Chicago Journal of Theoretical Computer Science

The MIT Press

Volume 1996, Article 5

5 December 1996

ISSN 1073–0486. MIT Press Journals, 55 Hayward St., Cambridge, MA 02142 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://www-mitpress.mit.edu/jrnls-catalog/chicago.html*
- *http://www.cs.uchicago.edu/publications/cjtcs/*
- *gopher.mit.edu*
- *gopher.cs.uchicago.edu*
- anonymous *ftp* at *mitpress.mit.edu*
- anonymous *ftp* at *cs.uchicago.edu*

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

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.

This article is included in the *Special Issue on Self-Stabilization*, edited by Shlomi Dolev and Jennifer Welch. The article presents a randomized self-stabilizing orientation algorithm for unicyclic networks. The algorithm is designed for a system of identical processors that supports only read/write atomicity. Under such settings, randomization is necessary to break symmetry and orient the ring. The expected stabilization time of the algorithm is $O(n^2)$. The correctness of the algorithm depends in part on a two-processor token-passing protocol. The correctness of the token-passing protocol is established by a state-space search computing the probability that the scheduling adversary can avoid self-stabilization.

The editor for this article is Vassos Hadzilacos.

Uniform Self-Stabilizing Orientation of Unicyclic Networks under Read/Write Atomicity

H. James Hoover Piotr Rudnicki

5 December, 1996

Abstract

Abstract-1 A distributed system is *self-stabilizing* if its behavior is correct regardless of its initial state. A *ring* is a distributed system in which all processors are connected in a cycle and each processor communicates directly with only its two neighbors. A ring is *oriented* when all processors have a consistent notion of their left and right neighbors. A ring is *uniform* when all processors run the same program and have no distinguishing attributes, such as processor IDs. A well-known self-stabilizing uniform protocol for ring orientation is that of [IJ93b]. For a ring of size n, this protocol will stabilize in expected $O(n^2)$ processor activations. This assumes that processors are scheduled by a *distributed demon*—one in which the communication registers between processors can be atomically updated (a read followed by a write), and the processors have the ability to make random choices.

Abstract-2 This paper generalizes the notion of orienting a ring to that of orienting a *unicyclic* network, that is, a ring with trees attached. We present a very simple protocol for the self-stabilizing orientation of such unicyclic networks. It has the same expected $O(n^2)$ processor activation performance as the Israeli and Jalfon protocol for rings. But ours works under the more general scheduling assumption of a *read/write demon*—one in which a read or write of a communication register is atomic, but an update (a read followed by a write) is not. We similarly assume the ability to make random choices. A subresult of our protocol is a uniform self-stabilizing algorithm for the two processor token-passing problem under the read/write demon.

Abstract-3 Our protocol is uniform in the sense that all processors of the same degree are identical. In addition, observations of the behavior of the protocol on an edge yield no information about the degree of the processors at the ends of the edge.

1 Self-Stabilization and Ring Orientation

1-1 The prototypical distributed system is a collection of discrete-state machines connected by a network. Such a system is *self-stabilizing* if it has the property that regardless of its current state, it will eventually enter and remain within a well-defined set of stable states, called the *target set* (or the *safe* or *legitimate set*). Although a longstanding notion in control theory, its first application to computing systems is generally credited to Dijkstra [Dij74, Dij86].

1-2 Dijkstra's original paper was virtually ignored for a decade, with the exception of [Kru79]. It took almost another decade before there was sufficient material[1] to warrant the survey by [Sch93]. Although our paper is self-contained, we assume that the reader is familiar with the general perspective of self-stabilization as would be provided by Schneider's survey.

1-3 Two fundamental problems of self-stabilization are *mutual exclusion* [AEY91, BGW89, BP89, CGR87, Dij74, Dij86, DIM93, Gho91, Her90, Her92, IJ90, Kes87, Kru79, Lam86a, Lam86b, LH91] and *ring orientation* [ASW88, HR91, IJ93b, SP87]. Mutual exclusion can be cast in terms of a *token-passing* problem, in which the privilege of entering the critical section is conferred by the possession of a token that is passed among the processors. Ring orientation is the problem of obtaining consensus among the processors for a consistent notion of left and right. That is, if processor A thinks its right neighbor is processor B, then processor B must think that its left neighbor is processor A.

1-4 On a ring network topology, mutual exclusion and orientation are closely related. For example, orienting a ring first, and then passing a token from left to right is one way to ensure fair exclusive access to a resource. On the other hand, if a token is being passed among processors in a ring, as it moves between processors it induces an orientation (call left the direction it arrives from, and right the direction it leaves). If the token ever arrives at a

[1]Self-stabilization is now an active research area, with a number of World Wide Web sites easily located by a search engine.

processor from a different direction than it departed, then the passage of the token through the ring will have established a consistent orientation.

1-5 Both of these problems become more difficult when processors are required to be identical and anonymous (i.e., they lack any distinguishing attribute such as a processor ID). Self-stabilization becomes even more interesting in a message-passing model of communication, rather than the shared-variable model we use here. (See [KP93], for example.)

1-6 Israeli and Jalfon [IJ93b] solve the problem of the uniform self-stabilizing orientation of a ring of processors. In their model, each processor is a finite-state machine that can make random choices and can communicate directly only with its two neighbors. Processors communicate through shared variables (or registers). Activation of the processors is under the control of a *distributed scheduling demon*. Under the distributed demon, an atomic transition of a processor consists of reading all input registers, changing the state, and then writing all output registers.

1-7 In their paper, Israeli and Jalfon show that no deterministic protocol exists for the self-stabilizing orientation of a ring. They then give a protocol that achieves orientation in expected $O(n^2)$ processor activations under the distributed demon. Their protocol is moderately complex. It is composed of two self-stabilizing subprotocols, one of which is randomized, the other which is deterministic. The subprotocols are then combined into the main protocol using Dolev, Israeli, and Moran's technique [DIM93] for the fair composition of self-stabilizing protocols.

1-8 In more recent results, Hoepman [Hoe94] gives a deterministic uniform algorithm for orienting rings of odd length, and Tsai and Huang [TH95] give results for rings of arbitrary size under the central demon, and odd-length rings under the distributed demon.

1-9 Our result has much in common with the scheduler-luck games of Dolev, Israeli, and Moran [DIM95, DIM94], as will be mentioned in Section 2.4.

1.1 Outline

1.1-1 This paper is organized as follows. Section 2 introduces our standard model of computation, scheduling demon, and the general-state exploration method for determining self-stabilization. Section 3 then applies the state-exploration method to show that the two-processor token-passing problem is self-stabilizing (in the standard model). Section 4 introduces the extension of orientation from rings to unicyclic networks. Section 5 then develops a uniform protocol for orienting unicyclic networks on a custom computation model that assumes a primitive operation for arbitrating conflicts between adjacent processors. Section 6 implements the custom model on the standard model by implementing the arbitration primitive using the self-stabilizing token-passing protocol, thus establishing the main result of the paper.

2 The Standard Model

2.1 Processor Networks

2.1-1 Our orientation protocol will ultimately be executed on what we call the *standard model* of computation. It is a *processor network* in which processors are placed at the vertices of the network, and all communication between processors occurs over I/O ports associated with the edges. Each processor at a vertex of degree k in the network has exactly k ports, numbered 0, 1, ... , $k-1$. Each port consists of an input register and an output register, details of which appear later.

2.1-2 A processor network is described by a connected graph $G = (V, E)$ with vertex set $V = \{0, \ldots, n-1\}$ and edge set $E \subset (V \times \{0, 1, 2, \ldots\}) \times (V \times \{0, 1, 2, \ldots\})$. Processor P_v is associated with vertex v. Each port of P_v is assigned to a distinct edge of E incident on vertex v. An edge $((v, i), (w, j))$ of E indicates that processor P_v has its i port connected to port j of processor P_w. The connection ensures that the output register of port i is connected to the input register of port j, and vice versa. We use this elaborate edge description because the notion of orientation requires the ability to specify a particular port number, and it is possible that two processors may be connected by multiple edges.

2.2 Basic Processor Operations

2.2-1 The *state* of a processor consists of the contents of its port registers, along with the current state of the program controlling the processor. The basic atomic operations of a processor are as follows: *read* on port i, which transfers the contents of the neighbor's output register to the input register of port i; *write* on port i, which writes a new value into the output register of port i; make an *internal step*, which simply changes the state of the internal program of the processor without altering the port registers; and make a uniform *random choice* from a finite set. In the case that the two processors on an edge activate simultaneously, a read on a port will complete before the write of the other processor to that port. Thus, when multiple processors are activated, executions in this model can be serialized by the simple expedient of scheduling all the read operations before all the write operations.

2.3 The Scheduling Demon

2.3-1 Reasoning about distributed systems usually assumes that state changes cannot be observed while they are occurring. That is, the operations that change the state are atomic, and are considered to occur instantaneously. Alternatively, one can think of atomic operations as sequences of internal actions that must be completed before the scheduler can activate another processor.

2.3-2 Reasoning proceeds by analyzing how the system behaves when the execution of the atomic operations are scheduled by various styles of demon. For example the *central demon* (introduced by Dijkstra in [Dij74]) schedules only one atomic shared-variable update operation at a time, while the *distributed demon* (introduced by Brown and co-workers in [BGW89]) is permitted to schedule simultaneous atomic update operations from different processors. The highly adversarial *read/write demon* (introduced by Dolev, Israeli, and Moran in [DIM93]) breaks shared-variable-update operations into read and write phases, and allows interleaving of the phases among processors.

2.3-3 We assume the read/write demon for our execution scheduler. In this model, the actions of the processors are serializable, that is, if two or more processors are activated simultaneously, it is possible to activate them sequentially and achieve the same result. The scheduler decides which single processor to activate at each step, and actively works to defeat the protocol.

2.3-4 Because of the way processors are defined, it is always possible for a processor to be activated at a scheduling step. Thus, without any restrictions on the scheduler, it could simply activate the same processor continuously, and avoid stabilization. One kind of restriction is that the scheduler be *fair*, as in [DIM93]. A fair scheduler must ensure that no processor has a monopoly on activations, nor is any processor excluded from activations.

2.3-5 Instead of worrying about the fairness of the scheduler, we will require that any activations by the scheduler must, if possible, make progress with nonzero probability. Intuitively this means that the scheduler can be as unfair as it wants, so long as it does not simply keep activating one processor in a busy-wait cycle. This serves to abstract out any busy-wait cycles from the execution of the protocol, and enables us to mechanically explore the state space.

Definition 2.1 *The read/write scheduling demon makes* progress *if it activates a processor that:*

1. *executes a write, or*

2. *executes a read that obtains a value different from the current contents of the input port, or*

3. *executes a read that obtains the same value as the current contents of the input port, but for which the next write, with nonzero probability, would result in a change in the writing processor's state.*

2.3-6 These conditions prevent the scheduler from activating a processor in the two-step cycle that first does a read that obtains no new data, and then does a write that writes no new data, leaving the processor back in its original state. Such a sequence is completely invisible to the other processors, and so could continue forever.

2.4 State Space Exploration

2.4-1 Reasoning about the behavior of a supposed self-stabilizing system is complicated by the fact that the system can be in any possible state, yet must eventually enter one of the target states. The reasoning is further complicated by the ability of processors to make random choices and the fact that the scheduler is an adversary that is working to defeat the protocol.

2.4-2 Let Q be the state space of the system. We assume that the designer has specified a target set $T \subset Q$ which contains the states that characterize the desirable behavior of the system. We also assume that T is *closed*, that is, once the system enters set T, all future states remain in T.

Definition 2.2 *Let target set T be a subset of the state space Q of a system. We say that the system is* self-stabilizing *iff for any initial state from Q, with probability 1 the state of the system eventually enters and remains in the set T.*

In any state q, the scheduler can choose to activate any processor it desires so long as it makes progress according to Definition 2.1. Our state-exploration approach views the scheduler as an adversary that attempts to deter self-stabilization as long as possible.

2.4-3 Let $\Gamma(q)$ denote the set of possible scheduling choices for state q. For each possible scheduling choice $\gamma \in \Gamma(q)$ that the scheduler can make, there is a set of possible states $\delta(q, \gamma) \subseteq Q$ that the system can enter. Each state of $\delta(q, \gamma)$ may or may not make progress relative to q. The precise content of $\delta(q, \gamma)$ is determined by the random choices that the activated processors make, and by other details of the model (for example, how read/write conflicts are resolved).

2.4-4 Let progress$[q, q']$ be the predicate that is satisfied iff state q' results from the activation in state q of exactly one processor that satisfies the three conditions of Definition 2.1.

2.4-5 Suppose that the scheduler makes scheduling decision γ. Then the scheduler makes progress according to Definition 2.1 iff there is at least one state q' in $\delta(q, \gamma)$ such that progress$[q, q']$.

2.4-6 Let

$$\sigma(q) = \{\, \delta(q, \gamma) \mid \text{scheduling decision } \gamma \in \Gamma(q) \text{ makes progress} \,\}$$

that is, $\sigma(q)$ contains all the possible next-state sets that result from taking a progress-making scheduling choice from $\Gamma(q)$ at state q. Note that the elements of $\sigma(q)$ are sets of states.

Definition 2.3 *A state $q \in Q$ is* deadlocked *iff $\sigma(q)$ is empty.*

2.4-7 Now we consider the behavior of the scheduler. Its goal is to maximize its probability of avoiding the target set.

Definition 2.4 *Let T be the target set of the system. We say that a scheduler can* avoid T from q for at least i steps *iff starting at state q the scheduler in the next i steps can keep the system outside T, or can force the system into a deadlocked state outside T. For each scheduler, the* avoidance probability *$p_i(q)$ of a state q is the probability that it can avoid T when starting from q for at least i steps.*

This definition is convoluted, because the notion of a scheduling step makes sense only if the current state is not deadlocked, that is, the scheduler can actually make progress.

2.4-8 What kind of scheduling strategy delivers the "best" avoidance probabilities? Given that the scheduler cannot predict the results of the independent equiprobable random choices made by the system, the best it can do is to maximize its probability of avoiding the target set.

Proposition 2.5 *The following avoidance probabilities are tight upper bounds for the avoidance probabilities of all schedulers. Let $q \in Q$. For $i = 0$, or $\sigma(q)$ being empty:*

$$p_i(q) = \begin{cases} 0 & \text{if } q \in T \\ 1 & \text{if } q \notin T \end{cases}$$

For $\sigma(q)$ being nonempty:

$$p_{i+1}(q) = \max_{S \in \sigma(q)} \left(\frac{1}{|S|} \sum_{q' \in S} p_i(q') \right)$$

Proof of Proposition 2.5 The scheduler is attempting to stay away from the target set as long as possible. If it can make progress, then to avoid the target set T for at least $i+1$ steps, it should make a choice from $\Gamma(q)$ that leads to a set of next states $S \in \sigma(q)$ that has maximum probability of avoiding the target set for at least i steps, or that leads to a deadlock state not in T.

Proof of Proposition 2.5 □

2.4-9 We can now give a sufficient condition for nonstabilization.

Proposition 2.6 *Let $n = |Q|$. Let $q \in Q$ be such that $p_n(q) = 1$. Then $p_i(q) = 1$ for all $i \geq 0$, and the scheduler can prevent the system from stabilizing when started in state q.*

Prove Prop 2.6-1 **Proof of Proposition 2.6** Consider how the computation proceeds from state q for n activations. The scheduler makes choices of which processors to activate. There may be many possible successor states resulting from the choice, so the possible future of the computation for the next n steps can be described by a tree of depth $\leq n$.

Prove Prop 2.6-2 Consider any path from root q to a leaf l. Either this path ends in a deadlocked state, or it has length n. If the path ends in a deadlocked state, that state must not be in T, because $p_n(q) = 1$. If the path does not end in a deadlocked state, then it has length n, and so it contains $n+1$ states. By the "pigeon-hole principle," there must be two identical states q' on the path. Break the transition to the deeper instance of q', and connect it to the one closer to the root. This transformation can be performed for each leaf in the tree, and the result is a graph whose only leaf nodes are deadlocked states not in T. All future state changes remain in the graph.

Prove Prop 2.6-3 Thus the scheduler can avoid T forever starting in state q, and thus $p_i(q) = 1$ for all $i \geq 0$.

Proof of Proposition 2.6 □

Lemma 2.7 *For all $q \in Q$, $i \geq 0$, we have $0 \leq p_i(q) \leq 1$, and $p_i(q) \geq p_{i+1}(q)$.*

Proof of Lemma 2.7 By induction on i.

Proof of Lemma 2.7 □

Lemma 2.8 *Suppose that there exists an $n > 0$ such that for every $q \in Q$, $p_n(q) < 1$. Let $p = \max_{q \in Q}(p_n(q))$. Then for all $q \in Q$, $i > 0$, $p_{ni}(q) \leq p^i$.*

Proof of Lemma 2.8 Consider avoiding T for segments of n steps. Now the probability of avoiding T for $\geq n$ steps is $\leq p$. Given that T has been avoided for n steps, by Lemma 2.7, avoiding T for another n steps or more has probability at most p. Thus, the probability of avoiding T for $\geq 2n$ steps is $\leq p^2$, and by induction, the probability of avoiding T for $\geq ni$ steps is $\leq p^i$.

Proof of Lemma 2.8 □

2.4-10 We can now give a sufficient condition for self-stabilization.

Proposition 2.9 *Let Q be the state space of a system with target set T. Suppose T is closed. If there exists an n, $0 \leq n \leq |Q|$, such that for every $q \in Q$, $p_n(q) < 1$, then the system is self-stabilizing.*

Proof of Proposition 2.9 If $n = 0$, then $Q = T$ and the system is trivially self-stabilizing. Suppose that there exists an $n > 0$ such that for every $q \in Q$, $p_n(q) < 1$. By Proposition 2.6, if such an n exists, it must be $\leq |Q|$. Then by Lemma 2.8 and Lemma 2.7, for every $q \in Q$,

$$\lim_{i \to \infty} p_i(q) = 0$$

Thus, the probability of reaching the target set T from any state is 1, and the system is self-stabilizing.

Proof of Proposition 2.9 □

2.4-11 Propositions 2.6 and 2.9 combine to give a necessary and sufficient condition for self-stabilization.

Theorem 2.10 *Let Q be the state space of a system with target set T. Suppose T is closed. There exists an n, $0 \leq n \leq |Q|$, such that for every $q \in Q$, $p_n(q) < 1$ iff the system is self-stabilizing.*

Proof of Theorem 2.10 Observe that the negation of the conditions for Proposition 2.6 imply the conditions for Proposition 2.9, and vice-versa.

Proof of Theorem 2.10 □

2.5 Expected Time to Self-Stabilization

2.5-1 We now consider how to estimate the expected time that the scheduler can avoid the target set when starting at some state q. The expected path length before intersecting the target set T is

$$l(q) = \sum_{i=0}^{\infty} i p_i(q)(1 - p_{i+1}(q))$$

This comes from observing that the probability that T can be avoided for exactly i steps is the probability that it can be avoided for at least i steps times the probability that it cannot be avoided for at least $i+1$ steps.

2.5-2 By directly computing the $p_i(q)$, $0 \leq i \leq n$, we can obtain a value for the initial part of the summation.

$$\sum_{i=0}^{n-1} i p_i(q)(1 - p_{i+1}(q))$$

And we need an estimate for the tail

$$e_n(q) = \sum_{i=n}^{\infty} i p_i(q)(1 - p_{i+1}(q))$$

Proposition 2.11 *Let n be such that for all q, $p_n(q) < 1$. Let $p = \max_{q \in Q} p_n(q)$. Then*

$$e_n(q) = \sum_{i=n}^{\infty} i p_i(q)(1 - p_{i+1}(q)) \leq \frac{np(3n + p - 1 - np)}{2(1-p)^2}$$

Proof of Proposition 2.11 We have

$$e_n(q) \le \sum_{i=n}^{\infty} i p_i(q)$$

By Lemma 2.8, the probability of avoiding T for $\ge i$ steps is $\le p^{\lfloor \frac{i}{n} \rfloor}$. Thus

$$\begin{aligned}
e_n(q) &\le \sum_{i=1}^{\infty} i p^{\lfloor \frac{i}{n} \rfloor} \\
&= \sum_{i=1}^{\infty} \sum_{j=0}^{n-1} (ni+j) p^{\lfloor \frac{ni+j}{n} \rfloor} \\
&= \sum_{i=1}^{\infty} \sum_{j=0}^{n-1} (ni+j) p^{i} \\
&= \sum_{i=1}^{\infty} p^{i} \sum_{j=0}^{n-1} ni+j \\
&= \sum_{i=1}^{\infty} p^{i} (n^2 i + n(n-1)/2) \\
&= n^2 (\sum_{i=1}^{\infty} i p^{i}) + n(n-1)/2 (\sum_{i=1}^{\infty} p^{i}) \\
&= \frac{n^2 p}{(1-p)^2} + \frac{n(n-1)p}{2(1-p)} \\
&= \frac{np(3n+p-1-np)}{2(1-p)^2}
\end{aligned}$$

as required.

Proof of Proposition 2.11 □

2.6 The Scheduler-Luck Game

2.6-1 Another approach to reasoning about self-stabilization is with the scheduler-luck game of Dolev, Israeli, and Moran [DIM95]. Self-stabilization is viewed as a game in which the players, scheduler and luck, take turns. The scheduler has control over which processor is activated, while luck can control the outcomes of the random coin tosses used by processors. The scheduler must be fair, but is otherwise unrestricted. Luck need not intervene to affect the outcome of a coin toss.

2.6-2 The scheduler is attempting to prevent entry into the target set, while luck is attempting to force entry by intervening to avoid undesirable random state transitions. Self-stabilization is established by finding a winning strategy for luck. That is, luck has to be able to force a path from any member of the initial-state set to some member of the target state. Luck is said to have an (f, r)-strategy iff for every initial state and every scheduler, the system reaches the target set in expected number of at most r rounds, and at most f interventions of luck. The scheduler-luck game provides an upper bound on the time to stabilize: if an (f, r)-strategy exists, then the system will stabilize from that initial state in at most $r2^f$ expected number of rounds.

2.6-3 In the usual application of the scheduler-luck model, one determines a strategy for luck and then proves that it is a winning one. This can be quite difficult, since the difference between a system that is obviously self-stabilizing and one that is not is often quite subtle (see Section 3.1 for an example).

2.6-4 On the other hand, one could take a state-space-exploration approach to the scheduler-luck game. It would require a search to show that luck, from every initial state, could intervene to force the path to reach the target set. It is not obvious how to determine the minimum number of interventions by luck in order to minimize the upper bound on time to stabilize.

3 Token Passing in the Standard Model

3-1 The token-passing problem is to find a protocol for a pair of processors such that exactly one token is continuously passed between the processors. We will apply the state-exploration method of Section 2.4 to show that token passing has a uniform self-stabilizing solution under the read/write demon. Although this result stands on its own, it will be key to the ultimate implementation of the orientation protocol of Section 6.

3-2 This particular algorithm, in minor variations, appears in [IJ93a] and in [HR91], but this is the first complete proof for the case of the read/write demon. A nonuniform version of this problem appears in [DIM93]. In that proof, the processors run different programs, but the system is also self-stabilizing under the read/write demon.

3-3 Consider a pair of processors, executing the protocol of Figure 1. The port alphabet is $\{W, P, R\}$, with the intuition behind the symbols as follows: at an input port, a W means that the generating processor is waiting to receive the token; a P means that it is in the process of passing the token to the other processor; and an R means that it has received the token and is willing to pass it back. The intent is that two processors executing this protocol will, when stabilized, go through the cycle of modes:

$$WP, RP, RW, PW, PR, WR, WP, \ldots$$

3-4 We now analyze the behavior of the scheduling demon in terms of this pair of processors. First, we have to identify the state space. Each processor has a mode, the contents of its port registers, and a position in the program. We examine where the processor actually interacts with its neighbor, and consider the program position as two phases. The read phase is the single instruction that reads the input port. The write phase consists of the sequence of instructions that computes the next value of *mode* and writes it to the output port. For the purposes of the scheduling demon, we can consider the phases to be atomic operations, as each phase contains exactly one information exchange with the neighboring processor. Note that the read and write phases are serializable with respect to processor activations.

3-5 The state of an individual processor can then be captured as a triple

$$\langle \mathit{phase}, \mathit{inport}, \mathit{mode} \rangle$$

where $\mathit{phase} \in \{r, w\}$ indicates the phase that is next to be executed on activation, $\mathit{inport} \in \{R, W, P\}$ is the current contents of the input port, and $\mathit{mode} \in \{R, W, P\}$ is the current value of the mode.

3-6 The state of the system is the pair of states of the processors. The state space Q is all possible such pairs of processor states. The σ function of Section 2.4 is then defined under the assumption that scheduling is being done by a read/write demon.

Proposition 3.1 *The token-passing algorithm is self-stabilizing under the read/write scheduling demon. The expected number of activations before stabilization is at most 166, and the average over all states is at most 81.*

Prove Prop 3.1-1 **Proof of Proposition 3.1** We prove self-stabilization by a computation that establishes the conditions of Theorem 2.10. This is performed by the C code of *explore.c*. The results of running this code, given in *explore.out*,

$mode$: $\{R, W, P\}$ (Current state of protocol)
$inport$: $\{R, W, P\}$ (Input port)
$outport$: $\{R, W, P\}$ (Output port)

do forever {

(Read Phase)

read $inport$;

(Write Phase)

(Compute next mode from the table below)

Next Mode		Input W	Input P	Input R
Current Mode	W	(W, P, R)	R	W
	P	P	(W, P, R)	W
	R	P	R	(W, P, R)

if $(mode = inport)$ {
$mode := random\{W, P, R\}$;
} **else if** $(mode = W \wedge inport = P)$ {
$mode := R$;
} **else if** $(mode = P \wedge inport = R)$ {
$mode := W$;
} **else if** $(mode = R \wedge inport = W)$ {
$mode := P$;
}
write $mode$ **to** $outport$;

}

Figure 1: Pair token-passing protocol

substantiate the claim. These files, plus supporting materials, are provided alongside this article in the journal archives (see the Appendix).

Prove Prop 3.1-2 What remains to be argued is the correctness of the program. This issue has two components: the correctness of the overall algorithm, and the correctness of the specific portions related to the token-passing problem.

Prove Prop 3.1-3 The overall algorithm is quite simple:

1. Define the problem-specific state-space data, Q and σ.

2. Verify that there are no deadlocks (as per Definition 2.3).

3. Define the problem-specific target set, T.

4. Compute avoidance probabilities (as per Proposition 2.5), and expected times to stabilization (as per Proposition 2.11).

The implementation of the overall algorithm (i.e., *explore.h*, *explore.c*) requires problem-specific information, that is, Q, σ, and T. Assuming the correctness of this data, verifying that the main exploration algorithm is correct is a matter of a straightforward code walk-through of the simple computations in steps 2 and 4.

Prove Prop 3.1-4 The C code for generating the problem-specific data, given in *problem.h*, *problem.c*, is not particularly subtle in design or implementation. Its verification involves the following main components:

1. The state of an individual processor is defined, and then the state of the system is defined as a pair of processor states. One must ensure that the states of the system can be mapped 1-1 onto indexes in the range $0 \ldots |Q| - 1$. The conversions between states and indexes are simple modular arithmetic.

2. It must be checked that the function *next_proc_state* properly captures the result of activating one processor executing the program of Figure 1.

3. The *progress* function must be checked for a proper implementation of the progress predicate.

4. The function *sigma_fn* must be checked for a proper implementation of the σ function.

5. One must verify that the function *compute_target* follows a cycle in the state space.

All of these functions have simple structures and trivial invariants.

Prove Prop 3.1-5 The final step is to verify that the target set determined (and output) by *compute_target* really is that of the pair token-passing protocol.

Proof of Proposition 3.1 □

3.1 Remarks on Computational Verification

3.1-1 First, we note that if instead of randomly choosing from three modes, we select from $\{R, P\}$ (or any other two of the three), then the resulting protocol is still self-stabilizing, but the maximum expected stabilization time increases to 350, and the average is 142 steps.

3.1-2 The subtlety of the token-passing problem and the advantage of mechanical state exploration are illustrated by one of our initial attempts at a protocol:

Next Mode		Input		
		W	P	R
Current Mode	W	(W, R)	R	W
	P	P	W	W
	R	P	R	W

The intuition here is that arbitrations only occur in one state. This protocol was proven correct by hand, but, in fact, it does not self-stabilize. An instrumented version of the state-space exploration program found the counter-example strategy of Figure 2. This protocol does work if the random choice from $\{W, R\}$ is replaced by one from $\{W, R, P\}$. In this case, the maximum expected stabilization time is 597 activations, and the average over all starting states is 296.

3.1-3 One concern of the exploration method is that state spaces grow exponentially, and so exploring them quickly becomes infeasible. But large problems are usually solved on a custom abstract machine with primitive operations tailored to the task at hand. The custom machine simplifies the design of, and reasoning about, the algorithm that solves the problem. It is typically not nearly as rich in terms of scheduling possibilities as the standard model, and so is easier to reason about. Of course, the custom machine then must be implemented correctly on that standard model of comutation. For example, in this paper, token passing is used to implement the edge-arbitration primitive of the custom machine for the orientation protocol. By building a

Label	State		Action
	A	B	
174:	wRR	wWR	Activate B
167:	wRR	rWP	Activate B
173:	wRR	wRP	Activate A
29:	rRW	wRP	Activate A
299:	wPW	wRP	Activate B
289:	wPW	rRW	Activate B
301:	wPW	wWW	Activate B, randomly goto 291 or 292
292:	wPW	rWW	Activate B, goto 301
291:	wPW	rWR	Activate A
300:	wPW	wWR	Activate A
120:	rPR	wWR	Activate A, goto 174

The current state is for a pair of processors, A and B. The triple in the state of the processor indicates the $\langle phase, inport, mode\rangle$ as described above. All transitions go to the next row of the table unless indicated by a goto. The state labels are the same ones as generated by the exploration code.

Figure 2: Counterexample for bad pair token-passing protocol

custom model of computation, full state-space exploration is then reserved for the task of verifying the implementation of the custom machine's primitives on the standard model. In these cases, the state space is much smaller and more tractable to explore.

3.1-4 Also, from our limited observations, if a system does stabilize, the conditions for Theorem 2.10 are satisfied for small values of n relative to the size of the state space.

3.1-5 As a final note, one could raise the issue of the correctness of the program used in this kind of proof. But in practice, it is no more difficult to check the correctness of a program than it is to check the correctness of a proof. Like programs, proofs that are well structured with simple components are easy to check, while baroque, convoluted ones are not. Once verified, a program for testing self-stabilization can be used on many different instances of the problem.

4 Overview of Unicyclic Network Orientation

4-1 The notion of orienting a ring can be generalized to orienting any unicyclic graph, that is, a ring with attached trees. Consider a processor network with vertices of possibly different degree. To each vertex v of degree k is assigned an *orientation* $o(v) \in \{0, \ldots, k-1\}$. It is convenient to visualize each vertex as having an internal pointer whose head is directed toward port $o(v)$. The orientation of a vertex v is generally a function of the state of the processor P_v. The orientation of the vertices at the ends of an edge induces a state for that edge. We say that an edge is *oriented* when exactly one of the vertices at the ends of each edge is oriented toward that edge.

4-2 We say that a network is *oriented* iff every edge is oriented. There must be exactly as many vertices as edges for a network to be orientable, so such a network consists of a single ring with trees attached. There are precisely two ways to orient the ring, and it is easy to show that each edge outside of the ring must be oriented to point toward the ring. Such an oriented network is illustrated in Figure 3.

4-3 In general terms, the *orientation problem* is to specify a protocol whose execution, given an initial configuration of orientations of vertices, eventually results in an oriented network. We say that an orientation protocol is *correct* under a given computation model if it orients every permissible initial configuration of every permissible network topology.

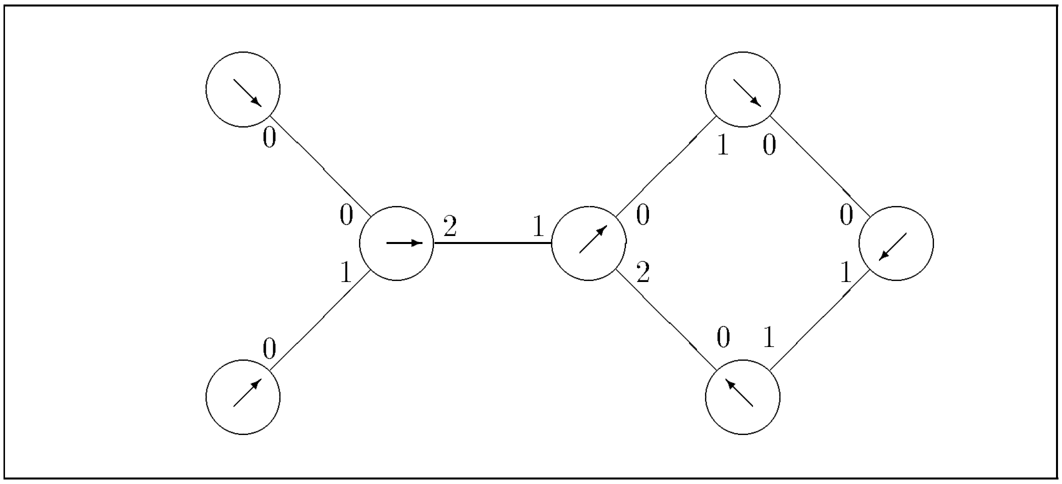

Figure 3: An oriented network

4-4 There are obvious scenarios in which the orientation problem is solved by the simple expedient of orienting the network at construction time. We are not interested in these. Nor are we interested in situations that use an elected leader to specify the orientation.

4-5 We are interested in *uniform* orientation protocols in which the processors have no IDs and no idea of the global network topology. That is, the protocol computation by the processor at a vertex is a function only of the degree of the vertex. But we achieve an even more restrictive notion of uniformity. Although different-degree processors are clearly executing different programs, they do not communicate this fact to their neighbors. That is, by observing an edge, it is not possible to determine the degree of either processor at the end of the edge.

4-6 It is the fact that such a simple protocol works for the generalization of rings to unicyclic networks, and does so under an even more restricted notion of uniformity, that makes this problem interesting.

4-7 We will present our orientation protocol first as a high-level protocol executed on a custom abstract machine. Then we will show how to implement the custom machine under our standard model with its read/write scheduling demon.

4-8 The custom machine for the high-level protocol will be a network of finite-state machines in which the state of an edge will cause its pair of adjacent machines to make simultaneous state transitions. We will show that this protocol is correct and has the claimed performance. The correctness of the high-level protocol will be established by standard proof methods.

4-9 We do not explicitly use randomness in the protocol on the custom machine. Instead, we assume that there is a primitive operation that arbitrates a certain kind of conflict between adjacent processors. Of course, the fact that randomness is required eventually is due to the deterministic impossibility result for rings of [IJ93b]. Section 6 will show how to implement the custom machine on the standard model by using the token-passing protocol of Section 3.

5 The High-Level Protocol and Custom Machine

5-1 The high-level protocol will be described using a simpler computation model than is used for token passing. In this custom model of computation, the processors in the network are finite-state machines with a somewhat unusual scheduling demon. At each step of the computation, the scheduler examines the *edges* of the network, and selects a *pair* of adjacent processors that satisfy the conditions of the protocol (described in detail in Figure 5). It then simultaneously changes the states of both processors. Each application of the protocol to a current configuration of two adjacent processors which produces a next configuration of the two processors is termed an *interaction*. If no interactions are possible on any edge, then the high-level protocol is said to be *deadlocked*.

5-2 We now describe the processors appearing at each vertex. In addition to an orientation as described above, each processor has a *mode* that is either P or W. We denote the mode of the vertex v by $m(v)$. The mode P stands for *passing mode*, which can be thought of as the processor being in possession of a token and wanting to pass it to the neighbor it is oriented toward. The mode W stands for *waiting mode*, which can be thought of as the processor waiting for a token to appear.

5-3 This induces a condition for each edge described by the orientation and mode of the processors at its ends. Figure 4 defines the various states of an edge. An edge has two ways of being oriented, and two ways of being disoriented. The protocol is designed so that processors will only interact with the neighbor they are oriented toward, and thus the rules for the protocol

Edge Name	Condition	Pictograph
Properly Oriented Edge	$o(v) = i$ $m(v) = W$ $o(w) \neq j$	$(\overset{W}{\rightarrow})$ —— $(\rightarrow)$
Improperly Oriented Edge	$o(v) = i$ $m(v) = P$ $o(w) \neq j$	$(\overset{P}{\rightarrow})$ —— $(\rightarrow)$
Properly Disoriented Edge	$o(v) = i$ $m(v) \neq m(w)$ $o(w) = j$	$(\overset{P}{\rightarrow})$ —— $(\overset{W}{\leftarrow})$
Improperly Disoriented Edge	$o(v) = i$ $m(v) = m(w)$ $o(w) = j$	$(\overset{P}{\rightarrow})$ —— $(\overset{P}{\leftarrow})$ or $(\overset{W}{\rightarrow})$ —— $(\overset{W}{\leftarrow})$
Ignored Edge	$o(v) \neq i$ $o(w) \neq j$	$(\leftarrow)$ —— $(\rightarrow)$

Figure 4: States of an edge $e = ((v, i), (w, j))$

Current Configuration	Next Configuration
$o(v) = i,\ m(v) = P$ $o(w) = j,\ m(w) = W$ $(\xrightarrow{P})$ — $(\xleftarrow{W})$ $(\xrightarrow{W})$ — $(\xleftarrow{P})$	$o(v) = i,\ m(v) = W$ $o(w) = (j+1) \bmod \deg(w),\ m(w) = P$ $(\xrightarrow{W})$ — $(\xrightarrow{P})$ $(\xleftarrow{P})$ — $(\xleftarrow{W})$
$o(v) = i,\ o(w) = j$ $m(v) = m(w)$ $(\xrightarrow{W})$ — $(\xleftarrow{P})$ or $(\xrightarrow{P})$ — $(\xleftarrow{W})$	$o(v) = i,\ o(w) = j$ $m(v) \neq m(w)$ (arbitrarily) $(\xrightarrow{P})$ — $(\xleftarrow{P})$ or $(\xrightarrow{W})$ — $(\xleftarrow{W})$

Figure 5: The high-level protocol at edge $e = ((v, i), (w, j))$

are very simple: *all progress toward orientation occurs at disoriented edges.*

5-4 Suppose that $e = ((v, i), (w, j))$ is a disoriented edge. The protocol at e is described by Figure 5. Note that the protocol applies even when one of the vertices is a leaf.

5-5 The behavior of the protocol depends on whether the edge is properly or improperly disoriented. For a properly oriented edge, the passing processor drops into waiting mode, and the waiting processor reorients itself to its next port and enters passing mode. On the other hand, for an improperly oriented edge, the mode conflicts must first be *arbitrated*, converting the edge into a properly oriented one. We assume only that some arbitration mechanism exists. It need not be fair. (In the implementation, this arbitration will be done randomly.)

5-6 The following is a direct consequence of the definition of the high-level protocol.

Proposition 5.1 *A network executing the high-level protocol is deadlocked if and only if it is oriented.*

5-7 We must now prove that every possible configuration of the network eventually deadlocks. How does the protocol make progress? Consider a possible execution of the protocol on a path as described by Figure 6. For clarity,

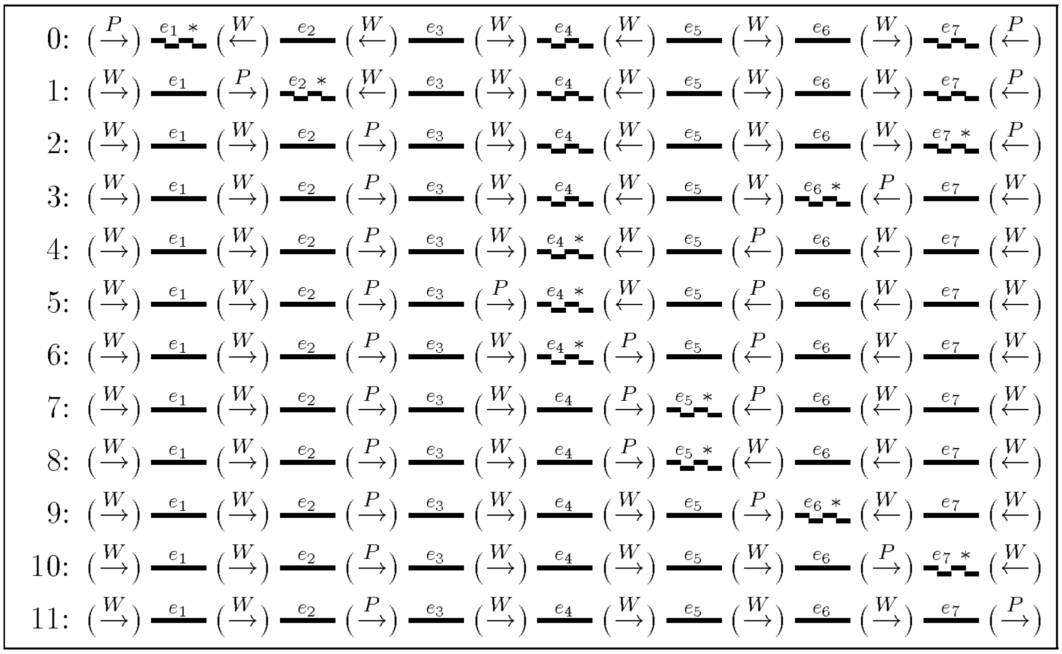

Figure 6: Example protocol execution

disoriented edges have been indicated by , and the edge marked with a $*$ at each step is the interaction.

5-8 We can think of the protocol as transferring the disorientation of an edge (e.g., e_1 at Step 0) to an adjacent edge (e.g., e_2 at Step 1), leaving the first edge oriented. The disorientation state keeps moving in its original direction until it either collides with an ignored edge and is absorbed (e.g., e_2 absorbed by e_3 when moving from Step 1 to Step 2), or reflects off of an improperly disoriented edge (e.g., the disorientation initially at e_7 reflects off of the improper disorientation initially at e_4). A collision with an ignored edge reduces the number of unoriented edges, and so the protocol makes progress. Reflection off of an improperly disoriented edge sometimes makes progress in orientation, and always ensures that arbitration is never required again at that edge.

5-9 In addition to improperly oriented edges that may be present at the beginning of execution, the reorientation of processors during execution can create improperly oriented edges (e.g., e_3 at Step 2). However, the definition of the protocol ensures the following.

Lemma 5.2 *During the execution of the protocol, at most one arbitration can occur at each edge. Furthermore, the execution of the protocol cannot generate any ignored edges.*

Proof of Lemma 5.2 Arbitration occurs only at $(\overset{P}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{P}{\leftarrow})$ or $(\overset{W}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{W}{\leftarrow})$ edges. In either case, the arbitration results in either a $(\overset{P}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{W}{\leftarrow})$ or $(\overset{W}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{P}{\leftarrow})$ edge. Now the state of an edge changes only when both processors are oriented toward the edge. Assuming neither processor is a leaf, the next state of $(\overset{P}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{W}{\leftarrow})$ is $(\overset{W}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{P}{\rightarrow})$, and the next state of $(\overset{W}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{P}{\leftarrow})$ is $(\overset{P}{\leftarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{W}{\leftarrow})$. The protocol rules determine that a processor turns away from an edge only when it is in W mode, and must enter into P mode after turning away. Thus, the next state of the edge $(\overset{W}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{P}{\rightarrow})$ must be $(\overset{W}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{P}{\leftarrow})$, and the next state of the edge $(\overset{P}{\leftarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{W}{\leftarrow})$ must be $(\overset{P}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{W}{\leftarrow})$. Thus, the cases that result in an arbitration at the edge can never re-occur. We have a slightly different behavior in the event that one processor is a leaf (both cannot be). Suppose the left processor is a leaf. Then the next state of $(\overset{W}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{P}{\leftarrow})$ is $(\overset{P}{\rightarrow}) \rule[0.5ex]{1.5em}{1.5pt} (\overset{W}{\rightarrow})$, and the same argument against arbitration applies.

Proof of Lemma 5.2 □

5-10 Since arbitrations and reflections occur only at improperly disoriented edges, there is an upper bound on the number of reflections that can occur at an edge. Ignored edges are also important, as they are points at which orientation conflicts are resolved.

5-11 It is convenient to reason about the orientation protocol's behavior on a tree. An *edge-rooted tree* T with root vertex v and root edge e is constructed by taking a tree with root v and edge e incident on v, and deleting the vertex at the other end of e. All of our trees will be edge-rooted, so we simply use the term *tree*. Figure 7 illustrates a prototypical tree used in the proofs that follow.

5-12 The next two lemmas show that ignored and disoriented edges are balanced in a network. Let $\#_I(G)$ and $\#_D(G)$ denote, respectively, the number of ignored and disoriented edges in the network G (either proper or improper). When applied to a tree T, the root edge of T is not counted.

Lemma 5.3 *Let T be a tree with root vertex v and root edge e. If v is oriented toward e, then $\#_I(T) = \#_D(T)$. If v is oriented away from e, then $\#_I(T) = \#_D(T) - 1$.*

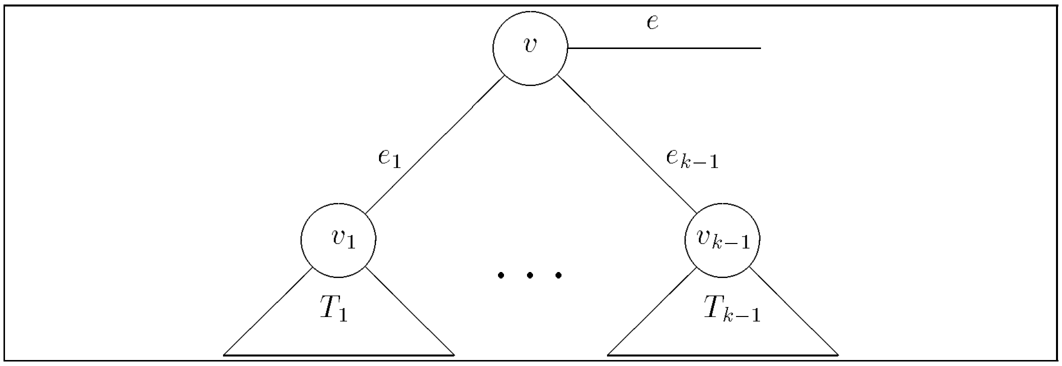

Figure 7: A typical edge-rooted tree

Prove Lemma 5.3-1 **Proof of Lemma 5.3** We proceed by induction on the size of T. For the case of T being a single vertex (i.e., degree 1), v must be oriented toward e, and we have $\#_I(T) = \#_D(T) = 0$.

Prove Lemma 5.3-2 Suppose that vertex v has degree $k > 1$, and edges $e_1, \ldots, e_{k-1}$ in addition to e. Then T looks like the tree of Figure 7.

Prove Lemma 5.3-3 When v is oriented toward e, each of the edges e_l, $1 \leq l < k$, is either ignored or oriented. If e_l is oriented, then v_l is oriented toward e_l, and by induction $\#_I(T_l) = \#_D(T_f l)$. If e_l is ignored, then v_l is oriented away from e_l, and by induction $\#_I(T_l) = \#_D(T_l) - 1$. Adding the ignored edge e_l maintains the balance between ignored and disoriented edges.

Prove Lemma 5.3-4 When v is oriented away from e, then it is oriented toward exactly one edge e_l, which is either oriented or disoriented. Balance is maintained for the other subtrees as above. If e_l is oriented, then v_l is oriented away from e_l, and by induction $\#_I(T_l) = \#_D(T_l) - 1$, and so $\#_I(T) = \#_D(T) - 1$. If e_l is disoriented, then v_l is oriented toward e_l, and by induction $\#_I(T_l) = \#_D(T_l)$. Accounting for the disoriented e_l we have $\#_I(T) = \#_D(T) - 1$.

Proof of Lemma 5.3 □

Lemma 5.4 *Let G be a unicyclic network. Then $\#_I(G) = \#_D(G)$.*

Prove Lemma 5.4-1 **Proof of Lemma 5.4** If G is oriented, then $\#_I(G) = \#_D(G) = 0$. Suppose that G is not oriented. Pick any edge $e = ((v, i), (w, j))$ on the ring of G and cut it, attaching a leaf u to vertex w with the edge $f = ((u, 0), (w, j))$. The net result is a tree T with root vertex v and root edge e.

Prove Lemma 5.4-2 If e was originally disoriented, then edge f will be disoriented, and v will be directed toward e. That is, edge $e : (\rightarrow)$ —— $(\leftarrow)$ becomes the edge $f : (\rightarrow)$ —— $(\leftarrow)$, and e becomes the root edge $(\rightarrow)$ ——. Applying Lemma 5.3 to T we have $\#_I(T) = \#_D(T)$. Since the disoriented edge f in T accounts for the originally disoriented edge e, we have balance for G.

Prove Lemma 5.4-3 If e was originally ignored, then edge f will be oriented, and v will be directed away from e. That is, edge $e :\ (\leftarrow)$ —— $(\rightarrow)$ becomes the edge $f :\ (\rightarrow)$ —— $(\rightarrow)$, and e becomes the root edge $(\leftarrow)$ ——. Applying Lemma 5.3 to T, we have $\#_I(T) = \#_D(T) - 1$. The oriented edge f is not counted in T, nor is the originally ignored edge e. Accounting for e we have $\#_I(G) = \#_D(G)$, and so we have balance for G.

Prove Lemma 5.4-4 If e was originally oriented, we have one of the above cases.

Proof of Lemma 5.4 □

5-13 Since progress is made as disoriented edges move about the tree, we need to know how they can interact. In a tree T, we say that an edge e is *between* edges e_1 and e_2 if e lies on a path between e_1 and e_2. We say that a disoriented edge e_1 in tree T is *covered* if there exists an ignored edge between e_1 and the root edge of T.

Lemma 5.5 *Let T be a tree with root vertex v and root edge e. If v is oriented toward e, then every disoriented edge in T is covered. If v is oriented away from e, then all but one disoriented edge in T is covered. Furthermore, every ignored edge covers some disoriented edge.*

Prove Lemma 5.5-1 **Proof of Lemma 5.5** We proceed by induction on the size of T. Assume that the lemma is true for all trees of smaller size than T. For the case of T being a single vertex (i.e., degree 1), we have no disoriented edges.

Prove Lemma 5.5-2 Suppose that vertex v has degree $k > 1$, and edges $e_1, \ldots, e_{k-1}$ in addition to e. Then T looks like the tree of Figure 7.

Prove Lemma 5.5-3 When v is oriented toward e, each of the edges e_l, $1 \leq l < k$, is either ignored or oriented. If e_l is oriented, then v_l is oriented toward e_l, and by induction all disoriented edges of T_l are covered. If e_l is ignored, then v_l is oriented away from e_l, and by induction T_l has one uncovered disoriented edge f. But e_l is between f and the root edge e, and so f is covered by e_l. Thus, all disoriented edges in T are covered.

Prove Lemma 5.5-4 When v is oriented away from e, then it is oriented toward exactly one edge e_l which is either oriented or disoriented. Any uncovered disoriented edges for the other subtrees are covered as above. If e_l is oriented, then v_l is oriented away from e_l, and by induction T_l has an uncovered, disoriented edge that remains uncovered in T. If e_l is disoriented, then v_l is oriented toward e_l, and by induction T_l has all disoriented edges covered. Thus e_l is the only uncovered edge of T.

Proof of Lemma 5.5 □

5-14 Ignored edges serve as separators between disoriented edges in the following manner.

Lemma 5.6 *Let T be a tree with root vertex v and root edge e with v oriented away from e. Then T can be partitioned into $1 + \#_I(T)$ subtrees such that the root edge of each subtree is either e or an ignored edge in T, and every subtree contains exactly one disoriented edge.*

Prove Lemma 5.6-1 **Proof of Lemma 5.6** We proceed by induction on the size of T. Suppose that the lemma is true for all trees smaller than T.

Prove Lemma 5.6-2 First, in the case that $\#_I(T) = 0$, then T is partitioned into one tree with root edge e. By Lemma 5.3 $\#_D(T) = 1$, and thus there is exactly one disoriented edge.

Prove Lemma 5.6-3 Now consider the case where $\#_I(T) > 0$. Since v is oriented away from e, then by Lemma 5.5 there is exactly one disoriented edge f that is not covered. That is, f lies in the subtree rooted at e obtained by cutting T at the ignored edges nearest v on any path from v to a leaf. Call this tree T_f. Suppose that k ignored edges were cut, resulting in subtrees $T_1, \ldots, T_k$, each of whose root edges was an ignored edge of T. Now T_f does not contain any ignored edges, and k ignored edges were destroyed by the cutting process, so $\#_I(T) = k + \sum_{i=1}^{k} \#_I(T_i)$. By induction, the trees T_i can be partitioned into $\sum_{i=1}^{k}(1 + \#_I(T_i)) = \#_I(T)$ subtrees with the required properties. In particular, the root edge of each subtree is an ignored edge in T. Since the root edge of T_f is e, and T_f contains the disoriented edge f, we have the required partition into $1 + \#_I(T)$ subtrees.

Proof of Lemma 5.6 □

5-15 Now consider how a single proper disorientation moves about a tree of otherwise properly oriented edges. By Lemma 5.3, there is one disoriented edge and no ignored edges, thus the root vertex is oriented away from the root edge. The protocol forces the disoriented edge to move about the tree in a depth-first order that is induced by the port numbers at each vertex. For the typical tree (Figure 7), suppose that e_1 is properly disoriented with v in passing mode P. The disorientation moves from e_1 into subtree T_1, moves about T_1 in depth-first order, and returns to e_1 with v_1 in mode P and v in waiting mode W. The disorientation then passes to e_2. This process continues until v is oriented toward e in mode P. That is, the disorientation has moved out of T. We call the sequence of edges that a disorientation follows as it depth-first searches the tree a *trip*.

5-16 Two things can affect the trip that a disorientation takes in an arbitrary tree. One is encountering an ignored edge. When this happens, the protocol replaces the ignored and disoriented edges with properly oriented ones, and the trip terminates. The other possibility is that the disorientation will encounter an improperly oriented edge (e.g., e_4 of Figure 6). In this case, it is possible for the resulting arbitration to make the disoriented edge bounce, causing the subtree below to be either skipped (when approaching from above), or traversed again (when approaching from below). A bounce, since it requires an arbitration, can occur at most once at each particular edge. Call an edge that has not yet participated in an arbitration an *unarbitrated edge*.

5-17 Thus, we can measure progress in the protocol by observing the decrease in the number of ignored and unarbitrated edges.

Lemma 5.7 *Let T be a tree with root vertex v and root edge e. Then:*

1. *T contains only oriented edges; or*
2. *every oriented edge in T is properly oriented, and there is exactly one disoriented edge; or*
3. *in at most* $2\text{size}(T)$ *interactions between processors of T, the total number of ignored plus unarbitrated edges in T will decrease by* 1.

Prove Lemma 5.7-1 **Proof of Lemma 5.7** In order for the protocol to be active, T must contain at least one disoriented edge, so we assume that (1) does not hold. If there are no ignored edges in T, then arbitrations will only occur if some edges are improperly oriented, so we also assume that (2) does not hold. Then T can contain exactly one improperly disoriented edge; or exactly one properly disoriented edge and some improperly oriented edges; or some ignored or some disoriented edges.

Prove Lemma 5.7-2 All interactions that occur in T are at disoriented edges, and these cause each such edge to progress along its depth-first trip through T.

Prove Lemma 5.7-3 If there is exactly one disoriented edge, and it is improper, then an arbitration will occur at the edge to turn it into a properly disoriented one, thus decreasing the number of unarbitrated edges by one.

Prove Lemma 5.7-4 If there is exactly one properly disoriented edge, then in at most $2\text{size}(T)$ interactions the disorientation must encounter an improperly oriented edge and cause an arbitration. Note that the disorientations could move out of T and a new one could enter—interactions are all that is important.

Prove Lemma 5.7-5 The final case occurs when there is more than one disoriented edge. By Lemma 5.3, there must be at most one more disoriented than ignored edges in T. By Lemma 5.6, the motions of the disoriented edges are occurring in disjoint portions of T connected by ignored edges. The only way that a disorientation can miss an ignored edge is for it to bounce off of an unarbitrated, improperly oriented edge, which results in an arbitration. If this does not happen, at most $2\text{size}(T)$ interactions are required before one of the disoriented edges cancels with an ignored edge.

Prove Lemma 5.7-6 In all cases, the number of ignored plus unarbitrated edges is reduced by one.

Proof of Lemma 5.7 □

Lemma 5.8 *In at most $O(\text{size}(T)^2)$ interactions within tree T with root vertex oriented away from the root edge, exactly one edge is properly disoriented and all other edges are properly oriented.*

Proof of Lemma 5.8 By Lemma 5.2, no ignored edges are ever created. Since resolving an ignored edge can require an arbitration, the number of ignored plus unarbitrated edges is bounded by $2\text{size}(T)$. By Lemma 5.7, after every $2\text{size}(T)$ interactions this number is reduced by one.

Proof of Lemma 5.8 □

Lemma 5.9 *Let G be a unicyclic network. Then the protocol deadlocks on G.*

Prove Lemma 5.9-1 **Proof of Lemma 5.9** We proceed by induction on the size of G, and suppose that the claim holds for all networks of smaller size.

Prove Lemma 5.9-2 Suppose for contradiction that some particular execution of the protocol does not deadlock on G. Then there is at least one disoriented edge in G, and by Lemma 5.4, there is an equal number of ignored edges.

Prove Lemma 5.9-3 Since ignored edges are never created, there must be an ignored edge $e = ((v,i),(w,j))$ that existed at the beginning of the execution and that will exist forever. So the processors at both ends of e never orient toward e.

Prove Lemma 5.9-4 Suppose that the edge e is on the ring of G. We cut the network at edge e, and add a leaf vertex u with edge $((u,0),(w,j))$ to create a tree T with root vertex v and root edge e. Since neither v nor w orient toward edge e, the particular execution of the protocol must also fail to deadlock when projected onto T.

Prove Lemma 5.9-5 But, since vertex v is oriented away from e, by Lemma 5.7, eventually T will contain one properly disoriented edge, and all others will be properly oriented. This properly disoriented edge must eventually move toward e, and so e cannot remain ignored. This contradicts the choice of e.

Prove Lemma 5.9-6 Thus, e must be inside a tree. It must connect a subtree T_w of size at least 2 to the rest of G. (T_w cannot be a leaf, because then e would not be ignored.) So we can cut the network at edge e, and add a leaf vertex u with edge $((u,0),(w,j))$ to create a new network G'. Vertex w never orients toward this new edge, so the particular execution of the protocol behaves the same when projected onto G', and so must not deadlock. But G' is smaller than G, and so this contradicts the inductive assumption.

Prove Lemma 5.9-7 Thus, the protocol always deadlocks on G.

Proof of Lemma 5.9 □

Lemma 5.10 *Let G be a unicyclic network. Then in at most $O(\text{size}(G)^2)$ interactions between processors of G, the protocol deadlocks.*

Prove Lemma 5.10-1 **Proof of Lemma 5.10** Consider a possible serialization of a protocol execution on G, and consider the edge e of the ring of G that remained ignored for the longest time. So long as e is ignored, the protocol is behaving as if it is on a tree. So by Lemma 5.8, this e could have remained ignored for at most $O(\text{size}(G)^2)$ interactions. After these interactions, no edges of the cycle are ignored, and since there are exactly as many processors as edges on the cycle, the cycle is oriented.

Prove Lemma 5.10-2 Any remaining ignored edges occur in subtrees of G, and further interactions cannot involve processors on the ring of G, so these interactions are confined to subtrees.

Prove Lemma 5.10-3 Consider a subtree T, and its ignored edge e closest to the root. By Lemma 5.8, for the subtree S with root edge e, in $O(\text{size}(S)^2)$ interactions within S, every edge of S is properly oriented except for one properly disoriented edge f. By Lemma 5.7, edge f is covered by the ignored edge e, and in $O(\text{size}(T))$ interactions they will cancel.

Prove Lemma 5.10-4 Thus at most $O(\text{size}(G)^2)$ interactions occur before the protocol deadlocks.

Proof of Lemma 5.10 □

5-18 Thus, by Proposition 5.1 and Lemma 5.10 we have:

Proposition 5.11 *If the unicyclic network G is started in any state, then in $O(\text{size}(G)^2)$ interactions, the protocol deadlocks with the network oriented.*

6 The Orientation Protocol on the Standard Model

6-1 We now show how to implement the orientation protocol under the standard model with its read/write scheduling demon. Each processor runs the program of Figure 8. The program has an integer constant, *degree*, which specifies the degree of the processor.

6-2 This protocol is running the token-passing protocol of Section 3 on each edge. But the processor only interacts with the single processor that is its focus of attention. It suspends its side of the token-passing protocol that is being executed with its other neighbors. This suspension is invisible to the other side of the edge, as the processor goes through exactly the same sequence of mode changes (from R to P) that the token-passing protocol does.

6-3 Now the fact that the processor only gives attention to one neighbor at a time does not affect the fact the token-passing protocol on the edge will, if given enough attention, eventually self-stabilize in $O(1)$ steps. Once stabilized, the modes at the ends of an edge between processors A and B will go through the following cycle:

$$WP,\ RP,\ RW,\ (A \text{ suspends}),\ PW,\ PR,\ WR,\ (B \text{ suspends}),\ WP,\ \ldots$$

where the possibility exists that the protocol can be suspended by a processor turning its focus elsewhere. Note that while the focus is elsewhere, the

$degree$: integer constant (Degree of this processor)

$focus$: $\{0, 1, \ldots, degree - 1\}$ (Current focus of attention)

$mode$: $\{R, W, P\}$ (Current mode of protocol at focus of attention)

$inport$: **array** $0..degree - 1$ **of** $\{R, W, P\}$

$outport$: **array** $0..degree - 1$ **of** $\{R, W, P\}$

do forever {

(Read Phase)

read $inport[focus]$;

(Write Phase)

(Compute mode and focus transition from the table below)

Next Mode		Input W	Input P	Input R
Current Mode	W	(W, P, R)	R	W
	P	P	(W, P, R)	W
	R	$\begin{bmatrix} P \\ \text{increment } focus \end{bmatrix}$	R	(W, P, R)

```
if mode = inport[focus]) {
    mode := random{W, P, R};
} else if (mode = W ∧ inport[focus] = P) {
    mode := R;
} else if (mode = P ∧ inport[focus] = R) {
    mode := W;
} else if (mode = R ∧ inport[focus] = W) {
    mode := P;
    focus := focus + 1 mod degree;
}
write mode to outport;
```

}

Figure 8: Low-level orientation protocol

processor at the other end of the edge cannot advance in the token-passing protocol and so must wait for focus to return.

6-4 We now map the states of an edge under the token-passing protocol onto the states of an edge under the high-level protocol. This is done by mapping P at the low level to P at the high level, and W, R at the low level to W at the high level. The focus of the processor at the low level corresponds to the orientation of the processor at the high level.

6-5 For the stabilized token-passing protocol, we get the following behavior at an edge.

Level							
Low	WP	RP	RW	PW	PR	WR	WP
High	$(\stackrel{W}{\rightarrow})(\stackrel{P}{\leftarrow})$	$(\stackrel{W}{\rightarrow})(\stackrel{P}{\leftarrow})$	$(\stackrel{P}{\leftarrow})(\stackrel{W}{\leftarrow})$	$(\stackrel{P}{\rightarrow})(\stackrel{W}{\leftarrow})$	$(\stackrel{P}{\rightarrow})(\stackrel{W}{\leftarrow})$	$(\stackrel{W}{\rightarrow})(\stackrel{P}{\rightarrow})$	$(\stackrel{W}{\rightarrow})(\stackrel{P}{\leftarrow})$

Stabilization at the low level exactly corresponds to arbitration of improperly disoriented edges at the high level.

6-6 Observe that when the network is oriented, so is every edge: therefore the token-passing protocol is suspended at each edge, and it is impossible for the scheduler to make any progress. In the high-level protocol, orientation may be achieved even when an edge is improperly oriented. Similarly, at the low level, orientation may be achieved even before the token-passing protocol has stabilized at an edge. Thus:

Theorem 6.1 *Under the read/write scheduling demon, there exists a uniform orientation protocol that will self-stabilize on any unicyclic network G into an oriented configuration within $O(\mathrm{size}(G)^2)$ expected number of progress-making processor activations.*

7 Acknowledgments

7-1 We wish to thank Amos Israeli for showing us that the problem was interesting, Joe Culberson for much stimulating discussion and for catching subtle bugs in our earlier versions, and the referees for their extraordinarily thorough work and excellent suggestions.

Appendix: Supplementary Code

A-1 The journal archive containing this article also contains files providing an executable program to explore the state space of the token-passing algorithm in Figure 8, verifying that the system is self-stabilizing.

- *readme.txt* is a brief description of the files below.
- *explore.c*, *explore.h* implement a general-purpose state-exploration algorithm.
- *problem.h* provides the functional interface to the state transitions for a given problem.
- *problem.c* implements the state transitions for the particular token-passing algorithm of Figure 8.
- *Makefile* compiles the programs above, producing an executable file called *explore*. On most UNIX systems you need only type *make*.
- *explore.out* is the output of *explore*.

Acknowledgement of support: H. James Hoover's research was supported by the Natural Sciences and Engineering Research Council of Canada, grant OGP 38937. Piotr Rudnicki's reseach was supported by the Natural Sciences and Engineering Research Council of Canada, grant OBP 09207.

References

[AEY91] Efthymios Anagnostou and Ran El-Yaniv. More on the power of random walks: uniform self-stabilizing randomized algrithms (preliminary report). In S. Toueg, P. G. Spirakis, and L. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579 of *Lecture Notes in Computer Science*, pages 31–51. Springer-Verlag, October 1991.

[ASW88] H. Attiya, M. Snir, and M. K. Warmuth. Computing on an anonymous ring. *Journal of the ACM*, 35(4):845–875, October 1988.

[BGW89] G. M. Brown, Mohamed G. Gouda, and C. L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38(6):845–852, 1989.

[BP89] James E. Burns and Jan Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, April 1989.

[CGR87] Ernest J. Chang, Gaston H. Gonnet, and Doron Rotem. On the costs of self-stabilization. *Information Processing Letters*, 24(5):311–316, March 1987.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[Dij86] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[DIM94] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform self-stabilizing leader election part 1: complete graph protocols. Technical Report CS807, Technion, May 1994.

[DIM95] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Analyzing expected time by scheduler-luck games. *IEEE Transactions on Software Engineering*, 21(5):429–439, May 1995.

[Gho91] Sukumar Ghosh. Binary self-stabilization in distributed systems. *Information Processing Letters*, 40(3):153–159, November 1991.

[Her90] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, June 1990.

[Her92] Ted Herman. Self-stabilization: randomness to reduce space. *Distributed Computing*, 6(2):95–98, 1992.

[Hoe94] J. H. Hoepman. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. In Gerard Tel and Paul Vitanyi, editors, *Distributed Algorithms, 8th International Workshop*, volume 857 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, 1994.

[HR91] H. James Hoover and Piotr Rudnicki. The uniform self-stabilizing orientation of unicyclic networks. Technical Report TR 91–02, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G2H1, August 1991.

[IJ90] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *9th Annual ACM Symposium on Principles of Distributed Computation*, pages 119–131, August 1990.

[IJ93a] Amos Israeli and Marc Jalfon. Modular construction of uniform self-stabilizing protocols. Technical report, Technion, June 1993.

[IJ93b] Amos Israeli and Marc Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104(2):175–196, 1993.

[Kes87] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29(1):39–42, September 1987.

[KP93] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[Kru79] H. S. M. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8(2):91–95, February 1979.

[Lam86a] Leslie Lamport. The mutual exclusion problem: part I—a theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.

[Lam86b] Leslie Lamport. The mutual exclusion problem: part II—statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[LH91] Edward A. Lycklama and Vassos Hadzilacos. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Transactions on Programming Languages and Systems*, 13(4):558–576, October 1991.

[Sch93] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[SP87] V. Syrotiuk and J. Pachl. A distributed ring orientation algorithm. In J. van Leeuwen, editor, *Distributed Algorithms, 2nd International Workshop*, pages 332–336. Springer-Verlag, July 1987.

[TH95] Ming-Shin Tsai and Shing-Tsaan Huang. Self-stabilizing ring orientation protocols. In *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, pages 16.1–16.14, May 1995. The proceedings appeared as an unnumbered Technical Report, Department of Computer Science, University of Nevada Las Vegas, Box 454019, Las Vegas, NV, 89154-4019.