On Relaxing Interleaving Assumptions

James E. Burns*

Mohamed G. Gouda†

Raymond E. Miller‡

Abstract

In verifying concurrent systems, it is frequently convenient to assume that any operations that might occur concurrently are actually executed in some serial order. We show that for some systems this (sometimes unrealistic) assumption is unnecessary and simultaneous actions can be allowed without affecting correctness.

In 1974, Dijkstra [*Comm. ACM*, 17 (1974), pp. 643–644] presented the problem of self-stabilization and gave three solutions. In the two solutions using only a constant number of states per processor, Dijkstra only claimed correctness under the condition that simultaneous actions were prohibited. Several other authors have presented solutions that do not require this condition, but, until now, apparently no one has noticed that Dijkstra's solutions are correct even without the condition. The techniques developed in this paper allow us easily to extend the correctness of Dijkstra's solutions to the case where simultaneous actions are allowed.

Key words. Self-stablization, fault-tolerance, distributed computing

AMS subject classifications. 68M15, 68N25, 68Q10

*School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332-0280

†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712-1188

‡Department of Computer Science, University of Maryland, College Park, MD 20742

1 Introduction

Whenever we design a solution to a problem, we prefer to make the weakest assumptions possible about the environment in which it operates, so that the correctness of our solution is as robust as possible. In concurrent systems, it is common to assume that all computations can be described as a serial ordering (interleaving) of elementary operations. This is a reasonable assumption in many cases, but it is clearly not as weak as possible. In this paper we consider concurrent systems in which elementary operations can take place simultaneously. By "simultaneous," we mean the synchronous execution of two or more actions, rather than an arbitrary overlapping of actions. The main question of interest is whether computations of the "parallel" system are essentially different from those of the serial system (which is presumed to be correct). Our main results give conditions under which certain important properties of concurrent systems hold for parallel systems if they hold for the corresponding serial systems. This will often provide simpler proofs for parallel systems by allowing us to reason about the simpler serial systems.

Almost all previous work on concurrent systems is based on models that involve interleaving elementary operations. For example, in the theory of concurrent databases [Pap86], the concurrent execution of transactions is modeled by considering each transaction as a sequence of elementary, atomic operations. These elementary operations are then interleaved to represent overlapping transactions. Note that the elementary operations themselves are assumed

to be executed in some particular order and not to occur simultaneously (or, equivalently, simultaneous operations have the same effect as executions of the operations in some order). Similarly, in Petri net theory [Pet81], simultaneous actions (firings) are allowed only if their actions to not interfere, so that simultaneous firings are equivalent to firing the actions in any order. (In fact, prohibiting firings that interfere with one another is a major feature of Petri nets.)

To illustrate how our definition differs from standard models based on interleaving, consider the following example. Suppose a system has three processors, P_0, P_1, P_2, and three variables, V_0, V_1, V_2. Each processor P_i has a single atomic operation:[1]

$$V_i \leftarrow V_{i-1 \pmod 3} + V_i + V_{i+1 \pmod 3}$$

We want to allow simultaneous execution of two such transitions. For example, from a configuration $\langle 0, 1, 2 \rangle$ (representing $\langle V_0, V_1, V_2 \rangle$), let both P_0 and P_1 execute simultaneously. The resulting configuration should be $\langle 3, 3, 2 \rangle$. This is impossible to achieve by executing transitions by P_0 and P_1 in either possible interleaved order. We could achieve the effect by using more primitive atomic operations, say by defining P_i as follows, where L_i is a local variable and loc_i

[1] We use 'mod' as an operator giving remainder of the first operand when divided by the second, rather than the mathematical meaning of modulo congruence.

is a location counter (necessary for locally sequential operations):

$$\textbf{if } loc_i = 0 \textbf{ then } L_i \leftarrow V_{i-1 \pmod 3} + V_{i+1 \pmod 3};\ \ loc_i \leftarrow 1 \textbf{ fi}$$

$$\textbf{if } loc_i = 1 \textbf{ then } V_i \leftarrow V_i + L_i;\ \ loc_i \leftarrow 0 \textbf{ fi}$$

The desired effect is then obtained by interleaving steps in the order P_0, P_1, P_0, P_1 (where the location counters are initially zero). Unfortunately, we also get undesired effects (computations that could not occur in the original system). For example, executing steps in the order: $P_0, P_1, P_0, P_2, P_2, P_1$, gives a result of $\langle 3, 3, 6 \rangle$, which is not a reachable configuration in the original system.

One place where simultaneous actions have been explicitly considered is the case of concurrent-read, concurrent-write parallel random access machines [FW78, FRW84]. However, the assumption normally made with this model are usually equivalent to assuming that the simultaneous actions occur in some interleaved order. We are interested in cases where this does not hold.

Lamport [Lam86] has considered systems where operations can overlap in more complex ways than we consider here. In particular, operations effectively take place over intervals, so that, for example, one operation could overlap two others that do not overlap each other. This possibility is not allowed in the model that we are considering, since we take operations to be atomic (effectively instantaneous). We take advantage of this restriction to obtain simple ways of showing that parallel computations in our model are equivalent to serial ones.

Section 2 gives our definitions for serial and parallel semantics of con-

current systems. Section 3 presents our main results regarding reachability. Section 4 applies our results to Dijkstra's self-stabilization protocols.

2 Serial and Parallel System Semantics

A **concurrent system** $S = (P, V)$ consists of a finite set of **processors** $P = \{P_0, P_1, \ldots, P_n\}$ and a finite set of **variables** $V = \{V_0, V_1, \ldots, V_m\}$. (Throughout the paper, S, P and V will be used as described here.) The **configurations of** S constitute the set $\Gamma = V_0 \times V_1 \times \cdots \times V_m$, where V_j here is the set of values taken on by variable V_j. Let $\gamma = (v_0, v_1, \ldots, v_m)$ be a configuration of S. Then for any $V_j \in V$, the **value of** V_j **at** γ is $V_j(\gamma) = v_j$.

Each system has an associated transition relation, $\rightarrow$, which is a subset of $\Gamma \times \Gamma$. The particular form of $\rightarrow$ depends on the semantics of the system, which we do not want to restrict except as follows. Every transition $\gamma \rightarrow \gamma'$ (called a **step** of S) has an associated nonempty set of processors, $\alpha \subseteq P$. We denote this association by $\gamma \xrightarrow{\alpha} \gamma'$. It is possible that another processor set $\alpha' \neq \alpha$ could be associated with the same transition (so that $\gamma \xrightarrow{\alpha'} \gamma'$) and that α could also be associated with a step to some $\gamma'' \neq \gamma'$ (so that $\gamma \xrightarrow{\alpha} \gamma''$). If α is a singleton set, say $\alpha = \{P_i\}$, we write $\gamma \xrightarrow{i} \gamma'$ and call this transition a **serial step** of S. (A step by multiple processors could be called a parallel step, and a step by all processors together could be called a synchronous step.)

A **computation of** S **from** $\gamma_0 \in \Gamma$ is a finite or infinite sequence $C = \langle \gamma_0, \gamma_1, \ldots \rangle$ of configurations in Γ such that $\gamma_{i-1} \rightarrow \gamma_i$ for all integers i, $0 <$

$i < length(C)$.[2] A computation is **serial** if every step in the computation is serial. A computation is **maximal** if it is infinite or it is finite and the final configuration no possible step leading from it.

For some of our results, we require more structure on the form of a system. A system $S = (P, V)$ **without multi-writer variables** has the following special characteristics:

1. Each processor $P_i \in P$ has an associated set of irreflexive partial functions A_i called **actions** from Γ to Γ. If $\gamma \xrightarrow{i} \gamma'$ is a step then $(\gamma, \gamma') \in a$ for some $a \in A_i$. Since actions are functions, we can write $P_i^a(\gamma) = \gamma'$ in this case.

2. V is partitioned into $Own_0, Own_1, \ldots, Own_n$ such that if $\gamma \xrightarrow{i} \gamma'$ and $V_j(\gamma) \neq V_j(\gamma')$ then $V_j \in Own_i$. Let $Own_i[\gamma]$ denote the projection of γ onto the variables in Own_i.

3. Let $\alpha \subseteq P$. Then $\gamma \xrightarrow{\alpha} \gamma'$ if and only if for every $i \in \alpha$ there is a γ_i such that $\gamma \xrightarrow{i} \gamma_i$ and $Own_i[\gamma'] = Own_i[\gamma_i]$, and for every $j \in V - \cup_{i \in \alpha} Own_i$, $V_j(\gamma') = V_j(\gamma)$.

(1) means that every serial step can be attributed to some particular action by a processor. (2) requires that no variable can be changed by two distinct processors. (3) defines the system transition relation so that if several processors move together in the same step, they all change their own variables as if they had moved alone. This is unambiguous because there are no multi-writer variables.

[2] If C is finite then $length(C)$ is just the number of elements in C. If C is infinite then $length(C)$ stands for a value, often denoted by ω, greater than all integers.

The correctness of a concurrent system is determined by its behavior over a set of computations. Usually, these computations are determined by an initial configuration or set of configurations, by whether computations are serial or not, and perhaps by some additional criteria (such as fairness). For our purposes, we will say a system is **serial-correct** if it is correct for those serial computations that are allowed by the problem specification. On the other hand, a system that is correct when the computations are extended to allow simultaneous actions is **parallel-correct**. The next section provides some tools for showing when a serial-correct system is also parallel-correct.

3 Lemmas on Reachability and Eventuality

A computation **from** γ **to** γ' is a finite computation with initial configuration γ and final configuration γ'. A configuration γ' is **(serially) reachable** from configuration γ if there is a (serial) computation from γ to γ'. For any $\gamma_0 \in \gamma$, define $\mathit{parallel}(\gamma_0)$ to be the subset of Γ that is reachable from γ_0. We define $\mathit{serial}(\gamma_0)$ analogously for serial reachability.

Reachability is often called a *safety* property. Another kind of property of interest is a *progress* (sometimes called *liveness*) property. Let Λ be a subset of Γ and $\gamma_0 \in \Gamma$. We say S **makes progress toward** Λ **from** γ_0 if every maximal computation from γ_0 contains an element of Λ, in which case $\mathit{eventually}(\Lambda, \gamma_0)$ holds. The analogous predicate for serial progress is $\mathit{serial\text{-}eventually}(\Lambda, \gamma_0)$.

In order to show that the desired condition (*serial-eventually*) is obtained,

we need another property of the set $\Lambda \subseteq \Gamma$. We say Λ is **(serially) closed** if whenever $\gamma \in \Lambda$ and $\gamma \rightarrow \gamma'$ is a (serial) step of S, then $\gamma' \in \Lambda$.

Our first lemma is an easy consequence of the definitions.

LEMMA 3.1 *Let $S = (P, V)$ be a concurrent system, $\gamma_0 \in \Gamma$, and Λ be a serially closed subset of Γ. If for every step $\gamma \rightarrow \gamma'$ of S there is a serial computation of S from γ to γ', then $serial(\gamma_0) = parallel(\gamma_0)$ and $serial\text{-}eventually(\Lambda, \gamma_0)$ if and only if $eventually(\Lambda, \gamma_0)$.*

Proof: Since serial computations are also parallel computations, trivially $serial(\gamma_0) \subseteq parallel(\gamma_0)$ and $eventually(\Lambda, \gamma_0)$ implies $serial\text{-}eventually(\Lambda, \gamma_0)$. It is easy to see by induction on the number of steps that any configuration reachable from γ_0 is also serially reachable, so $parallel(\gamma_0) \subseteq serial(\gamma_0)$.

Now suppose $eventually(\Lambda, \gamma_0)$ does not hold, so there is a maximal computation, C, from γ_0 that avoids Λ. We will construct a serial computation, C', that also avoids Λ. Since (not $eventually(\Lambda, \gamma_0)$) implies (not $serial\text{-}eventually(\Lambda, \gamma_0)$) is equivalent to $serial\text{-}eventually(\Lambda, \gamma_0) \Rightarrow eventually(\Lambda, \gamma_0)$, this completes the proof. Let C' be the corresponding serial computation, formed (inductively) by chaining together the serial computations between the adjacent configurations in C that exist by the assumption of the lemma. Suppose that C' does not avoid Λ, i.e., it contains a configuration in Λ. But Λ is serially closed, so all the remaining configurations in C' are in Λ. In particular, the next configuration in C' that is also an element of C is in Λ, contradicting the assumption that C avoids Λ. □

Note that the serial closure condition of the lemma is necessary. For example, consider a system with processors P_0 and P_1, and binary variables V_0 and V_1. Define the transitions of P_0 and P_1 as follows:

$$P_0: \quad V_0 \leftarrow \textbf{not } V_0$$

$$P_1: \quad V_1 \leftarrow \textbf{not } V_1$$

From an initial configuration with $V_0 = 0$ and $V_1 = 1$, there is a parallel computation that cycles forever through the configurations $\langle 0,1 \rangle$ and $\langle 1,0 \rangle$. Let Λ be the subset of Γ for which $V_0 = V_1$. Although the other conditions of the lemma hold, Λ is not serially closed, and even though *serial-eventually*(Λ, γ), it is not true that *eventually*(Λ, γ).

Unfortunately, Lemma 3.1 is difficult to apply, since it essentially requires consideration of a possibly infinite set of serial computations. A slightly more restricted version has simpler application. We say a transition $\gamma \xrightarrow{\alpha} \gamma'$ of S has a **linear connection** if there is a serial computation in S from γ to γ' consisting of exactly one step by each processor in α. In other words, a linear connection exists if there is some interleaving exactly one step of each processor involved in the transition that produces the same result. A concurrent system S is **linearly connected** if for every transition of S has a linear connection. In a linearly connected system, serial reachability between a pair of configurations can be checked by considering only a finite number of finite serial computations.

COROLLARY 3.2 *Let $S = (P, V)$ be a linearly connected concurrent system, $\gamma_0 \in \Gamma$, and Λ be a serially closed subset of Γ. Then serial(γ_0) = parallel(γ_0) and serial-eventually(Λ, γ_0) if and only if eventually(Λ, γ_0).*

Showing a system is linearly connected simplifies proving the properties that we want, but it could require a lot of work. If the way that processors can affect one another has a certain structure, then the desired properties are easier to check. One processor can only be affected by others if they change variables that alter the behavior of the processor. This notion is formalized (for systems without multi-writer variables) by the following definitions.

Let $S = (P, V)$ be a concurrent system without multi-writer variables. We need some preliminary definition in order to define a dependency relation between processors at a given configuration. Let γ, γ' be configurations and P_i be a processor. The **action set** of P_i at γ is the set of actions of P_i that are defined at γ, which we denote by $A_i(\gamma)$. P_i is **enabled** at γ if its action set at γ is not empty. The **effect** of P_i is **different** at γ and γ' if either $A_i(\gamma) \neq A_i(\gamma')$ or there is some $a \in A_i(\gamma)$ such that $Own_i[P_i^a(\gamma)] \neq Own_i[P_i^a(\gamma')]$. Thus the effect of a processor is different if some actions have been added or subtracted from its action set or if the same action produces different results.

Now let $P_i, P_k \in P$ be distinct processors. We say P_i **depends on** P_k **at** $\gamma \in \Gamma$ if there are sets α and α' of processors and configurations γ' and γ'' such that

- α contains P_k but not P_i.

- $\gamma \xrightarrow{\alpha} \gamma'$.
- If α is a singleton set, then $\gamma'' = \gamma$; otherwise, $\gamma \xrightarrow{\alpha'} \gamma''$ where $\alpha' = \alpha - \{k\}$.
- The effect of P_i is different at γ' and γ''.

Thus, P_i depends on P_k if the participation of P_k in a step can alter the effect of a step by P_i. Of course, we are really more interested in cases where one processor *cannot* affect another.

For the next lemma, we introduce a digraph that corresponds to the dependency relation on the processors at a configuration. Let γ be a configuration of a system S and α be a subset of P. The **dependency digraph of α at γ**, $D_\alpha(\gamma)$, is the directed graph with the processors of S as vertices and edge set

$$\{(P_i, P_j) \in \alpha \times \alpha : P_i \neq P_j,\ P_i \text{ is enabled at } \gamma, \text{ and } P_i \text{ depends on } P_j \text{ at } \gamma\}.$$

If $D_P(\gamma)$ is acyclic we say γ is **acyclic**; otherwise, γ is **cyclic**. If $D_\alpha(\gamma)$ is acyclic, define a **root of α at γ** to be a vertex with in-degree zero in $D_\alpha(\gamma)$. (It is well-known that in any finite, acyclic graph there must be a vertex with in-degree zero, so a root always exists.)

The following preliminary lemma shows that every configuration reachable in one step from an acyclic configuration is also serially reachable if there are no multi-writer variables. (Note that the restriction on multi-writer variables is necessary. For example, consider a system with processors P_0, P_1 and variables

V_0, V_1, and V_2. Suppose P_0 has transition $V_1 \leftarrow V_0$; $V_2 \leftarrow V_0$ and P_1 has transition $V_2 \leftarrow V_1$. If the initial (acyclic) configuration is $\langle 0,1,2 \rangle$, a non-serial step can reach $\langle 0,0,1 \rangle$, but no serial computation can.)

LEMMA 3.3 *Let $S = (P, V)$ be a concurrent system without multi-writer variables and γ be an acyclic configuration in Γ. If $\gamma \rightarrow \gamma'$ is a transition of S, then there is a linear connection from γ to γ'.*

Proof: Let $\gamma \xrightarrow{\alpha} \gamma'$ be a transition of S with $|\alpha| = k$, and for each $i \in \alpha$ let a_i be the action of processor P_i that occurs in the transition. Define $\gamma_1, \gamma_2, \ldots, \gamma_{k+1}$ and $\alpha_1, \alpha_2, \ldots, \alpha_{k+1}$ as follows.

- $\gamma_1 = \gamma$ and $\alpha_1 = \alpha$.
- For $1 \leq j \leq k$, $\alpha_{j+1} = \alpha_j - \{P_{i_j}\}$ and $\gamma_{j+1} = P_{i_j}^{a_{i_j}}(\gamma_j)$, where P_{i_j} is a root of α_j at γ_j. (Note: we will show that γ_j is acyclic, so that there is always a root of α_j at γ_j.

The proof proceeds by induction on j in the range 1 to $k+1$ with the following induction hypothesis:

1. The effect of $P_{i_1}, \ldots, P_{i_{j-1}}$ in reaching γ_j is the same as if the processors had acted in parallel. That is, $\gamma \xrightarrow{\alpha'} \gamma_j$, where $\alpha' = \{P_{i_1}, \ldots, P_{i_{j-1}}\}$ and the actions of the processors in α' are $a_{i_1}, a_{i_2}, \ldots, a_{i_{j-1}}$, respectively.
2. $D_{\alpha_j}(\gamma_j)$ is acyclic, so P_{i_j} exists.

Condition (1) holds vacuously for $j = 1$ and condition (2) follows from a premise of the lemma, so the basis of the induction holds. Suppose the

hypothesis holds for $1, 2, \ldots, j$ for some j, $1 \leq j \leq k$. In particular, α_j is acyclic and P_{i_j} is a root of α_j at γ_j. Suppose the effect of P_{i_j} is different from γ_j than from γ. Then there is an ℓ, $1 \leq \ell < j$ such that the effect of P_{i_j} is different at γ_ℓ and $\gamma_{\ell+1}$. But this would imply that P_{i_j} depends on P_{i_l} at γ and so P_{i_l} is not a root of α_l at γ_l, contradicting the inductive assumption. Therefore, the effect of P_{i_j} is the same from γ and γ_j. Since no variable is multiple writer, this implies condition (1) holds for $j+1$.

We still need to show that $D_{\alpha_{j+1}}(\gamma_{j+1})$ is acyclic. Using a similar argument to that in the previous paragraph, it can be seen that the effect of every processor in α_{j+1} is the same at γ_{j+1} as at γ. Thus, $D_{\alpha_{j+1}}(\gamma_{j+1})$ is just the digraph induced by the vertices in α_{j+1} on $D_\alpha(\gamma)$. Since $D_\alpha(\gamma)$ is acyclic, so is $D_{\alpha_{j+1}}(\gamma_{j+1})$. □

LEMMA 3.4 *Let $S = (P, v)$ be a concurrent system without multi-writer variables, $\gamma_0 \in \Gamma$, and Λ be a serially closed subset of Γ. If every configuration $\gamma \in parallel(\gamma_0)$ is acyclic, then $serial(\gamma_0) = parallel(\gamma_0)$ and serial-eventually(Λ, γ_0) if and only if eventually(Λ, γ_0).*

Proof: Since every configuration is acyclic, Lemma 3.3 implies that every transition of S has a linear connection. Thus, S is linearly connected and Corollary 3.2 applies, giving the result. □

When the last lemma does not apply directly, it is sometimes possible to show that the property of being acyclic eventually holds. Let $Acyc_S$ be the set of acyclic configurations of S. If *eventually*$(Acyc_S, \gamma_0)$ holds and $Acyc_S$ is

serially closed, then we can still show that *serial eventually* and *eventually* are equivalent for S.

LEMMA 3.5 *Let $S = (P, V)$ be a concurrent system without multi-writer variables, $\gamma_0 \in \Gamma$, and Λ be a serially closed subset of Γ. If eventually$(Acyc_S, \gamma_0)$ and $Acyc_S$ is closed, then serial-eventually(Λ, γ_0) if and only if eventually(Λ, γ_0).*

Proof: Since *eventually*$(Acyc_S, \gamma_0)$, in any maximal computation from γ_0 an acyclic configuration, γ, is eventually reached. Since $Acyc_S$ is closed, all configurations reachable from γ are acyclic. The result then follows from Lemma 3.3 and an argument similar to that in the proof of Lemma 3.1. □

4 Applications to Self-Stabilization

Some of the results of the previous section are illustrated here by application to a well-known problem: self-stabilization. Self-stabilization was originally defined by Dijkstra in 1974 [Dij74]. For the two solutions that use only a constant number of states per processor, Dijkstra only claimed his solutions to be serial-correct. Several other authors have given solutions that are parallel-correct [BGW89, Bur87, Tch81], but apparently no one has noticed before that Dijkstra's solutions also happen to be parallel-correct, as we show in this section[3].

[3]Note that some problems do have serial-correct solutions but no parallel-correct solutions; see for example [BP89].

In the self-stabilization problem for rings, there are $n + 1$ processors, $P_0, P_1, \ldots, P_n$, connected in a ring. Each processor behaves as a finite state machine with transitions that depend on its own state and those of its two neighbors in the ring. In our model, variables $V_0, V_1, \ldots, V_n$ are used to hold the states of the processors. There are no other variables, so, since each processor can only change its own state, the system has no multi-writer variables, and Lemma 3.4 applies.

A correct solution to the self-stabilization problem requires that a closed, proper subset Λ of Γ, called the **legitimate configurations**, be given such that there is only one enabled processor in any legitimate configuration. We also require that the system cannot deadlock (some processor is enabled at every configuration), and that a fairness condition holds: every processor is enabled infinitely often in any infinite computation consisting of legitimate configurations. The final required property, which is the essence of self-stabilization, is that a legitimate configuration will be reached from any starting configuration within a finite number of steps. Thus, a system satisfying the other conditions is serial-correct if for all $\gamma \in \Gamma$, *serial-eventually*(Λ, γ). Parallel correctness requires that *eventually*(Λ, γ) holds for all $\gamma \in \Gamma$. We want to show that *serial-eventually*(Λ, γ) implies *eventually*(Λ, γ) for Dijkstra's solutions, so that serial correctness implies parallel correctness.

In the next subsections, we present Dijkstra's solutions in our own notation. In describing the transition functions, the state of P_i is designated by V_i.

The left neighbor of P_i is $P_{i-1 \pmod{n+1}}$ and the right neighbor is $P_{i+1 \pmod{n+1}}$. In the following discussions, recall that the state of P_i at configuration γ is denoted $V_i(\gamma)$.

4.1 The Three State Solution.

The three state solution is denoted by *S3*. Each processor can be in one of three states: 0, 1, or 2. The transitions of *S3* are given by the following single actions for P_0 and P_n and by pairs of actions for P_i, $0 < i < n$. It should be apparent that this is a system without multi-writer variables since P_i only changes V_i. (Note: we have transliterated Dijkstra's original algorithm, replacing the local state he denoted by 'S' by V_i, the state of the left processor 'L' by V_{i-1}, and the state of the right processor 'R' by V_{i+1}.)

For processor P_0:
if $(V_0 + 1) \bmod 3 = V_1$ **then** $V_0 \leftarrow (V_0 - 1) \bmod 3$ **fi**

For processor P_i, $0 < i < n$:
if $(V_i + 1) \bmod 3 = V_{i-1}$ **then** $V_i \leftarrow V_{i-1}$ **fi**
if $(V_i + 1) \bmod 3 = V_{i+1}$ **then** $V_i \leftarrow V_{i+1}$ **fi**

For processor P_n:
if $V_{n-1} = V_0$ **and** $(V_{n-1} + 1) \bmod 3 \neq V_n$ **then** $V_n \leftarrow (V_{n-1} + 1) \bmod 3$ **fi**

The set of legitimate configurations, $\Lambda \subseteq \Gamma$, is the set of all configurations such that exactly one processor is enabled.

Suppose that γ is a cyclic configuration. Since P_0 can only depend on P_1 and the other processors can only depend on their neighbors in the ring,

the dependency graph of γ either has an $n+1$ cycle or a 2-cycle. But since if P_n depends on P_0 it also depends on P_{n-1}, we only need to consider 2-cycles. Suppose P_{i-1} and P_i depend on one another for some i, $0 < i \leq n$. (We need not consider $i = 0$ because P_0 cannot depend on P_n.) Since P_i depends on P_{i-1} at γ, we have $(V_{i-1}(\gamma) + 1) \bmod 3 \neq V_i(\gamma)$ (this follows directly if $i = n$ or indirectly from $(V_i(\gamma) + 1) \bmod 3 = V_{i-1}(\gamma)$ otherwise). But since P_{i-1} also depends on P_i at γ, we have $(V_{i-1}(\gamma) + 1) \bmod 3 = V_i(\gamma)$. These assertions are mutually contradictory, so there cannot be a 2-cycle in the dependency graph of γ for any $\gamma \in \Gamma$.

Since every configuration is acyclic, by Lemma 3.4, if *S3* is serial-correct then it is also parallel-correct. Therefore, a demon is not essential to the correctness of *S3*, and Dijkstra's proof of $S3$ [Dij86] applies to parallel as well as serial systems. (Dijkstra assumed the existance of a *demon* that would arbitrate between two adjacent processors that were simultaneously enabled. Thus, he did not have to reason about executions that could occur when neighbors moved simultaneously. Our results show this assumption was unnecessary.)

4.2 The Four State Solution.

The four state solution, *S4*, has a pair of Boolean variables for each processor which together constitute its state. For P_i, we denote these by V_i and W_i, which correspond to variables x and *up*, respectively, in Dijkstra's original

presentation. W_0 and W_n are not actually variables. W_0 is never referred to and W_n is taken to be a constant with the value '**false**.' The actions of the processors are given below. Again, this is clearly a system without multi-writer variables.

For processor P_0:
if $V_0 = V_1$ **and not** W_1 **then** $V_0 \leftarrow$ **not** V_0 **fi**

For processor P_i, $0 < i < n$:
if $V_i \neq V_{i-1}$ **then** $V_i \leftarrow$ **not** V_i; $W_i \leftarrow$ **true fi**
if $V_i = V_{i+1}$ **and** V_i **and not** W_{i+1} **then** $V_i \leftarrow$ **false fi**

For processor P_n:
if $V_n \neq V_{n-1}$ **then** $V_n \leftarrow$ **not** V_n **fi**

As in *S3*, the legitimate configurations are those containing exactly one enabled processor.

Since P_0 and P_n cannot depend on one another, if there is any cycle, there must be a 2-cycle. But, for $0 \leq i < n$, if P_i depends on P_{i+1} at γ, then $V_i(\gamma) = V_{i+1}$, while if P_{i+1} depends on P_i then $V_{i+1} \neq V_i(\gamma)$, which is impossible. (We need not consider $i = n$ since P_n cannot depend on P_0.) Therefore, there is no 2-cycle in any configurations, so Lemma 3.4 applies. Once again, serial correctness implies parallel correctness, so a demon is unnecessary.

4.3 The K State Solution.

The K state solution, *SK*, uses states 0, 1, ..., $K - 1$, and the following transitions.

For processor P_0:
if $V_n = V_0$ **then** $V_0 \leftarrow (V_0 + 1) \bmod K$ **fi**

For processor P_i, $0 < i \leq n$:
if $V_{i-1} \neq V_i$ **then** $V_i \leftarrow V_{i-1}$ **fi**

As for the other solutions, the legitimate configurations are those with exactly one enabled processor.

The correctness of the solution depends on the value of K relative to n. When $K = n$, the system is serial-correct, but not parallel-correct, because there is a cycle of illegitimate configurations. It has long been known that the system is parallel-correct when $K > n$. One of the difficulties in the proof of this fact is showing that the system is parallel-correct if it is serial-correct. Our techniques provide a simple way to do this.

To show that SK is parallel correct when $K > n$, we will apply Lemma 3.5. To do this, we must show that the $Acyc_{SK}$ is closed and *eventually-parallel*$(Acyc_{SK}, \gamma)$ holds for all $\gamma \in \Gamma$.

Since processor P_i can only depend on $P_{i-1 \,(\mathrm{mod}\, n+1)}$, the only possibility that a configuration is cyclic is if there is a cycle of all $n+1$ processors. Let $\gamma \rightarrow \gamma'$ be a transition of SK and assume that γ' is cyclic. If P_i moves in the transition, but $P_{i-1 \,(\mathrm{mod}\, n+1)}$ does not, then P_i is not enabled at γ', and so γ' could not be cyclic. Thus, since some processor must move in the transition, they all must move. This implies that all processors are enabled at γ and hence γ is cyclic. Therefore an acyclic configuration cannot be transformed

into a cyclic configuration, so $Acyc_{SK}$ is closed.

Assume that *eventually-parallel*$(Acyc_{SK}, \gamma)$ does not hold for SK and some $\gamma \in \Gamma$. Since there are only a finite number of configurations, there is a cycle of configurations of SK

$$\gamma_0 \rightarrow \gamma_1 \rightarrow \cdots \rightarrow \gamma_{k-1} \rightarrow \gamma_k = \gamma_0$$

such that for $0 \leq j \leq k$, γ_j is cyclic. We can assume without loss of generality (by using multiple copies of the cycle if necessary) that $k \geq n$. Let $s = V_0(\gamma_0)$. Then by the actions of P_0, $V_0(\gamma_1) = s + 1 \bmod K$, $V_0(\gamma_2) = s + 2 \bmod K$, ..., $V_0(\gamma_n) = s + n \bmod K$. Also, by the actions of $P_1, P_2, \ldots, P_n$, $V_1(\gamma_1) = s$, $V_2(\gamma_2) = s$, ..., $V_n(\gamma_n) = s$. Since P_0 is enabled at γ_n, we also have $V_0(\gamma_n) = V_n(\gamma_n)$ so that $s = s + n \bmod K$. But this is impossible since $K > n$. Therefore, Lemma 3.5 applies, so serial correctness implies parallel correctness, and a demon is unnecessary in this case.

5 Concluding Remarks

We have derived several lemmas that can be used to show that the correctness of a serial system will not be affected by allowing simultaneous execution of atomic operations. The utility of these lemmas has been demonstrated by applying them to Dijkstra's self-stabilization protocols, showing that demons are unnecessary for the three and four state solutions and (as was previously known) that a demon is unnecessary for the K state solution if $K > n$. Our

techniques should also be useful in other problems where the model of computation developed here is applicable.

References

[BGW89] G.M. Brown, M.G. Gouda, and C.-L. Wu, *Token systems that self-stabilize*, IEEE Trans. Comp. 38, (1989), pp. 845–852.

[Bur87] J.E. Burns, *Self-stabilizing rings without demons*, Technical Report GIT-ICS-87/36, School of Information and Computer Science, Georgia Institute of Technology, Nov. 1987.

[BP89] J.E. Burns and J. Pachl, *Uniform self-stabilizing rings*, ACM Trans. Prog. Lang. and Systems 11, (1989), pp. 330–344.

[Dij74] E.W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, Comm. ACM 17, (1974), pp. 643–644.

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

[FRW84] F.E. Fich, P.L. Ragde, and A. Wigderson, *Relations between concurrent-write models of parallel computation*, in Proc. 3rd Annual ACM Symp. on Principles of Distributed Computing, Vancouver, BC, Canada, 1984, ACM, pp. 179–189.

[FW78] Fortune and Wylie. Parallelism in random access machines. nth ACM Symposium on Theory of Computing 1978, pp. 114–118.

[Lam84] L. LAMPORT, *Solved problems, unsolved problems, and non-problems in concurrency*, in Proc. of the Third Symp. on Principles of Distributed Computing, Vancouver, BC, 1984, ACM, pp. 1–11.

[Lam86] L. LAMPORT, *On interprocess communication, parts I and II*, Distributed Computing I, 2 (1986), pp. 77–85 and 86–101.

[Pap86] C. PAPADIMITRIOU, *The Theory of Database Concurrency Control*, Computer Science Press, Rockville, MD, 1986.

[Pet81] J.L. PETERSON, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

[Tch81] M. TCHUENTE, *Sur l'auto-stabilisation dans un réseau d'ordinateurs*, RAIRO Inf. Théor. 15, (1981), pp. 47–66.