3] 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, they 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

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

in its domain that is consistent with its parent's value.

However, their tree protocol is limited to finding *static* solutions to constraint satisfaction. Thus constraint satisfaction can be used to solve a static problem like coloring but not a *dynamic* problem like mutual exclusion or network reset. In other words, 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 actions continue to occur. In order to allow this, we added an extra modification beyond the 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 were satisfied before the action is executed. This modification is crucial to ensure stability of the local predicates (Lemma 3.5) 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 3.3). None of these notions are required for the result in [CDK91].

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 towards its closest center. We believe that their method could be applied to make our theorem applicable to uniform tree networks. [CDK91] also has a number of other results on general networks, including a self-stabilizing procedure to find a feasible solution using distributed backtracking.

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] or [APV91b] imply our tree correction theorem.

6 Conclusions

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.

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 [APV91b] 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.

Although we have not described it here for lack of space, Dijkstra's first protocol in [Dij74] can also shown to be a locally checkable protocol that works on a line graph, and thus is amenable to tree correction. The correction actions we add are once again different from the original actions in [Dij74]. However, the corrections actions we add (and consequently the proofs) are much more transparent than the original version. See [Var93] for details.

As we have 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].

References

[AG90] Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AGV94] Anish Arora and Mohamed G. Gouda and George Varghese. Constraint Satisfaction as a Basis for Designing Nonmasking Fault-tolerance. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol 18, American Mathematical Society, 1994.

[APV91b] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[CDK91] Z. Collin, R. Dechter, and S. Katz. On the Feasibility of Distributed Constraint Satisfaction. *Proceedings of the 12th IJCAI*, August 1991.

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

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

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

[HS91] P. Humblett and S. Soloway. A Fail Safe Layer for Distributed Network Algorithms and Changing Topologies. Techical Report LIDS-P-1702, Lab for Information and Decision Sciences, MIT, May 1987.

[Var93] George Varghese. Self-stabilization by local checking and correction. Ph.D. Thesis MIT/LCS/TR-583, Massachusetts Institute of Technology, 1993.