Chapter 4

Local Checking and Correction in a Shared Memory Model

In Dijkstra's [Dij74] model, a network protocol is modelled 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 be modelled 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. For this chapter only, we will use Dijkstra's original model to *introduce* the method of local checking and correction. In later chapters, we will use a more realistic message passing model. Thus the goals of this chapter are:

- To describe some simple examples of local checking and correction that are more interesting than than the trivial examples given in Chapter 1.

- To show that existing work in [Dij74] and [AG90] can be understood very succinctly using the framework of local checking and correction.

The main result of the chapter is a theorem (Theorem 4.3.1) that states that any locally checkable protocol on a tree can be efficiently stabilized. To motivate this theorem, we begin in Section 4.2 with a reset protocol [AG90] due to Arora and Gouda. We examine the behavior of the Arora-Gouda protocol in a good state and conclude that the protocol is in a good state when all link subsystems are in a good state. Then we show how to add correction actions to the protocol such that if a link subsystem is in a bad state, it can be corrected to a good state. We also determine an order in which link subsystems can be corrected so as to ensure that the correction process converges.

In Section 4.3 we generalize the procedure followed in Section 4.2 to obtain Theorem 4.3.1. Then in Section 4.4 we show how one of Dijkstra's protocols [Dij74] can easily be understood using Theorem 4.3.1.

4.1 Modelling Shared Memory Protocols

We will use the version of the timed I/O Automaton model [MMT91] described in Chapter 2. How can we map Dijkstra's model into this model? Suppose each node in Dijkstra's model is a separate automaton. Then in the Input/Output automata 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 modelled 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.

4.2 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.

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 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 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,[1] 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 4.1 and Figure 4.2. Notice that besides the actions we have already described, there is a CORRECT$_i$ action

[1] in this chapter, we will always use the execution stabilization definitions

in Figure 4.2. 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 4.2 [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 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 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 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 4.1: 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 4.2: Original correction action in Arora and Gouda's Reset Protocol [AG90].

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.

The tree topology once again suggests a simple strategy. We remove the old action CORRECT_i in Figure 4.2 and add a new action CORRECT_CHILD_i as shown in Figure 4.3. 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. Rather than prove that this modified automaton stabilizes, we will prove a more general result in the next

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_n.

Figure 4.3: Modified Correction action for Arora and Gouda's Reset Protocol. All other actions are as in Figure 4.1.

section: that *any* locally checkable tree automaton can be locally corrected into a good global state.

We will return to the network reset problem in Chapter 7. Our stabilizing reset protocol is more efficient than the reset protocol of [AG90] and is also designed to work in a message passing model.

4.3 Tree Correction for Shared Memory Systems

In the last section, we described informally the problem of stabilizing a reset protocol on a tree. We also suggested a technique of adding correction actions to every node. That example motivates us to ask whether there is a general result for trees. To describe and prove such a general result, we start with the following definitions.

We will continue to 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 automaton* $\mathcal{N}$ 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].

In this chapter, we will often use the phrase "tree automaton" to mean a "shared memory tree automaton" and the phrase "network automaton" [2] to mean a "shared memory network automaton".

A *closed predicate* [3] 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

[2] In all subsequent chapters, the terms tree and network automaton have different meanings.

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

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 $L_{u,v}$.

Recall the definition of stabilization in terms of executions and the definition of an unrestricted automaton from Chapter 3. Recall that 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 4.3.1 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:

- 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 4.4. 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 $\mathcal{T}^+$.

- We add an extra correction action $\textsc{Correct}_u$ for every node u in the tree that is not the root. $\textsc{Correct}_u$ is also described in Figure 4.4. Intuitively, this extra action "corrects" the link between node u and its parent if this link is not in a "good" state. Each $\textsc{Correct}_u$ 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 4.4 is realizable. The careful reader will have noticed that we made two assumptions. First, in the $\textsc{Correct}_u$ 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.

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

Lemma 4.3.2 *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 4.3.3 *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 4.3.2, 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 labelling procedure depends crucially on the fact that the topology is a tree. ∎

To prove the theorem, we will use the Execution Convergence Theorem (3.4.5). 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 $v = \mathit{parent}(u)$ 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 4.3.4 *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: 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 4.3.3 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 4.3.3 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 4.3.5 *For any any (u, v) link subystem 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 $\textsc{Correct}_u$ or $\textsc{Correct}_v$ 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 of the (w, x) subsystem.

Using this partial order and Lemmas 4.3.4 and 4.3.5, we can apply the Execution Convergence Theorem (Theorem 3.4.5), to show 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.

4.3.1 Weakening the Fairness Requirement

In the previous subsection, 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.

It is interesting that the protocols in [Dij74, AG90] only require that some enabled action in the entire network occur in bounded time. In other words, all the actions in the entire automaton can be placed in a single class. How can we ensure the eventual correction property in a model in which the only guarantee is that some enabled action (in the entire network) will occur in bounded time? To make sure the eventual correction property holds in such a model, we need to show that it is impossible to remain for an unbounded amount of time in a state in which $L_{u,v}$ does not hold and in which some other action other than $\textsc{Correct}_u$ is enabled. This property can be established[4] quite

[4] I am grateful to Anish Arora for pointing this out to me.

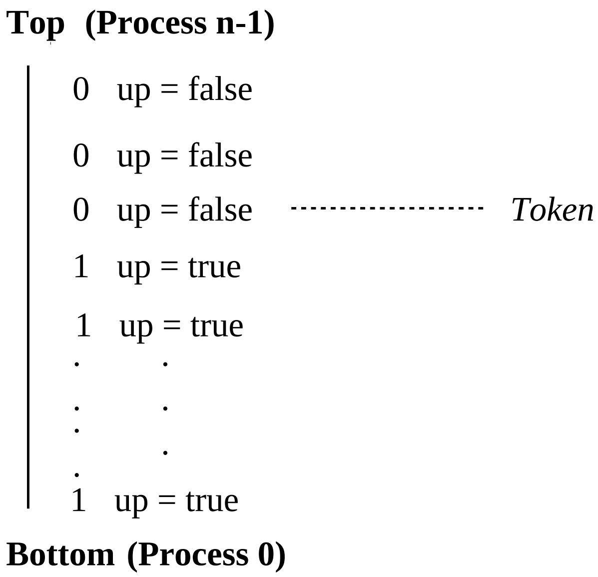

Figure 4.5: Dijktra's protocol for token passing on a line

easily for the protocols in [Dij74] and [AG90].

4.4 Rediscovering Dijkstra's Protocols

In this section, we will begin by reconsidering 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 4.5. 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.

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

Dijkstra's second example is modelled by the automaton $D2$ shown in Figure 4.6. Each process i has a boolean variable up_i, and a bit x_i. Roughly, up_i is a pointer at

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 $up_0 = true$ and $up_{n-1} = false$ by definition

In the initial state $x_i = 0$ for $i = 0 \ldots n-1$ and $up_i = false$ for $i = 1 \ldots n-1$

MOVE_UP$_0$ (*action for the bottom process only to move token up*)
Precondition: $x_0 = x_1$ and $up_1 = false$
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:
$x_i := x_{i-1}$;
$up_i := true$; (*point upwards 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 $up_i = true$ and $up_{i+1} = false$
Effect: $up_i := false$; (*point downwards in direction token was passed*)

All actions are in a single class with upper bound t_n.

Figure 4.6: 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 4 states.

node i that points in the direction of the token, and x_i is a bit that is used to implement token passing. Figure 4.5 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; else the token is at the lower node.

For the present, assume that all processes start with $x_i = 0$. Also, initially assume that $up_i = false$ for all processes other than process 0. We will remove the need for such initialization below. We start by understanding the correct executions of this protocol when it has been corectly initialized.

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 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.

In the good states for Dijktra's second example, the line can be partitioned into two bands as shown in Figure 4.7. 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 downwards). All nodes within the upper band point downwards 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 downwards; if the two bit values are unequal the token is moving downwards.

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. Intuitively, we see that if two neighboring nodes have the same pointer value, then their bits are equal; also if a node is pointing upwards, 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 $x_{i-1} = x_i$.

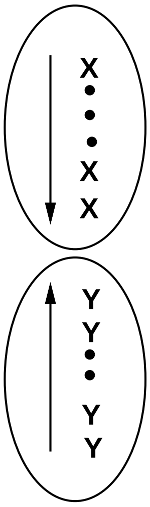

Figure 4.7: In the good states for Dijktra's second example, the line can be partitioned into 2 bands with the token at the boundary.

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

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 $up_{n-1} = \mathit{false}$ 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 $up_i = \mathit{false}$ and $up_{i-1} = true$. Consider the first such pair. Then the second predicate guarantees us 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 $\textsc{Move\_Down}_{i-1}$ is enabled. If $x_i = x_{i-1}$ and $i-1 = 0$ then only $\textsc{Move\_Up}_0$ is enabled. If $x_i \neq x_{i-1}$ and $i \neq n-1$ then only $\textsc{Move\_Up}_i$ is enabled. If $x_i \neq x_{i-1}$ and $i = n-1$ then only $\textsc{Move\_Down}_{n-1}$ is enabled.

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

Let L be the predicate of $D2$ which consists 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 4.3.1 to convert $D2$ into a new automaton $D2^+$ which stabilizes to executions in which 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 corrections actions we add (and consequently the proofs) are much more transparent than the original version.

Dijkstra also described two more stabilizing mutual exclusion protocols, one using K-state machines, where K is greater than the number of processes, and a solution using 3-state machines. In both solutions the topology is assumed to be a ring (i.e., the bottom and top processes are connected). The first protocol is easily seen to be locally checkable. Unfortunately it is no longer a tree automaton and hence Theorem 4.3.1 does not apply directly. However, with some extra work it is possible to derive the final protocol as a basic protocol augmented with correction actions that ensure that each link predicate becomes true. An even simpler way is to derive the K-state protocol using an idea that we call counter flushing (see Chapter 10 and Appendix E).

Dijkstra's three state protocol uses a different idea altogether. The protocol is not even locally checkable. This protocol seems extremely specific to the ring topology used. There are two main ideas. First, tokens are passed up and down a line in normal operation just as in the second example. Thus if there is more than one token and tokens must keep moving, the tokens must eventually "collide" at some node. This node can then destroy one of the tokens. This idea seems limited to a line/ring topology. The first idea, however, is not sufficient to detect a situation in which there are no tokens. The second idea is to exploit the neighbor relation between the top and bottom processes to detect the presence of no tokens. The states of the system are such that if there are no tokens, then all processes will have the same state. In normal operation, the states of the top and bottom nodes are always different. Thus the absence of tokens can be "suspected" locally if the state of the bottom node is not equal to that of the top node. In that case, a token is manufactured. The actual protocol can be understood using these ideas.

4.5 Summary

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. The main contribution of this chapter 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.

However, as we have argued at the beginning of this chapter, we believe that message passing models are more useful and realistic. For the rest of the thesis we will concentrate on message passing models. *The definitions of network automata, local predicates, local checkability, local correctability, and link subsystems that we used in this chapter are specific to shared memory systems.* In the next chapter (Chapter 5) we will introduce definitions of these concepts for networks in which nodes communicate by message passing. The definitions in Chapter 5 will be used for the remainder of the thesis.

The main theorem in this chapter 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 that is described at the end of Chapter 6.