MCC Technical Report Number STP-379-89

Proceedings of the MCC Workshop on Self-Stabilizing Systems

Michael Evangelist and Shmuel Katz

Software Technology Program

Classification: Non Proprietary

November 10, 1989

Abstract. This proceedings contains the papers presented at the MCC Workshop on Self-Stabilizing Systems, held in Austin on August 11, 1989 .

	contents	distribution			
✓	NON proprietary (White)		STP participants		STP Internal Only
	MCC proprietary (Light B) (P)			✓	unrestricted
	MCC proprietary (Dark B) (Q)				
	quality	**external publication**			
	draft				
	technical review		proposal		approval
✓	finished				
	reviewer:	(Program Director)		(See External Release Form)	

APPROVALS

	Author	*Project Director*	*Program Director*
Name	Michael Evangelist		Les Belady
Signature	Michael Evangelist		L. Belady
Date	11 / 10 / 89	/ /	11 / 10 / 89

Introduction

The topic of self-stabilizing computing systems has received growing attention in recent years. A system is called *self-stabilizing* if it is guaranteed to converge to a safe state from any state within a finite number of steps. The interest in these systems derives in part from a desire to find a coherent approach to fault tolerance (a self-stabilizing system would recover from a hardware fault that places the system in an unsafe state) and in part from the applicability of the concept to reactive systems (where the idea of an initial state is less important than in terminating sequential computations).

The Software Technology Program at MCC held a Workshop on Self-Stabilizing Systems in Austin on August 11, 1989. The Workshop attracted an enthusiastic international audience (see attached List of Attendees), which heard eight presentations on various approaches to the subject. Although most of the papers will be submitted to journals and other publications, this *Proceedings* collects them into a single volume for the convenience of the attendees and other interested researchers.

Michael Evangelist
Shmuel Katz

Workshop on Self-Stabilizing Systems

Software Technology Program, MCC

Austin, Texas

August 11, 1989

FINAL PROGRAM

9:00-9:45 "Convergence of Iteration Systems",
Anish Arora, Paul Attie, Michael Evangelist, and Mohamed Gouda.

9:45-10:30 "Self Stabilization of Dynamic Systems",
Shlomo Moran, Amos Israeli, and Shlomi Dolev.

10:30-10:45 *Break.*

10:45-11:30 "Self-stabilizing Extensions for Message-passing Systems",
Shmuel Katz and Ken Perry

11:30-12:15 "Proofs of Two Self-Stabilizing Termination Detection Algorithms",
Charles Richter.

12:15-1:30 *Catered Lunch.*

1:30-2:15 "On Relaxing Interleaving Assumptions",
James Burns, Mohamed Gouda, and Raymond Miller.

"Stabilization and Pseudo-Stabilization",
James Burns, Mohamed Gouda, and Raymond Miller.
(paper not available at the time of preparation of these proceedings)

2:15-3:00 "The Instability of Self-Stabilazation",
Mohamed Gouda, Rodney Howell and Louis Rosier.

3:00-3:45 *Break.*

3:15-4:00 "On Self-Stabilization, Nondeterminism, and Inherent Fault Tolerance",
Farokh Bastani, I-Ling Yen, and Yi Zhao.

4:00-5:00 *OpenDiscussion. Future Meetings.*

LIST OF ATTENDEES

Computer Science Dept.
TAMU
College Station, TX 77843
abello@cssun.tamu.edu

Jim Anderson
Computer Science Dept.
University of Texas
Taylor Hall 2124
Austin, TX 78712
jha@cs.utexas.edu

Anish Arora
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
MCC
P. O. Box 200195
Austin, TX 78720
arora@mcc.com

Paul Attie
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
MCC
P. O. Box 200195
Austin, TX 78720
attie@cs.utexas.edu
attie@mcc.com

Michael Barnett
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
mbarnett@cs.utexas.edu

Sanjoy Baruah
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
sanjoy@cs.utexas.edu

Farokh Bastani
Dept. of Computer Science
University of Houston
Houston, TX 77004
coscfb@cs.uh.edu

Glenn Bruns
MCC
P. O. Box 200195
Austin, TX 78720
bruns@mcc.com

James E. Burns
2837 Ponderosa Circle
Decatur, GA 30033
(404) 894–3816
burns@gatech.edu

Ken Calvert
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
calvert@cs.utexas.edu

E. Allen Emerson
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Atustin
Austin, TX 78712
MCC
P. O. Box 200195
Austin, TX 78720
emerson@cs.utexas.edu
emerson@mcc.com

Ira R. Forman
MCC
P. O. Box 200195
Austin, TX 78720
forman@mcc.com

M.G. Gouda
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Atustin
Austin, TX 78712
gouda@cs.utexas.edu

Ted Herman
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
ccen001@uta3081.cc.utexas.edu

Tom Jacob
Dept. of Computer Science
University of North Texas
Denton, TX 76203
jacob@dept.csci.unt.edu

Ravi Jain
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
rjain@cs.utexas.edu

Shmuel Katz
Computer Science
The Technion
Haifa, Israel
katz@techsel.bitnet

Edgar Knapp
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
knapp@cs.utexas.edu

Jacob Kornerup
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
kornerup@cs.utexas.edu

Al Mok
Tay 3.140C
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
mok@cs.utexas.edu

Shlomo Moran
Computer Science
The Technion
Haifa, Israel
moran@techsel.bitnet

Rob Munoz
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
munozr@cs.utexas.edu

Dave Nanmann
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
nanmann@cs.utexas.edu

Michael Ojukwu
Dept. of Electrical & Computer Engineering
University of Texas at Austin
Austin, TX 78712
MCC
P. O. Box 200195
Austin, TX 78720
ojukwu@mcc.com

Colin Potts
MCC
P. O. Box 200195
Austin, TX 78720
potts@mcc.com

Charlie Richter
MCC
P. O. Box 200195
Austin, TX 78720
richter@mcc.com

Lou Rosier
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
rosier@cs.utexas.edu

Gene Thorne
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712

Don Varvel
Dept. of Computer Science
Taylor Hall 2124
University of Texas at Austin
Austin, TX 78712
varvel@cs.utexas.edu

Duane Voth
MCC
P. O. Box 200195
Austin, TX 78720
duanev@mcc.com

Zvi Weiss
MCC
P. O. Box 200195
Austin, TX 78720
weiss@mcc.com

I-Ling Yen
Dept. of Computer Science
University of Houston
Houston, TX 77004
ilyen@cs.uh.edu

Yi Zhao
Dept. of Computer Science
University of Houston
Houston, TX 77004
yzhao@cs.uh.edu

Convergence of Iteration Systems[†]

Anish ARORA [1,2] *Paul ATTIE* [1,2] *Michael EVANGELIST* [1]
Mohamed GOUDA [1,2]

Abstract

An iteration system is a set of assignment statements whose computation proceeds in steps: at each step, an arbitrary subset of the statements is executed in parallel. The set of statements thus executed may differ at each step; however, it is required that each statement is executed infinitely often along the computation. The convergence of such systems (to a fixed point) is typically verified by showing that the value of a given variant function is decreased by each step that causes a state change. Such a proof requires an exponential number of cases (in the number of assignment statements) to be considered. In this paper, we present alternative methods for verifying the convergence of iteration systems. In most of these methods, upto a linear number of cases need to be considered.

[†] 1. Microelectronics and Computer Technology Corporation, Austin
2. Department of Computer Sciences, The University of Texas at Austin

1 Introduction

Iteration systems are a useful abstraction for computational, physical and biological systems that involve "truly concurrent" events. In computing science, they can be used to represent self-stabilizing programs, neural networks, transition systems and array processors. This wide applicability derives from the simplicity and generality of the formalism.

Informally, an iteration system is defined by a finite set of variables, V. Associated with each variable is a function called its *update function*. The computation of the system proceeds in steps. At each step, the variables in an arbitrary subset of V are updated by assigning each variable the value obtained from applying its update function to the current system state. The set of variables thus updated may differ at each step; however, it is required that each variable in V be updated infinitely often.

In allowing an arbitrary subset of variables to be updated at each step, our formalism admits a large number of widely varying computations. These range from the sequential computations in which exactly one variable is updated at every step to the parallel computation in which each variable is updated at every step. By comparison, traditional semantics admit computations of lesser variety. For example, interleaving requires one enabled event to be executed at each step (see, for instance, the work on CSP [Hoa], UNITY [CM] and I/O Automata [Lyn]), whereas maximal parallelism requires that all enabled events are executed at every step (see, for instance, the work on systolic arrays [KL] and cellular automata [Wol]).

A property of interest in iteration systems is convergence. This property is useful in studying the self-stabilization of distributed programs (cf. [Dij1], [BGW], [Dij] and [GE]), convergence of iterative methods in numerical analysis (cf. [Rob] and [BT]), and self-organization in neural networks (cf. [Koh] and [Arb]). Informally, an iteration system is called convergent if on starting in an arbitrary state the system is guaranteed to reach a fixed point; that is, a state in which no update can cause a state change. The standard method for verifying that an iteration system is convergent is to exhibit a variant function (cf. [Gri]) whose value is bounded from below and is decreased by each step that causes a state change. Since any subset of the variables can be updated in a step, the number of cases that need to be considered are $2^n - 1$, where n is the number of variables in the system. In this paper, we discuss new methods for verifying the convergence of iteration systems. Nearly all these methods require upto n cases to be considered.

The rest of this paper is organized as follows. In Section 2, we formally define iteration systems and their dependency graphs. (The dependency graph of an iteration system captures the "depends on" relation

between the variables in the system.) In Section 3, we identify two classes of iteration systems, namely those whose dependency graphs are acyclic or self-looping, and present a theorem that establishes efficient proof obligations for verifying the convergence of these two classes. This theorem is then extended to general, deterministic iteration systems in Section 4. In Section 5, we show that, with minor modifications, our results continue to hold in nondeterministic iteration systems. Concluding remarks are in Section 6.

2 Iteration Systems

An *iteration system*, I, is defined by the pair (V, F), where

- V is a finite, nonempty set of variables. Each variable v in V has a predefined domain Q_v. Let Q denote the cartesian product of the domains of all variables in V.
- F is a set of "update" functions with exactly one function f_v associated with each variable v in V, where f_v is a mapping from Q to Q_v.

A *state* q of I is an element of Q. We adopt the notation q_v to denote the value of variable v in state q. A state q is called a *fixed point* of I iff for each variable v in V, $f_v(q) = q_v$.

A *step* of I is defined to be a subset of V; informally, a step identifies those variables that are updated when the step is executed. A *round* of I is a minimal, finite sequence of steps with the property that each variable in V is an element of at least one step in the round. A *computation* of I is an infinite sequence of rounds. Notice that since each variable is updated at least once in every round, each variable is updated infinitely often in every computation.

The *application* of a finite sequence of steps S to a state q, denoted $S \circ q$, is the state q' defined inductively as follows:

- if S is empty, then $q' = q$
- if S is a single step, then for every variable v in V,

$$q'_v = \begin{cases} f_v(q) & \text{, if } v \in S \\ q_v & \text{, otherwise} \end{cases}$$

- if S is the concatenation of two sequences $S = S'; S''$, then $q' = S'' \circ (S' \circ q)$.

A computation C is called *convergent* iff for every state q, there exists a finite prefix, S, of C such that $S \circ q$ is a fixed point of I. An iteration system is called *convergent* iff all of its computations are convergent.

As shown in the following examples of iteration systems, it is convenient to represent an iteration system by a set of assignment statements, one for each variable. Each statement has the form ⟨*variable*⟩ := ⟨*corresponding update function*⟩. We will later prove each of these iteration systems to be convergent.

Example 1: (Greatest Common Divisor)

Let x, y and z be variables that range over the natural numbers. Then, the three assignment statements

$$x := \underline{\text{if}}\ x > y\ \underline{\text{then}}\ x - y\ \underline{\text{else}}\ x$$
$$y := \underline{\text{if}}\ x < y\ \underline{\text{then}}\ y - x\ \underline{\text{else}}\ y$$
$$z := \underline{\text{if}}\ x = y\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ z + 1$$

define a convergent iteration system. At fixed point, the value of x is the greatest common divisor of the initial values of x and y, and $y = x$ and $z = 0$. □

Example 2: (Minimum of a bag)

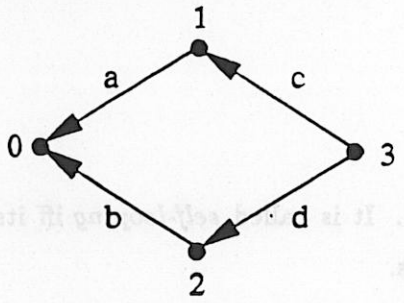
Let x be an integer array of size n. Then, the following assignment statements:

$$x[1] := x[1]$$
$$x[2] := \min(x[2], x[1])$$
$$\vdots$$
$$x[n] := \min(x[n], x[n-1])$$

define a convergent iteration system. At fixed point, the value of each $x[i]$ is the minimum of the initial values in the sub-array $x[1], x[2], \ldots, x[i]$. □

Example 3: (Shortest Path)

Consider the directed graph



It has four nodes 0-3, and four edges. Each edge is labeled with a non-negative integer constant denoting its length. Associated with each node i is a variable $v[i]$, of type *record*, that has two integer components, $first[i]$ and $second[i]$. The assignment statements

$$v[0] := (0, 0)$$
$$v[1] := (a, 0)$$
$$v[2] := (b, 0)$$
$$v[3] := \underline{\text{if}}\ first[1] + c \leq first[2] + d\ \underline{\text{then}}\ (first[1] + c, 1)\ \underline{\text{else}}\ (first[2] + d, 2)$$

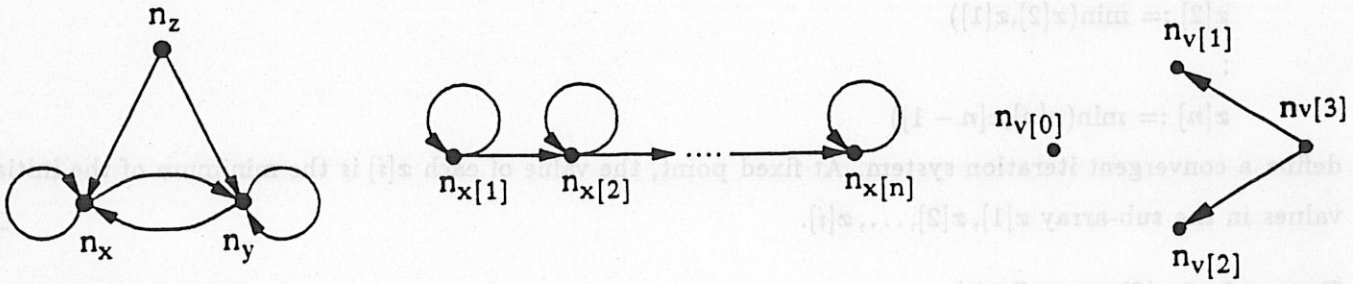
define a convergent iteration system. At fixed point, each $first[i]$ is the length of the shortest path from node i to node 0, and $second[i]$ is the nearest neighbor to node i along this path. Extending the above system for an arbitrary directed graph is straightforward. □

The objective of this paper is to identify proof obligations that are sufficient to establish the convergence of iteration systems. Towards this end, the following two definitions will prove useful shortly.

Let v and w be variables in V. We say v *depends on* w iff there exist two states q and q' of I such that q and q' differ only in their value of w and $f_v(q) \neq f_v(q')$. Informally, v *depends on* w iff a change in the value of w can cause a change in the value assigned to v by its update function f_v.

The *dependency graph* of I is a directed graph whose nodes correspond to the variables in V and whose directed edges correspond to the *depends on* relation; that is, the set of nodes of the dependency graph is $\{n_v | v \in V\}$ and its set of directed edges is $\{(n_v, n_w) | v \in V,\ w \in V,$ and v *depends on* $w\}$.

The dependency graphs for the iteration systems in Examples 1, 2 and 3 are as follows:



Henceforth, we shall use 'variable' and 'node' interchangeably when referring to the dependency graph of an iteration system.

3 Convergence of Non-Cyclic Systems

An iteration system is called *acyclic* iff its dependency graph is acyclic. It is called *self-looping* iff its dependency graph has one or more cycles, and all its cycles are self-loops.

In this section we state and prove a fundamental theorem concerning the convergence of acyclic and self-looping iteration systems. The implications of this theorem are discussed subsequently.

Theorem 1:

If an iteration system is

(a) acyclic, then it is convergent, and if it is

(b) self-looping and has one convergent computation, then it is convergent.

Proof:

To prove the theorem we need to introduce the following new concepts: "*rank*", "*stable*", "*#steps*", "$\uparrow$", and "$\downarrow$".

For an acyclic or self-looping iteration system, define *rank* to be the function that assigns to each variable v in V a positive integer as follows:

$$rank(v) = 1 + \max\{rank(w) \mid (w \in V \wedge v \neq w \wedge v \textit{ depends on } w)\}$$

By convention, the value of "max" applied to the empty set is 0; thus, the rank of a variable that does not depend on any other variable is 1. Notice that this definition is recursive and requires, in order to be well-defined, that the dependency graph has no cycle of length two or greater. Therefore, this definition applies only to acyclic and self-looping systems.

A variable v in V is *stable* in state q iff for every finite sequence of steps, S, the value of v in q is the same as its value in state $S \circ q$.

Let $\#steps(C, q, k)$ denote the partial function that returns the number of steps in the minimal prefix S of computation C such that every variable of rank at most k is stable in the state $S \circ q$. The value of $\#steps(C, q, k)$ is undefined when no such prefix exists.

Let k be any positive integer, then $C \uparrow k$ denotes the unique prefix of computation C that consists of exactly k rounds, and $C \downarrow k$ denotes the computation that results after removing the prefix $C \uparrow k$ from computation C.

Proof of part (a):

The proof is by a straightforward induction on the rank of variables. We argue that after the application of the first k rounds ($k \leq |V|$) of an arbitrary computation C to an arbitrary state q, all variables with rank at most k are stable in the resulting state $(C \uparrow k) \circ q$. Since the rank of any variable in V is at most $|V|$, it follows that after $|V|$ rounds the system is at a fixed point. For the base case, note that the update functions of variables with rank = 1 are constant functions.

Proof of part (b):

We need to prove that if some computation C' is convergent then for an arbitrary computation C and an arbitrary state q there exists a positive integer i such that all variables in V are stable in state $(C \uparrow i) \circ q$; that is, $(C \uparrow i) \circ q$ is a fixed point.

The proof proceeds by induction on the rank k of variables. Let the induction hypothesis be:

$$\exists i \; \forall v \quad (\; rank(v) \leq k - 1 \quad \Rightarrow \quad v \text{ is stable in } (C \uparrow i) \circ q \;)$$

Note that as C' is convergent, $\#steps(C', q, k)$ is well defined for every rank k.

Base Case: $k = 1$.

A variable whose rank is 1 depends on no other variable, therefore, the sequence of values it takes in successive updates is a function only of the number of updates to it. Hence, it is necessarily stable after $\#steps(C', q, 1)$ updates to it in any computation starting in state q. Since each round contains at least one update to every variable, each variables whose rank is 1 is stable in $(C \uparrow \#steps(C', q, 1)) \circ q$.

Induction Step: $k > 1$.

Let $q' = (C \uparrow i) \circ q$, where i is the least integer that satisfies the induction hypothesis. Hence,

$$\forall v \quad (\, rank(v) \leq k - 1 \quad \Rightarrow \quad v \text{ is stable in } q' \,) \tag{1}$$

Applying C' to q', we know that every variable with rank $\leq k$ will be stable within $\#steps(C', q', k)$ updates to that variable

$$\forall v \quad (\, rank(v) \leq k \quad \Rightarrow \quad v \text{ is stable in } (C' \uparrow \#steps(C', q', k)) \circ q' \,)$$

From (1), all variables of rank $\leq k - 1$ are stable in state q' . Thus, all computations starting in state q' will produce the same sequence of values (as a function of the number of updates) for each variable of rank k. As each round contains at least one update per variable, every variable of rank k will be stable after $\#steps(C', q', k)$ rounds of any computation. In particular, for the computation $C \downarrow i$ (that is, C with $(C \uparrow i)$ removed),

$$\forall v \quad (\, rank(v) \leq k \quad \Rightarrow \quad v \text{ is stable in } ((C \downarrow i) \uparrow \#steps(C', q', k)) \circ q' \,)$$

Let $j = \#steps(C', q', k)$. Now $((C \downarrow i) \uparrow j) \circ q' = ((C \downarrow i) \uparrow j) \circ ((C \uparrow i) \circ q) = (C \uparrow i); ((C \downarrow i) \uparrow j) \circ q = (C \uparrow (i + j)) \circ q$ and so,

$$\forall v \quad (\, rank(v) \leq k \quad \Rightarrow \quad v \text{ is stable in } (C \uparrow (i + j)) \circ q \,)$$

and the inductive hypothesis is established for rank k. □

The examples in Section 2 illustrate Theorem 1. For instance, the iteration system of Example 3 is acyclic; thus, each of its computations is convergent by Theorem 1(a). The iteration system of Example 2 is self-looping, and any computation where the first step updates $x[1]$, the second updates $x[2]$, and so on is convergent; thus, each computation of the system is convergent by Theorem 1(b).

As mentioned in the introduction, verifying the convergence of an iteration system is generally accomplished by exhibiting a variant function whose value is bounded from below and is decreased by each step that causes a state change. Such a proof requires $2^n - 1$ cases to be considered, where n is the number of variables in the system. In contrast, Theorem 1 shows that verifying the convergence of acyclic systems

requires no such case analysis.

The theorem also states that the convergence of all computations of a self-looping iteration system can be established from the convergence of a computation of choice. One possibility is to choose this computation to be the one in which each variable is updated at every step. The convergence of this computation can then be proved by the variant function method which, in this instance, needs only one case to be considered. Another possibility is to choose computations in which exactly one variable is updated at each step; in this instance, the variant function method requires n cases.

Theorem 1 cannot be made to apply to all iteration systems. Consider, for example, the iteration system defined by the two assignment statements

$x := y$

$y := x$.

Although this system has many computations that are convergent (for example, all computations where exactly one variable is updated at the first step), it also has a computation that is not convergent (for example, the computation where both x and y are updated at each step). Thus, unlike Theorem 1, one cannot establish that this system is convergent by exhibiting one convergent computation. Similar examples have been presented in [Dij] and [Rob].

In fact, it is straightforward to show that Theorem 1 cannot be made to apply to any class of iteration systems that properly includes acyclic and self-looping systems. The proof for this follows from a construction that exhibits, for each directed graph G that has a cycle of two or more nodes, an iteration system I such that the dependency graph of I is G, and I has both convergent and non-convergent computations.

The following lemma states that for any cyclic iteration system I there is a self-looping system which captures a subset of the computations of I and, thereby, is a possible implementation for I. This shows that the class of self-looping systems is rich.

Lemma 1:
For each iteration system I that is neither acyclic nor self-looping, there exists a self-looping iteration system I' that satisfies the following two conditions:

- There is a one-to-one correspondence between the states of I and those of I'.
- Every computation of I' is a computation of I.

Proof: Replace each maximally strongly connected component that has two or more nodes in I by a single node with a self-loop in the dependency graph of I'. This is accomplished by replacing all the

variables in the component with one variable of type record; the components of this record correspond, in a one-to-one manner, to the replaced variables. □

System I' in Lemma 1 is self-looping; hence, its convergence can be established by Theorem 1(b). (The convergence of I', however, does not necessarily imply the convergence of the original system I.) Consider, for instance, the iteration system in Example 1. This system is neither acyclic nor self-looping because its dependency graph has a maximally strongly connected component consisting of variables x and y. By replacing these two variables by one variable with two components, also called x and y for convenience, we obtain the following implementation of the system:

$$(x, y) := (\underline{\text{if}}\ x > y\ \underline{\text{then}}\ x - y\ \underline{\text{else}}\ x\ ,\ \underline{\text{if}}\ x < y\ \underline{\text{then}}\ y - x\ \underline{\text{else}}\ y\ )$$
$$z := \underline{\text{if}}\ x = y\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ z + 1.$$

As this system is self-looping, its convergence can be established by Theorem 1(b).

4 Convergence of Cyclic Iteration Systems

In this section, we generalize our analysis for the convergence of acyclic and self-looping systems to the convergence of general iteration systems. Our starting point is to note the basic characteristic of an iteration system that is neither acyclic nor self-looping, namely the existence of at least one maximally strongly connected component in its dependency graph that consists of two or more nodes. For convenience, we call a maximally strongly connected component that has two or more nodes a *district.*

Let D be a district in the dependency graph of an iteration system, I. The *iteration system associated with* D is the iteration system (V_D, F_D) that satisfies the following two conditions:

- The set of variables, V_D, is the set of all variables in D together with each variable that is not in D but some variable in D depends on it.
- The set of update functions, F_D, is defined as follows. The update function for a variable in D is the same as its update function in I, whereas the update function for a variable in $V_D \setminus D$ is the identity function for that variable.

For instance, the iteration system in Example 1 has one district whose associated iteration system can be defined by the two assignment statements

$$x := \underline{\text{if}}\ x > y\ \underline{\text{then}}\ x - y\ \underline{\text{else}}\ x$$
$$y := \underline{\text{if}}\ x < y\ \underline{\text{then}}\ y - x\ \underline{\text{else}}\ y.$$

An iteration system is called *district-convergent* iff the iteration system associated with each district in the dependency graph of the system is convergent. Since acyclic and self-looping systems do not have any districts in their dependency graphs, they are trivially district-convergent.

An iteration system is called *0-cyclic* iff its dependency graph has no maximally strongly connected component that consists of a single node with a self-loop; otherwise, the iteration system is called *1-cyclic*. Note that each iteration system is either 0-cyclic or 1-cyclic; in particular, acyclic systems are 0-cyclic whereas self-looping ones are 1-cyclic.

The following theorem generalizes Theorem 1.

Theorem 2

If a district-convergent iteration system is

(a) 0-cyclic, it is convergent, and if it is

(b) 1-cyclic and has one convergent computation, then it is convergent.

Proof:

We extend the definition of *rank* in the proof of Theorem 1 to an arbitrary iteration system, I, as follows. Consider the "condensation" of its dependency graph (i.e., collapse each district into a single node; see [Har]). It is straightforward to see that each cycle in the condensation is a self-loop. Assign ranks to the nodes in the condensation using the previous definition of rank. Now, the *rank* of a variable v in I is defined to be the rank of its corresponding node in the condensation.

The proof proceeds by induction on the rank k of variables, and is similar to the proof of Theorem 1. The induction hypothesis is:

> for an arbitrary computation C and an arbitrary state q in which every variable of rank lower than k is stable, there exists a finite prefix S of C such that all variables of rank k will be stable in $S \circ q$.

For both the base case and the induction step, the following arguments suffice:

- if variable v does not depend on any variable or depends only on variables of lower rank (which are stable in q), then v is clearly stable after the first round.
- if v depends on itself but on no other variable of the same rank, then the counting argument in the proof of Theorem 1(b) ensures that v will eventually be stable.
- finally, it may be the case that v is in some district D. We argue that, once all the variables of lower rank on which v depends on are stable, the convergence of the iteration system associated with D guarantees that v will eventually be stable.

□

In the remainder of this section we identify two proof obligations which are sufficient to establish the convergence of an iteration system that is associated with a district. These obligations consists of exhibiting either a variant function for each node in a selected set of nodes in the district (Lemma 2), or a single variant function for the whole district (Lemma 3).

The intuition underlying Lemma 2 is to "break" each cycle in the district by ensuring that some distinguished variable on the cycle will eventually reach a stable value. This is achieved by exhibiting a variant function for the distinguished variable whose value decreases each time the value of the variable is changed by an update. Once every distinguished variable in the district becomes stable (i.e. has a fixed value), the iteration system associated with the district starts to behave like an acyclic system and, so, eventually reaches a fixed point.

A more general approach to solving the same problem is to exhibit a variant function for all the variables in the district. The value of this function is decreased by each step that causes a state change. See Lemma 3 below.

In what follows, let

D be a district in some iteration system I,

(V_D, F_D) be the iteration system associated with D,

Q_D be the set of states of (V_D, F_D), and

f_v be the update function of a variable v in (V_D, F_D), and

$V_{D'}$ be the set of variables in D', a subgraph of D, and

N be an arbitrary set that is well-founded under some relation $<$.

Lemma 2: *(Local Variant)*

If each directed cycle in D has a variable v and a variant function $\# : Q_v \rightarrow N$ such that for each state q in Q_D,

$$\#(f_v(q)) < \#(q_v) \quad \vee \quad (f_v(q) = q_v)$$

then the iteration system (V_D, F_D) is convergent.

Lemma 3: *(Global Variant)*

If there is a variant function $\# : Q_D \rightarrow N$ such that for each state q in Q_D, and for each strongly connected component D' in D,

$$\#(V_{D'} \circ q) < \#(q) \quad \vee \quad (V_{D'} \circ q = q)$$

then the iteration system (V_D, F_D) is convergent.

Proofs of Lemmas:

To prove these lemmas, we augment the set of concepts introduced in the previous proofs by the following:

For a variable v in V and subset $W \subseteq V$, define $allpaths(v, W)$ to be the subgraph in the dependency graph of I containing (exactly) those paths that begin at v, do not contain any variable in W as an intermediate node and end at some variable in W.

A variable v is *unaffected* by variable w in state q iff for an arbitrary state q' that differs from q only in its value of w, and an arbitrary finite sequence of steps S,

$$(S \circ q)_v = (S \circ q')_v.$$

Intuitively, this implies that the value assigned to v in the application of any sequence to the state q is independent of the value of w.

Proof of Lemma 2:

The proof obligation is to show that for an arbitrary computation, C, of (V_D, F_D) and an arbitrary state, $q \in Q_D$, there exists a positive integer i such that $(C \uparrow i) \circ q$ is a fixed point of (V_D, F_D).

By the antecedent of the lemma, we can distinguish on each cycle in D some variable that satisfies the property stated in the lemma. Let W be the set of variables thus distinguished, and let U abbreviate the set $V_D \setminus D$.

We show that for an arbitrary variable v in D there exists a positive integer j such that v is stable in $(C \uparrow j) \circ q$. The maximum j for the variables in D is then the appropriate positive integer i for which $(C \uparrow i) \circ q$ is a fixed point. There are 3 cases to be considered.

- $v \in U$: v is already stable in q as it is updated by the identity function; that is, $j = 0$.
- $v \in W$: by the property stated in the lemma and the fact that N is well-founded under $<$, we are guaranteed that v will be stable after a finite number of steps in any computation. Let k be the least positive integer such that all variables in W are stable in $q' = ((C \uparrow k) \circ q)$.
- $v \in (V_D \setminus W)$: let x denote any variable not in $allpaths(v, W \cup U)$. We claim that v is *unaffected* by x in q'. To prove this, we first note that the construction of $allpaths(v, W \cup U)$ ensures that every path from v to x must pass through some variable in $W \cup U$. Since all variables in $W \cup U$ are stable in q', it follows that v is *unaffected* by x in q'.

  Next, we show that $allpaths(v, W \cup U)$ is acyclic. The only variables in $allpaths(v, W \cup U)$ that are not in D are in U, but these have, by construction, no successors in $allpaths(v, W \cup U)$. It follows

that all cycles in $allpaths(v, W \cup U)$ must be contained in D and ,therefore, must pass through some distinguished node in W. However, this is impossible in $allpaths(v, W \cup U)$ as no distinguished node has a successor. Thus, $allpaths(v, W \cup U)$ is acyclic.

Hence, the value of v is affected only by the variables in $allpaths(v, W \cup U)$, and since the latter is an acyclic graph with variables of rank 0 (that is, $W \cup U$) stable in state q', we conclude from Theorem 1(a) that on the application of $l = rank(v)$ rounds to q', v will be stable; that is, v is stable in $(C \uparrow j) \circ q$ where $j = k + l$.

Proof of Lemma 3:

To prove that the iteration system (V_D, F_D) is convergent, we show that for an arbitrary step $W \subseteq V_D$ and an arbitrary state $q \in Q_D$,

$$\#(W \circ q) < \#(q) \quad \vee \quad (W \circ q = q).$$

Since N is well-founded under $<$, there is no infinitely descending chain of values returned by $\#$ and, hence, after a finite number of steps the iteration system is guaranteed to be at a fixed point.

The proof is organized as follows: first, we show that $W \circ q$ is the same as the state that results from the application of a finite sequence of mutually disjoint steps to q, each step of which updates the variables in some strongly connected component of the dependency graph of (V_D, F_D). Then, we argue that by the property stated in the lemma, it must be the case that $\#(W \circ q) < \#(q) \quad \vee \quad (W \circ q = q)$.

Consider the "subgraph induced by" W, G_W, in the dependency graph of (V_D, F_D) (i.e., its maximal subgraph with node set W; see [Har]). Take the condensation of G_W. As observed in the proof of Theorem 2, the *rank* of the variables in W can be consistently computed via the condensation of G_W. Let k be the maximum *rank* thus assigned.

Next, we make the observation that if two variables corresponding to different nodes in the condensation are updated simultaneously, then

- if they are of different rank, the resulting state is the same as the one obtained by updating the higher rank variable first, and then updating the other variable, and
- if they are of the same rank, the resulting state is the same as the one obtained by updating them in any sequential order.

To complete the proof, consider all the variables in W of $rank = i, (1 \leq i \wedge i \leq k)$. These variables can be uniquely partitioned into sets, each of which corresponds to some node in the condensation of G_W.

Let W_i be an arbitrary sequence of the sets in this partition such that each set appears exactly once. By the observation made in the previous paragraph $(W_k; W_{k-1}; \ldots; W_1) \circ q = W \circ q$. However, by the property stated in the lemma, we know that the application of a step containing exactly the variables in some strongly connected component can only lower the value returned by the variant function if there is a change of state. Hence, if $(W \circ q \neq q)$ then $\#(W \circ q) < \#(q)$. □

As an example, both Lemma 2 and Lemma 3 can be used to show that the iteration system associated with the district in Example 1 is convergent. In using Lemma 2, let the variant functions for both x and y be their respective identity functions. In using Lemma 3, let the global variant function be the sum of x and y. In either case, the convergence of the entire system in Example 1 can now be established by Theorem 2.

5 Convergence of Nondeterministic Systems

So far, the definition of an iteration system associates exactly one (deterministic) update function with each variable. We now extend this definition to allow each variable to be updated by more than one update function. More specifically, we associate with each variable v a finite, non-empty set of update functions F_v. At each step in which v is updated, one of the functions in F_v is chosen to update v. This choice is arbitrary except for the requirement that each update function in F_v is chosen infinitely often in every computation. (Note that this is possible because each variable is updated infinitely often in every computation.)

In the next example, we represent a nondeterministic iteration system by a set of assignment statements (with choice), one for each variable. If $F_v = \{f, g, \ldots, h\}$ then the assignment statement that updates v has the form:

$$v := f \mid g \mid \ldots \mid h$$

Example 4. (Nondeterministic Shortest Path)
We exhibit a nondeterministic iteration system for the directed graph in Example 3. The nondeterminism makes it possible to reduce the 'atomicity' of the update functions. In fact, every update function refers uniquely to one edge in the graph, as follows:

$v[0] := (0, 0)$
$v[1] := (a, 0)$
$v[2] := (b, 0)$
$v[3] :=$ if $first[1] + c \leq first[3] \vee second[3] = 1$ then $(first[1] + c, 1)$ else $v[3]$

$$\mid \underline{\text{if}}\ first[2] + d \leq first[3] \vee second[3] = 2\ \underline{\text{then}}\ (first[2] + d, 1)\ \underline{\text{else}}\ v[3]$$

This system is convergent to some fixed point, and when it is at a fixed point, each $first[i]$ is the length of the shortest path from node i to node 0, and each $second[i]$ is the nearest neighbor to node i along this path. □

We now redefine four concepts that were introduced earlier in order to accommodate the extension to nondeterminism.

- A state q of an iteration system is a *fixed point* iff for each variable v and each update function f_v in F_v, $f_v(q) = q_v$.
- A computation is said to be *convergent* iff for each state q and for each choice of update functions in the computation there exists a finite prefix S of the computation such that $S \circ q$ is a fixed point.
- The *depends on* relation is redefined as follows: variable v *depends on* variable w iff there exist two states q and q' such that q and q' differ only in their value of w and $f_v(q) \neq f_v(q')$ for some f_v in F_v.
- We augment the notion of the *iteration system* (V_D, F_D) *associated with* a district D in the dependency graph of an iteration system I. With each variable v in V_D that is also in D, we now associate its set of update functions in I, i.e. F_v. The set of update functions for every variable in V_D but not in D is defined to be the set that contains only the identity function for that variable.

Next, we extend the previous results to nondeterministic iteration systems.

Theorem 3:

If a nondeterministic iteration system is

(a) acyclic and has a fixed point, then it is convergent, and if it is

(b) self-looping and has one convergent computation, then it is convergent.

Proof Sketch: The assumption that a fixed point exists can be used to show that the induction argument for the deterministic case continues to hold. In particular, the assumption is needed to assert that once all the variables of rank lower than that of variable v are stable then all the update functions of f_v compute the same value.

□

Theorem 4:

If a district-convergent nondeterministic iteration system is

(a) 0-cyclic and has a fixed point, it is convergent, and if it is

(b) 1-cyclic and has one convergent computation, then it is convergent.

Proof Sketch: To prove the convergence of an arbitrary computation with an arbitrary choice of functions at each step (that respects the selection restriction outlined above), we consider the convergent computation with the same choice of functions. Now a counting argument that is very similar to the one in the proof of Theorem 1(b) can be used to exhibit the required convergence. □

Lemma 4: *(Local Variant)*

If each directed cycle in D has a variable v and a variant function $\# : Q_v \to N$ such that for each state q in Q_D, and each update function f_v in F_v

$$\#(f_v(q)) < \#(q_v) \quad \vee \quad (f_v(q) = q_v)$$

then the iteration system $(V_D, F_D)$is convergent provided it has a fixed point. □

Lemma 5: *(Global Variant)*

If there is a variant function $\# : Q_D \to N$ such that for all states q in Q_D, for all strongly connected components D' in D, and for every choice of update functions for the variables in $V_{D'}$,

$$\#(V_{D'} \circ q) < \#(q) \quad \vee \quad (V_{D'} \circ q = q)$$

then the iteration system (V_D, F_D) is convergent. □

6 Conclusions

We have defined a very general model of computation that exhibits true concurrency, and have considered convergence as a typical property of systems expressed in this model. We established several results that reduce the proof burden involved in establishing convergence.

When analyzing concurrent systems, convergence can be used to model termination. Since many progress (that is, eventuality) properties of a concurrent system can be reduced to the termination of a derived system (see, for example, [GFMR]), our techniques can be used to verify general progress properties.

There are several issues that need to be investigated in extending this work. Other than identifying more general sufficiency conditions and determining their scope, a comparison of the 'rate' of convergence of various types of computations is needed. Methods regarding other properties, such as safety, also need to be studied.

References

[Arb] M.A. Arbib, *Brains, Machines and Mathematics*, Springer-Verlag,1987.

[BGW] G.M. Brown, M.G. Gouda, and C.-L. Wu, "Token Systems that Self-Stablize", *IEEE Transactions on Computers 38(6)*, pp. 845-852, 1989.

[BT] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice-Hall, 1989.

[CM] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley Publishing, 1988.

[Dij] E.W. Dijkstra, EWD306 "The Solution to a Cyclic Relaxation Problem", 1973. Reprinted in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, pp. 34-35, 1982.

[Dij1] E.W. Dijkstra,"Self-stabilizing Systems in Spite of Distributed Control", *Communications of the ACM 17(11)*, pp. 643-644, 1973.

[Gri] D.Gries, *The Science of Programming*, Springer-Verlag, 1981.

[GE] M.G. Gouda and M. Evangelist, "Convergence Response Tradeoffs in Concurrent Systems", *MCC Technical Report #STP-124-89*; also submitted to *ACM TOPLAS*.

[GFMR] O. Grumberg, N. Francez, J.A. Makovsky and W.P. deRoever, "A proof rule for fair termination of guarded commands", *Information and Control 66*, pp. 83-102, 1985.

[Har] F. Harary, *Graph Theory*, Addison-Wesley Publishing, 1972.

[Hoa] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[Koh] T. Kohonen, *Self-Organization and Associative Memory*, Springer-Verlag, 1984.

[KL] H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)," in *Sparse Matrix Proc.*, 1978.

[Lyn] N.A. Lynch, "I/O Automata: A Model for Discrete Event Systems," *Proc. of 22nd Annual Conference on Information Sciences and Systems*, 1988.

[Rob] F. Robert, *Discrete Iterations - A Metric Study*, Springer-Verlag, 1986.

[Wol] S. Wolfram, *Theory and Applications of Cellular Automata, Advanced Series on Complex Systems*, Vol.1, World Scientific Publishing, 1986.

Self Stabilization of Dynamic Systems

(D R A F T)

Shlomi Dolev† *Amos Israeli‡* *Shlomo Moran†*

ABSTRACT

Self stabilizing protocols for dynamic systems are defined and discussed. Three self-stabilizing protocols for dynamic systems are presented. The first protocol is a spanning tree protocol for systems with any communication graph. The second protocol is a mutual exclusion self stabilizing protocol for dynamic tree structured systems. The third protocol is a self stabilizing protocol for mutual exclusion. It requires nothing, except connectivity, from the system's communication graph. This last protocol is obtained by combining the previous two protocols. The combination is enabled by the flexibility achieved by using dynamic protocols.

† Department of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, ISRAEL. Part of this research was supported in part by Technion V.P.R. Funds - Wellner Research Fund, and by the Foundation for Research in Electronics, Computers and Communications, administrated by the Israel Academy of Sciences and Humanities.

‡ Department of Electrical Engineering, Technion - Israel Institute of Technology, Haifa 32000, ISRAEL.

1. Introduction

The concepts of *Self Stabilization* and *Self Stabilizing Systems* were introduced by Dijkstra in [Di74]. In this pioneering work Dijkstra has defined the new class of self stabilizing protocols for distributed systems. These protocols capture the notion of recovering from transient bugs, bugs which change the state of some components of the system but keep them in a working order. In the event of such a bug occurs the system may loose its consistency and enter a somewhat arbitrary configuration while all its components are still correct. A self stabilizing system is a system which regains its consistency by itself, without any kind of an outside intervention.

In [Di74] Dijkstra presented three self-stabilizing protocols for the mutual exclusion problem. Dijkstra's protocols work for a system whose communication graph is a directed ring. A ring in which all edges are directed in the same direction. Communication is allowed in both directions. Following Dijkstra several more works were done in this area. Self stabilizing protocols for a directed ring (or chain) were presented in [BGW87], [BP87], and [Bu87]. In [Kr79] a self stabilizing protocol for a tree structured systems was presented. This protocol allows more then one processor to be in the critical section. A self stabilizing mutual exclusion protocol for systems with arbitrary communication graphs is presented in [Tc81]. This protocol requires an extensive programming work for every individual system since it is recursively composed of protocols defined over covering subgraphs of the original communication graph.

In most of the work done in this field distributed systems are modeled by a set of finite automata. Each such automaton resides on a node of the system's *Communication Graph*. Automata which reside on neighboring nodes are called *neighbors*. Each automaton can observe the state of its neighbors. Its transition function depends on the state of the automaton, the states of its neighbors, and the orientation of the edges connecting them.

Part of the contributions of this paper is in redefining of self stabilizing systems. In the new definition the self stabilization property of a protocol is separated from its task. All self stabilizing protocols known to us are variants of mutual exclusion protocols. This trend is so strong that actually many times mutual exclusion is presented as an inherent requirement for self stabilization. Unlike previous works the task of one of our self stabilizing protocols is to find the spanning tree of the communication graph.

The second aspect addressed in this work is the atomicity level of the protocol. An *atomic step* of a processor is the "smallest" step that the processor executes uninterrupted. Dijkstra's work as well as some other works in this field assume the existence of a specific scheduler, called *Central Demon*. An atomic step in this model is initiated by

the demon pointing to some processor. Only the chosen processor is allowed to change state i.e. **read** the states of all its neighbors, decide whether to change its state, and then move to its new state. Such atomicity may pose difficult hardware problems and cause delays in executions. Some other protocols were design to avoid those implementation difficulties. Brown ,Gouda and Wu [BGW87], and Burns [Bu87], designed protocols that have the following *non-interfering* property: once a processor can move from one state to another, it will make that move regardless of other processors further steps. They proved that a self-stabilizing non-interfering protocol is guaranteed to stabilize even if some processors change state at the same time. In their model an atomic step is initiated by the scheduler pointing to one or more processors. The chosen processors are allowed to change state at the same time, i.e. **read** the states of all their neighbors, decide whether to change there state, and then move to there new state. However in distributed systems more general schedules are possible: For example a system with two processors P_1 and P_2, starting with P_1 reading then P_2 reading and writing, and at last P_2 writing.

Our protocols assume the most basic atomicity level. We assume that a processor has no direct access to the state of its neighbors. Specifically, we assume that the only way to pass information from one processor to another is via the use of shared registers. In all previous works the **read** and **write** actions were integrated into one state transition. In our protocols the only atomicity assumed is the atomicity of **read** and **write** operations to these shared registers. Moreover we step ahead and integrate the value read by a processor as part of its state, and require the system to stabilize in spite of inconsistency between the processor's state and the actual value read by it.

This latter, refined atomicity raises some problems. A processor may change its state according to values which are outdated, since its neighbors changed their states after it **read**. Consequently, it may commit "illegal" state transitions. Even if the transition made by the processor is "legal", no other processor can be informed on the new state before the processor **writes** it in its communication registers. Clearly, protocols which are correct without assuming a central demon, especially protocols that are correct with respect to **read** and **write** atomicity, keep their correctness in its presence. On the other hand, a protocol which is proven correct assuming this presence might not be correct in a system which guarantees only **read** and **write** atomicity. Even the non-interfering property does not ensure the correctness of protocols with **read** and **write** atomicity, as we will show in the sequel. Thus in this respect the protocols presented here are equal to or supersede all previous works.

The use of shared registers enables us to considerably extend the applicability of self stabilizing protocols, which is the third aspect we address here. Dijkstra's protocols and all other self stabilizing mutual-exclusion ring

protocols work only on a directed ring. It is important to notice that in all previous models it is impossible to have self stabilizing mutual exclusion protocols on undirected rings, or on rings whose edges are directed but not necessarily in a consistent way. In these previous models there is no way of breaking symmetry, even by using central demon. This is demonstrated by the following simple example.

Consider a four processor ring system as described in Figure 1.1 where P_2 and P_3 are identical processors. Assume that P_2 and P_3 "thinks" that P_1 is their left neighbor and P_4 is their right neighbor. Starting with P_2 and P_3 in the same state it is easy to see that there exist a fair schedule in which every step taken by P_2 is followed by the same step taken by P_3. Hence P_2 and P_3 either enter the critical section together infinitely often, or are starved, so no self stabilizing protocol for mutual exclusion for this system exist. Similarly, in case of a directed tree system, as described in Figure 1.2, with out sense of direction, as in Kruijer's model, no self stabilizing mutual exclusion protocol exist.

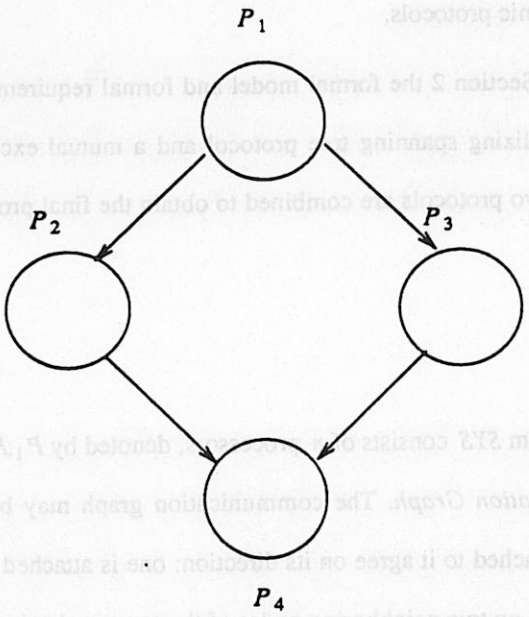


Fig 1.1

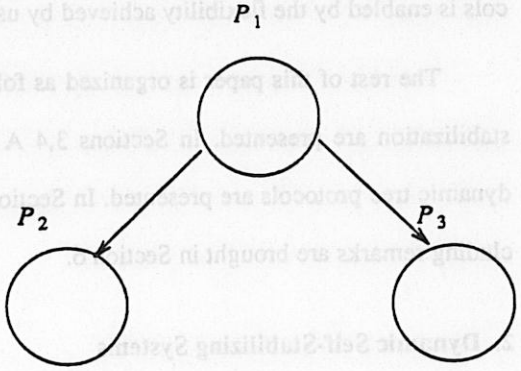


Fig 1.2

Due to the refined atomicity level of our system we achieve, for the first time, symmetry braking. We allow the processors to write different value to different neighbors and thus to break symmetry, when needed. The sense of direction becomes unnecessary and the design of self stabilizing mutual exclusion (and other asymmetric tasks) protocols, for any given graph, is possible. Moreover it is possible to design self stabilizing protocol in which the program of each processor depends only on the number of links connected to it (and their direction if the underlying

graph is directed). Using this property, *dynamic protocols* in which the processor's program accommodate the changes in their local topology (the number and direction of their links) can be designed. Such protocol can be applied to *dynamic systems*, which are systems whose topology may change during execution. Using the self stabilization property of the protocol and the ability to accommodate to the local topology, no initialization is needed following any such change.

In this paper we present three self stabilizing dynamic protocols. The task of the first protocol is to construct a rooted spanning tree of the system's communication graph. This protocol does not assume any special communication graph. The second protocol achieves mutual exclusion on a tree structured system. Both these protocols assume the most basic atomicity level, that is **read** and **write** atomicity. We conclude this work by combining the two aforementioned protocols into a single self stabilizing dynamic protocol for mutual exclusion in general graphs. This last protocol enjoys the same atomicity and dynamic properties as its two sub protocols. Combining the two protocols is enabled by the flexibility achieved by using dynamic protocols.

The rest of this paper is organized as follows: in Section 2 the formal model and formal requirement for self stabilization are presented. In Sections 3,4 A self stabilizing spanning tree protocol and a mutual exclusion on a dynamic tree protocols are presented. In Section 5 the two protocols are combined to obtain the final protocol. Concluding remarks are brought in Section 6.

2. Dynamic Self-Stabilizing Systems

2.1. Static Systems and Protocols - A distributed system *SYS* consists of n processors, denoted by $P_1, P_2, \cdots, P_n$. processor resides on a node of the system's *Communication Graph*. The communication graph may be directed, meaning that for each link the two processors that are attached to it agree on its direction: one is attached to the link tail and the other to its head. Two processors which reside on two neighboring nodes of the communication graph are called *neighbors*. Neighbors communicate between themselves by the use of *shared registers*. Every shared register is atomic (serializable) with respect to **read** and **write** operations, that is all **read** and **write** actions to the same register can be serialized in time. processor P_i is a RAM (as defined in first section of [AHU74]). The processors are *anonymous*, i.e. they do not have identities. The program of each processor is composed of atomic steps. An atomic step of a processor consists of an internal computation followed by a **read** or a **write** action. processor has an internal register called *program counter* (*pc*). The *pc* of each processor always points to the next atomic step to be exe-

cuted by that processor.

A protocol for a system *SYS* is a collection of programs of all processors in *SYS*. When no ambiguity occurs, it is convenient to identify processors with their programs. Hence from now on we refer by P_i to the i-th processor together with its program. One can look at such a processor as a state-machine. A state, s_j, of a processor, P_i, is defined by the contents of its memory, including the *pc*. Denote by S_i the set of states of P_i. Given a protocol *Pr* for a system *SYS*, we associate with each shared register r the set of symbols Σ_r that can be stored in r (the set Σ_r is not necessarily finite). A *configuration* of *SYS* is a vector of states of all processors and the content of all registers. Denote by $C=(S_1 \times S_2 \times \cdots S_n \times \Sigma_1 \times \cdots \times \Sigma_m)$ (where m is the number of shared registers in the system), the set of all possible configurations of *SYS* with the protocol *Pr*.

Without loss of generality we assume that at any given time exactly one atomic step is executed in the entire system. Let c_1 and c_2 be two configurations of *SYS*, where c_2 is reached from c_1 by a single atomic step of a single processor. We denote this fact by $c_1 \rightarrow c_2$. An *execution* of *Pr* is an infinite sequence of configurations $(c_1, c_2, ..)$ such that $c_i \rightarrow c_{i+1}$ for $i=1,2,\cdots$. A *schedule* is a sequence of processor numbers. For any execution E of *SYS*, we define the *schedule* of E as the sequence of processor numbers, which correspond to the order of the atomic steps taken by the processors during E. Note that an execution is uniquely determined by its initial configuration and by its schedule. For example if E is started by an action of P_j then the index j is first in its corresponding schedule. An (infinite) schedule S is *fair* if every processor number appears in S an infinitely often.

2.2. Self Stabilization - In this section we define the self stabilization property for distributed protocols. The definition suggested here includes the one in [Di74], but appears to be more general.

Let **T** be the task of the protocol i.e. Mutual Exclusion, Constructing spanning tree ect. A specification of a task **T** is given by a set $L(T)$ of *legitimate sequences of configurations* (The i'th legitimate sequence of configurations is called in short lsc_i). In the sequel we define L(*Spanning Tree*) and L(*Mutual Exclusion*).

Definition: A protocol *Pr* for *SYS* is *self stabilizing* for a task **T** if, starting from any configuration $c_i \in C$ every infinite fair execution of *Pr* has a suffix that belong to $L(T)$.

NOTE: Self Stabilization appears sometimes with no requirement from the scheduler to be fair, [Di74]. The only requirement from the schedule is that it always select an "privileged" processor. "Privileged" processor in the model of [Di74] is a processor that a certain boolean function of its state and the states of its neighbors is true. Whenever

the schedule select such a privilege processor, the processor makes a "move" i.e. changes its state.

In mutual-exclusion protocols it is obvious that whenever the system starts a lsc_i then the schedule becomes fair. However in the stabilizing period of the execution the schedule may not be fair. In our model "privileged" processor is a processor that when it is activated successively sufficiently many times it changes a communication register value. It can be shown that even if the schedule selects only the privileged processor all the protocols presented here are self stabilizing. When a not privileged processor is selected there is no influence on the communication registers value and hence this selection can be eliminated from the schedule.

2.3. Dynamic Systems - Once we deal with self stabilizing systems it is very natural to allow the system to change dynamically. The changes we allow are any kind of addition or deletion of an edge or a node of the communication graph G. Changes to a self stabilizing system should be regarded as just another kind of transient bugs. In other words, the self stabilization property of the system should be strong enough to allow the system to stabilize itself some time after the last topology change is done. In this context we want a topology change to be a very local matter. We would not wish, for instance, that due to a topology change in one part of the system many processors in other parts of the system will change their programs. Also we would not wish the program of a processor to depend on its location in the network. Therefore, it is very natural to require that the program of a processor depends only on the number and orientation of its attached links.

Dijkstra [Di74] has noticed that when all the processors are identical, there are situations in which no self stabilizing protocol for mutual exclusion exist, since there is no way to break symmetry of symmetric configurations. In our model even the existence of the schedule does not help: Assume a communication graph in which each processor has the same number of incoming and outgoing links, hence the same program, starting with all the processors in the same state we can schedule them for **read** actions and then **write** actions so they will **read** and **write** the same forever. To accommodate this requirement we adopt his approach, and assume that the system contains, exactly one of its processors as a *Special Processor*. Our protocols should be correct for any possible choice of special processor. Those considerations are summed up in the following **Uniformity** condition

(3) [Uniformity]

(3a) Exactly one processor in *SYS* is a special processor.

(3b) The program of each processor, depends only on its being or not being a special processor, the number of

its neighbors and the orientation of its adjunct edges (if exist).

(In dynamic protocols, the numbers of neighbors may be changed and is considered as a parameter of the program of each processor).

3. A Spanning Tree Protocol

In all the protocols presented in this paper every pair of neighbors, P_i and P_j communicate by using two shared registers, r_{ij} and r_{ji}. Processor P_i (P_j) writes into r_{ij} (r_{ji}) and reads from r_{ji} (r_{ij}). As mentioned before all these registers are serializable with respect to read and write operations. A processor *owns* all the shared registers it can write to. Each processor orders the links attached to it in an arbitrary way. We denote by α_i the order defined by P_i on its adjacent links, and by α the collections of all these ordering, $\alpha = (\alpha_1, \cdots, \alpha_n)$. Note that α_i induces in a natural way an ordering of the neighbors of P_i.

Our spanning tree protocol is a distributed BFS protocol. The output spanning tree is a BFS tree of the communication graph of *SYS*. This spanning tree is rooted at the special processor of the system. For obvious reasons this special processor is called the *root* processor.

A graph may have more then a single BFS tree rooted at the same node. Let $G(V,E)$ be a graph with orderings $\alpha = (\alpha_1, \alpha_2, \cdots \alpha_n)$ of the neighbors of each node $v_i \in V$ and a root $v_1 \in V$. Define the *First BFS Tree* of G to be a BFS tree, rooted at v_1. In case a node, v_i of distance $d+1$ from v_1 has more than one neighbor of distance d from v_1, v_i is connected to the first neighbor according to α_i among all its neighbors whose distance from v_1 is d. The protocol always produces the First BFS Tree of the system's communication graph, rooted at the *root* processor and with respect to the (arbitrary) orderings α of the neighbors of every processor.

Each communication register r_{ij} is composed of two fields. The first field, denoted by $r_{ij}.father$, is a binary field. The second field denoted by $r_{ij}.distance$ is a positive integer field. Once the output tree is constructed it is encoded by means of the communication registers as follows: If the value of $r_{ij}.father$ is 1 then (the node of) P_j is the "father" of (the node of) P_i in the tree. At the same time $r_{ij}.distance$ holds the distance between P_i and the *root* processor.

Each processor P_i keeps an internal variable for each neighbor. The internal variable, corresponding to the neighbor P_j is denoted by ir_{ji}. This variable stores the last value of r_{ji} that was read by P_i. The two fields of ir_{ji} are denoted by $ir_{ji}.father$ and $ir_{ji}.distance$ respectively. The code of the protocol, for the *root* and for the other

processors, appears in Figure 3.1. In writing the code for processor P_i we assume that it has k neighbors, denoted by $P_1, P_2, \cdots, P_k$ and ordered according to α_i.

```
Root: do forever
          for m:=1 to k do write r_im :=<0,0>;
      od

Other: do forever
(*)       for m:=1 to k do ir_mi := read (r_mi);
          first:=TRUE;
(**)      dist := min (ir_mi.dist)+1;
          for m:=1 to k
          do
              if first and r_mi.distance=dist-1
              then
                  write r_im :=<1,dist>;
                  first:=FALSE;
              else
                  write r_im :=<0,dist>;
          od
      od
```

Figure 3.1: The Spanning Tree Protocol for P_i.

In order to prove that the protocol is self stabilizing we first have to define the set L (*Spanning Tree*) of *lsc*. We choose to include in this set every sequence of configurations that, each configuration of it, encodes the First BFS Tree as defined above.

We now present a lemma which essentially shows that the protocol presented above is self stabilizing relative to L (*Spanning Tree*).

Lemma 3.1: Let P_i be an arbitrary processor whose distance from the *root* is l and let P_j be an arbitrary neighbor of P_i. Let c_0 be an arbitrary configuration and let S be an arbitrary fair schedule. For every index t denote by c_t the configuration reached by the system after it is activated t times starting from configuration c_0 and using the schedule S. For every distance $d \geq 0$ there exist t_d such that: For every $t \geq t_d$ c_t satisfies the following assertions:

(a) If $l \leq d$ then $r_{ij}.distance = l$.

(b) If $l \leq d$ then $r_{ij}.father$ has the "right" value. That is: if P_j is the first neighbor of P_i (using α_i) of distance $l-1$ from the *root* then $r_{ij}.father = 1$, and otherwise $r_{ij}.father = 0$.

(c) If $l < d$ then $ir_{ji} = r_{ji}$.

(d) If $l > d$ then $r_{ij}.distance > d$.

(e) If $l > d$ then $ir_{ji}.distance \geq d$.

Proof: We prove the Lemma by induction over d. In the proof we use repeatedly the fact that due to the fairness of S every processor is activated in S infinitely often.

Base Case (d=0):

Assertion (a): The only node of distance 0 from the root is the root itself. Assume that the root has k neighbors. After k activations of the root its registers hold the value <0,0>. The values stored in the registers of the root will not be changed any more.

Assertion (b): It is easy to see that all $r_{ij}.father$ of the root processor holds the value 0 after the first k activation.

Assertion (c): This assertion holds vacuously for the base case, since there are no processors of distance -1 from the root.

Assertion (d): Note that all values stored in the *distance* fields of all registers are non-negative. Therefore the value of the variable *dist* computed in line (**) of the protocol of all non-root processors is always >0. In particular whenever a value is written in any register by a non-root processor its *distance* field ≥ 1. Again, By the Fairness of S each processor eventually writes in all its registers. Once this happens the *distance* fields of all registers of all non-root processors remain positive forever, hence assertion (d) is also satisfied.

Assertion (e): Trivial, the value within any $ir_{ji}.distance$ of any processor ≥0.

Induction Step:

Let C_t be a configuration for which assertions (a)-(e) are satisfied for some integer $d > 0$. We show the existence of subsequent configuration C_τ ($\tau > t$) for which assertion (a)-(e) hold for $d+1$.

Assertion (a-b): Observe the configuration C_t. Assertions (a),(d) and (e) of the induction hypothesis guarantee that: all the processors in distance d holds the value d in there communication registers, and all the processors in distance $\geq d+1$ holds greater values. Let P_i be a processor in distance $d+1$ from the root. Tracing the behavior of P_i and its neighbors starting in C_t with arbitrary schedule we can conclude: After the first serial **read** actions of P_i, line (*), P_i must find that the minimal value of its neighbor registers is d and therefore **write** the distance $d+1$ in all its $r_{ij}.distance$. By the induction hypothesis no changes occur in the registers of the processors that are in distance d and the value of the rest neighbors of P_i will always be $> d$. From the above two facts we can conclude that the line (**) always get the same value i.e. $d+1$ and hence assertion (a) is satisfied. Moreover after P_i first serial **read**

actions P_i finds its first neighbor that hold the distance d and make it its father. Hence assertion (b) holds too.

Assertion (c): At this step we have already show that all the processors in distance $d+1$ eventually write $d+1$ in all there register and this value is fixed forever. In addition by the induction hypothesis all processors in distance d or less has fixed value in there registers. By using the fairness of the schedule we can be sure that any processor P_j in distance d reads the values of its neighbors so its internal variables are equal to the communication registers of its neighbors.

Assertion (d-e): Let P_i be an arbitrary processor of distance $>d+1$. The neighbors of P_i are all of distance $\geq d+1$ from the root. By assertion (d)-(e) of the induction hypothesis starting from C_t and onwards any neighbor of P_i, P_j, satisfies $r_{ji}.distance \geq d+1$. Therefore, whenever P_i execute line (*) and line (**) after C_t the value assigned to the variable *dist* is $>d+1$. Once P_i write to all its registers, assertion (d) is satisfied for P_i. The same holds for all processors of distance $>d+1$ from the root. Moreover whenever this is done the first serial read actions thereafter makes $ir_{ji}.distance \geq d+1$ of any processor P_i of distance $>d+1$. □

Corollary 3.2: The protocol presented above is self stabilizing relative to the set L (*Spanning Tree*). (Figure 3.2)

4. Mutual-Exclusion Protocol for Dynamic Tree System

This protocol runs on systems whose communication graph is a directed tree, rooted at the (special) root processor. Let P_f be an arbitrary processor. The sons of P_f are ordered from left to right by some arbitrary order α_i. Each processor has a portion of its code called the *Critical Section*. The aim of the protocol is to coordinate processor activity such that at any given time at most one processor executes its critical section and each processor enter the critical section infinitely often. The (single) processor executing the critical section is called the *privileged* processor. When an *privileged* processor completes execution of its critical section it passes the privilege to one of its neighbors.

Once the system start the suffix that belongs to L (*Mutual Exclusion*), execution of the protocol proceeds in phases. In each phase each processor becomes privileged at least once. The first privileged processor in each phase is the root. Following its first activation the root (recursively) passes the privilege to (the subtrees rooted at) its sons in a left to right order. Whenever a processor P_i becomes privileged it executes its own critical section and then passes the privilege to its leftmost son. Once the privilege is passed to all processors in the subtree rooted at this son it is returned to P_i. Subsequently the privilege is passed to the second from left son of P_i and so on. The phase ends when the rightmost son of the root returns the privilege to the root itself. Hence execution of each phase corresponds to a *DFS* tour of the whole tree.

To enable repeating executions of phases, we think of the tree in the beginning of each phase as colored in either white (0) or black (1). Each phase recolors the tree from its current color to the opposite color. Define a (1-0)-phase ((0-1)-phase) as a phase in which the tree is recolored from 0 to 1 (from 1 to 0). Let $e=(P_f \rightarrow P_s)$ be a link of the communication tree directed from a father P_f to its son P_s. The link is implemented using two communication registers, the *instruct* register, r_{fs}, is written to by P_f and read from by P_s. The *confirm* register, r_{sf}, is written to by P_s and read from by P_f. Both registers have a *color* field whose possible values are 0 and 1. In the sequel we refer to the value stored in the *color* field of a register r as the color of r. In case the color of r_{fs} is equal to the color of r_{sf} we say that e is *balanced* and that its color is the common color of its two registers, otherwise e is *unbalanced*. The privilege is passed from P_f to P_s by unbalancing e. The privilege is returned from P_s to P_f by rebalancing e. In addition to its *color* field each *instruct* register has a binary *close* field which is essential in assuring correctness of the protocol in the presence of read/write atomicity.

Let P_i be an arbitrary processor. For each neighbor P_j, P_i keeps in its local memory two internal variables, ir_{ji} and or_{ij}. These internal variables function as images of the color of r_{ji} and r_{ij} respectively. Whenever P_i reads r_{ji} it assigns the read value into ir_{ji}. Similarly, whenever P_i writes into r_{ij}, it assigns r_{ij} the contents of or_{ij}. Assume that P_i has l sons. Its state set S_i is defined by the values of the variables $or_{i1}, \cdots, or_{il}$ and $ir_{1i}, \cdots, ir_{li}$ corresponding to its outgoing links. In case P_i is not the root, its states are also defined by the value of the variables ir_{if} and or_{if} corresponding to its incoming link. The states in S_i satisfy the following specifications:

(1) For some j_0, $1 \le j_0 \le l$, either the value of or_{ij} is 0 if $j \le j_0$ and 1 if $j > j_0$, or the value of or_{ij} is 1 if $j \le j_0$ and 0 if $j > j_0$. When $j_0 < l$, the registers r_{ij_0} and r_{ij_0+1} are called the *left border register* and *right border register*, respectively.

(2) If P_i is not a root processor and if j_0 of (1) above is smaller than l then the value of or_{if} is equal to that of or_{il}.

(3) Any combination of values of $ir_{1i}, \cdots, ir_{li}$ and ir_{fi} is allowed.

The set C of system configurations is the cartesian product of the all the state sets S_i. The values of the communication registers are arbitrary (but legal, i.e. can be stored in the corresponding registers). Note that we allow configurations in which the value of the register r_{ij} is not equal to the value of the corresponding internal variables or_{ij} and ir_{ij}. In other words: It is not assumed that a processor has an apriori knowledge of the actual value written in any register.

The code of the protocol, for the root and for the other processors, appears in Figure 4.1. The code is written for P_i with l sons, denoted by $P_1, P_2, \cdots, P_l$. The father of P_i is P_f. Any processor that executes line 15 or 24 is a privileged processor. We define an *active* processor as processor that all its outgoing links are balanced and its incoming link is unbalanced. (The root processor is *active* whenever all its outgoing links are balanced and any leaf processor (i.e. processor without any sons) is *active* whenever its incoming link is unbalanced).

In the code several predicates are used. These predicates are defined over the possible values of internal variables only. Those predicates are:

```
1   Root: do forever
2       for m:=1 to l do write r_im := <or_im,0>;
3       for m:=1 to l do ir_mi:=read(r_mi);
4       if (all_out_link_balanced) then instruct_next_son;
5   od

6   Other: do forever
7       for m:=1 to l do write r_im := <or_im,0>;
8       write r_if:=or_if;
9       repeat    {try to read the instruct register of your incoming link}
10        ir_fi:=read(r_fi);
11      until ir_fi.close = 0;
12      for m:=1 to l do ir_mi:=read(r_mi);
13      if (all_out_link_balanced) and (in_link_unbalanced)
14        then   {you are privileged}
15          if done then begin or_if:=ir_fi; write r_if:=or_if  end; {balance incoming link}
16      else instruct_next_son;      {unbalance next outgoing link}
17  od

18  Procedure instruct_next_son:    {make your next son active by unbalancing its incoming link}
19  if exists_border then j := index_of_right_of_border_register
20  else j := 1;
21  write r_ij.close := 1;
22  ir_ji := read (r_ji);
23  if (all_out_link_balanced) and (in_link_unbalanced)
24      then begin or_ij :=not or_ij; write r_ij := <or_ij,0>; end
25  else write r_ij.close := 0;
```

Figure 4.1: The Mutual-Exclusion protocol for Dynamic Tree Systems.

(1) *all_out_links_balanced* holds if $or_{im} = ir_{mi}$ for $m = 1, \cdots, l$. (this holds vacuously for a leaf, i.e. when $l = 0$).

(2) *exists_border* holds if there exists j, $1 \le j < l$ such that $or_{ij} \neq or_{ij+1}$.

(3) *in_link_unbalanced* holds if $or_{if} \neq ir_{\Lambda}.color$. (this predicate is always true for the root processor).

(4) *done* is true when for $m = 1, \cdots, l$, $or_{im} = or_{i1}$ and $or_{if} \neq or_{i1}$ (i.e., when all sons have already been instructed).

(5) The function *index_of_the_right_of_border_register* returns the index of the *instruct* register that is in right of the border.

In order to prove that the protocol is self stabilizing we first have to define the set L (*Mutual exclusion*) of lsc. Any $lsc_i \in L$ (*Mutual Exclusion*) satisfies the following:

[Exclusion]: In each configuration $c_k \in lsc_i$ at most one processor is in the critical section.

[Fairness]: During lsc_i each processor is in the critical section infinity often.

An *arbitrary fair execution* $E = c_0, c_1, \cdots$, is an execution which starts from an arbitrary configuration, $c_0 \in C$, and using an arbitrary fair schedule, S. In the following lemmas we use E to denote arbitrary fair execution.

Lemma 4.1

If during E the colors of all registers in the system are constant then there exists a configuration c_t in E such that for every subsequent configuration c_u ($u \ge t$) the value of the *close* field of all instruct registers in the system is 0.

Proof:

We prove the lemma by showing that for every processor in the system P_i there is an index $o(i)$ ($o(i)$ depends on E) such that in the configuration $c_{o(i)}$ of E, the value of the *close* field of all the instruct registers of P_i is 0 and that this value is not changed in any configuration in E passed $c_{o(i)}$. The proof proceeds by induction on d, the distance of P_i from the root.

Base Case d=0:

In this case P_i is the root processor. The program for the root processor is a single loop. By the fairness of S it holds that during E P_i executes this loop infinitely often. In particular lines 2 and 3 of the root's program are executed infinitely often. In line 2 the root sets the value of the *close* field of all its instruct registers to 0. In addition in line 2 the root sets the values of the *color* field of every instruct register r_{im} to the value of its internal variable or_{im}.

In line 3 the root reads the confirm registers of all its sons; for every register r_{mi} P_i sets the value of the internal variable ir_{mi} to the value it read from r_{mi}. Let $c_{o(i)}$ be the first configuration in E after P_i executed lines 2,3 for the first time. In $c_{o(i)}$ the values of all *close* fields in all instruct registers of the root are 0 and for every son P_m of the root $ir_{mi}=r_{mi}$ and $or_{im}=r_{im}.color$. By the assumption that no *color* field is changed during E, any further read action does not change the value of ir_{mi}. By the same assumption the value of or_{im} is not changed either.

The only place where the value of the *close* field of any register is set to 1 is inside the procedure *instruct_next_son*. To complete the proof it suffices to show that following $c_{o(i)}$ the root never executes this procedure. Assume towards a contradiction that *instruct_next_son* is executed by the root starting from configuration c_k, $(k \geq o(i))$. We reach a contradiction by showing that in this case the root also changes the color of some instruct register. Following $c_{o(i)}$ *instruct_next_son* is executed only if the predicate *all_out_links_balanced* holds. While executing *instruct_next_son* the root computes the index j of the processor which will be "instructed". After that the root checks whether the link connecting it to P_j is balanced. We have already proved that in every configuration that follows $c_{o(i)}$ it holds that $ir_{ji}=r_{ji}$ and $or_{ij}=r_{ij}.color$, hence in c_k and every subsequent configuration all the out links of the root are indeed balanced. In particular the link between the root and P_j is balanced. In this case the root instructs P_j by changing the color of r_{ij}, contradiction.

Induction Step:

We assume correctness of the lemma for all processors of distance d from the root. Let P_i be an arbitrary processor of distance $d+1$ from the root. We show the existence of a configuration $c_{o(i)}$ after which the values of all *close* fields in all instruct registers of P_i are 0 forever.

The program of a non-root processor P_i is composed of a main loop and an inner loop. In the inner loop P_i waits until it finds that the value of all *close* fields of its father P_f is 0. Once this happens P_i proceeds to execute the main loop. P_f is in distance d from the root. By the induction hypothesis there is a configuration in E, $c_{o(f)}$, such that in every configuration of E following $c_{o(f)}$ it holds that the value of the *close* field of the register r_{fi} is 0. By the fairness of S, P_i is activated infinitely often during E. In particular P_i is activated infinitely often passed $c_{o(f)}$. Since after $c_{o(f)}$ the value of the *close* field of r_{fi} is always 0, P_i does not get stuck in its inner loop passed $c_{o(f)}$ and executes its main loop infinitely often. Let $c_{o(i)}$ be the configuration reached by the system right after P_i executes lines 7 to 12 for the first time after $c_{o(f)}$. By an argument similar to the one used in the proof of the base case it can be shown that in $c_{o(i)}$ the value of all *close* fields of P_i is 0 and that these registers will not be changed in E any more.

□

Lemma 4.2

In every arbitrary fair execution the color of at least one register in the system is changed.

Proof:

In the proof we use the following Observation: in every arbitrary configuration there is at least one active processor. When all the links in the system are balanced then the root processor is active otherwise consider an unbalanced link, $(P_f \rightarrow P_i)$, with maximal distance from the root then P_i is an active processor.

Assume towards a contradiction that E is a fair execution during which no processor changes the color of any of its registers. By lemma 4.1 there exist a configuration c_t $(t \geq 0)$ in E such that for every configuration c_u $(u \geq t)$ every *close* field holds 0.

case 1: In c_0 all the links of the root are balanced

By the assumption no *color* field is changed. Hence in any configuration in E the outgoing links of the root are balanced. The program for the root processor consist of a single loop. By the fairness of S this loop is executed infinitely often during E. In particular lines 2 and 3 are executed infinitely often. Whenever these two lines are executed the root reads the communication registers of all its sons. Once these registers are read the root "discovers" that all its outgoing links are balanced, and unbalances the *right_of_border* link by changing the color of its register, contradiction.

case 2: In c_0 there is at least one unbalanced link.

By the assumption no *color* field is changed. Hence any unbalanced (balanced) link in c_0 remains unbalanced (balanced) during E. Consider an unbalanced link $(P_f \rightarrow P_i)$ of maximal distance from the root. By the definition of $(P_f \rightarrow P_i)$ the incoming of P_i is unbalanced and all the outgoing links of P_i are balanced in any configuration of E. The program of the non-root processor P_i is composed of a main loop and an inner loop. After the system reaches c_t the value of $r_{fi}.close$ is zero and the main loop is executed infinitely often. Let c_u $(u \geq t)$ be the first configuration after P_i executed lines 7 to 12. It can be shown that in the subsequent execution after c_u P_i discovers that the predicates *all_out_links_balanced* and *in_link_unbalanced* holds. Once P_i discovered that the value of those predicates is true it either changes the color of the right of border register (in case the predicate *done* does not hold) or it balances the incoming link (in case the predicate *done* holds), con-

tradiction. □

Corollary 4.3:

In every arbitrary fair execution the color of at least one register is changed infinitely often.

Proof:

The proof is immediate by a repeated application of lemma 4.2. □

Next we show that a non-root processor, P_i, may make only a bounded number of changes in the color of its communication registers, unless the color of its father that is related to P_i's incoming link is changed too.

Lemma 4.4:

Let P_i be a processor with k outgoing links, and an incoming link $(P_f \rightarrow P_i)$. Let m_{ij} be the number of times P_i changes the color of r_{ij}. Then for any arbitrary fair execution E, $\Sigma_j\ m_{ij} \leq 2 \times k+3$, between two changes of $r_{fi}.color$.

Proof:

First we count all the possible changes in the color of the registers of P_i due to lines 7 and 8 in the code. Notice that during E, once these lines are executed the value of any or_{ij} is equal to the value of r_{ij} since an atomic step consist internal computation and read or write action. Hence no further changes in the color of the registers occur by execution these lines. The number of these changes is at most $k+1$. Next consider lines 15 and 24. P_i can execute these lines at most once before it reads the value of r_{fi} into ir_{fi}. If P_i sees the link balanced (i.e. $ir_{fi} = or_{fi}$) then P_i does not execute lines 15 or 24 before the value of ir_{fi} is changed. Otherwise, P_i may execute lines 15 or 24 only when it finds that it is an active processor. In this case, P_i repeatedly executes the procedure *instruct_next_son*, until all its outgoing links are balanced and have the same value as $r_{fi}.color$. Before this happens, P_i can execute line 15 at most once and line 24 at most k times as follow: first write in r_{if} and then in all its *instruct* registers. Here we use the fact that *instruct_next_son* repeatedly changes the values of the variables or_{ij}, in a cyclic way. Once *done* gets the value true, P_i balances its incoming link by writing in r_{if}. After this link is balanced by P_i, P_i does not change a color of a register any more until the incoming link is unbalanced again by P_f. □

Lemma 4.5:

Let P_i be a processor with k outgoing links. Let $(P_i \rightarrow P_s)$ be an arbitrary outgoing link of P_i. Let m_{ij} be the number of times P_i changes the color of r_{ij}. Then for any arbitrary fair execution E, $\Sigma_j\ m_{ij} \leq 2 \times k+3$, between two changes of r_{is}.

Proof:

First we count all the possible changes due to lines 2 or 7,8, similarly to the above lemma P_i can change its communication registers due to those line execution once i.e. $k+1$ times. Next consider lines 15 and 24. P_i can execute those lines at most once before it reads the value of its son, P_s, communication register. If P_i see $(P_i \rightarrow P_s)$ link unbalanced (i.e. $ir_{is} \neq or_{is}$) then P_i does not executes lines 15 or 24 any more. Otherwise, P_i may execute line 15 or 24 only when it finds that it is an active processor. In this case, P_i repeatedly executes the procedure *instruct_next_son*, until it unbalances the link $(P_i \rightarrow P_s)$. Before this happens, P_i can execute line 24 k times and execute line 15 once. (first write in r_{is} and then in all its other registers. Here we use the fact that *instruct_next_son* repeatedly changes the colors of P_i's registers in a cyclic way). Once P_i unbalances the link to P_s no farther changes are possible since the predicate *all_out_links_balanced* is not true. □

Corollary 4.6:

In every arbitrary fair execution:

(1) The color of every register is changed infinitely often.

(2) There is a configuration c_{t_1}, such that for any subsequent configuration c_t, $t > t_1$, the color of any register r_{ij} is equal to the color of or_{ij}.

(3) Every processor changes its communication registers, executing each line 15 (except the root processor) and line 24 (except processors with no sons), infinitely many times.

(4) There is a configuration c_{t_2}, such that any processor has already executed the first line of its code. (line 1 or 6).

proof:

(1) By corollary 4.3, and lemmas 4.4, 4.5, in every fair execution, every register is changed infinitely often.

(2) By (1) above every register is assigned infinitely often. The color of any register r_{ij} is equal to its correspondings internal variable or_{ij} after the first assignment since an atomic step is composed with internal computation and read or write action.

(3) Following c_{t_1} every register is changed infinitely often. Hence every processor changes the color of each of its registers, executing both line 15 (line 15 is not executed by the root processor) and line 24 (line 24 is not executed by processors without sons) infinitely many times.

(4) By the code of the processors, between any two executions of lines 15 or 24 the processor executes the first line of its code. By (3) above during E every processor executes line 15 or 24 infinitely often hence every processor executes the first line of its code infinitely often. □

A crucial property which every link in the system in our protocol has is a "proper link behavior". Consider a link $(P_f \rightarrow P_s)$, with associated r_{fs} and r_{sf}. Eventually it is true that, whenever P_f executes line 24, it makes the link unbalanced. Similarly, whenever P_s executes line 15 it makes the link balanced.

Following c_{t_1}, the status of any link $(P_f \rightarrow P_s)$ is completely determined by ir_{sf}, $r_{fs}.color$, r_{sf}, $ir_{fs}.color$, the four tuple of these values is called the link values in configuration c. Apriori, the link values may have any combination of binary values. Observe that the two central bits represent the actual colors of the link's registers, while the two leftmost (rightmost) bits represent the local information of P_f (P_s) on those colors (following c_{t_1} $r_{fs}.color$ is equal to the value of or_{fs}). We use the notation (1 0 → 0 0),[P_f **read**],(0 0 → 0 0) to denote the change of the link status from (1 0 → 0 0) to (0 0 → 0 0) by a **read** operation of P_f. We show that the link behaves properly, according to the protocol by proving the following Lemma:

Lemma 4.7:

In every arbitrary fair execution of the protocol, eventually the link values of every link in the network are changed repeatedly according to the following *legitimate* cycle:

(0 0 → 0 0),[P_f **write**],(0 1 → 0 0),[P_s **read**], (0 1 → 0 1),[P_s **write**],

(0 1 → 1 1),[P_f **read**],(1 1 → 1 1),[P_f **write**], (1 0 → 1 1),[P_s **read**],

(1 0 → 1 0),[P_s **write**],(1 0 → 0 0),[P_f **read**],(0 0 → 0 0).

Proof:

By corollary 4.6 there exist a configuration c_{t_1} in which the color of every r_{ij} is equal to or_{ij}. Following c_{t_1} every link that its link values are in the legitimate cycle has always only one possible change in its link values, the one that transfers it to the next link values in the cycle. Hence, it is sufficient to show that the links values do converges to one of the legitimate values.

We proved that any (non leaf) processor executes line 24 infinite times, hence executes the procedure instruct_next_son infinitely often. Every such execution consists of closing an instruct register, reading the value of the corresponding confirm register, writing (if necessary) to the instruct register and then opening it. Denote a

sequence of configurations from the configuration in which r_{fs} is closed to the first subsequent configuration in which it is open as a *close period* of r_{fs}. Observe P_f outgoing link ($P_f \rightarrow P_s$) and trace its values during the close period of r_{fs}. During this close period P_s can change the color of r_{sf} (executing line 15 or 24) at most once since after any such change, P_s must read r_{fs}. During the close period of r_{fs} any try of P_s to read r_{fs} keeps P_s in its inner loop and does not affect the link values. When P_f closes r_{fs} the link values may be in arbitrary combination. Consider the following cases of schedules within the close period of r_{fs}:

Case 1: P_s does not change r_{sf} during the close period.

Case 2: P_s changes r_{sf} before P_f read from it.

Case 3: P_s changes r_{sf} after P_f read from it.

In the two first cases P_f reads the updated value of r_{sf} and make the link unbalanced (if it is not already unbalanced) hence the link values in the end of the close period are (1 0 → 1 ?) or the symmetry case (0 1 → 0 ?) (the question mark stands for either 0 or 1). Fortunately the above link values are in the legitimate cycle. The third case start the same as the two first i.e. in the configuration that follows P_f writing, the link values are either (1 0 → 1 ?) or (0 1 → 0 ?). However in this case P_s changes the value of r_{sf} to be the same as ir_{fs} and balances the link. Hence the link values in the end of the close period are either (1 0 → 0 0) or (0 1 → 1 1) that are in the legitimate cycle. (Figure 4.5) □

By the above lemma there exist a configuration c_{t_3}, $t_3 \geq t_1$ and $t_3 \geq t_2$, such that every link holds the legitimate cycle values. We proceed the proof by combining the above results in order to show that eventually the system converge to a specific configuration c_{t_4}.

Lemma 4.8:

Following c_{t_3} there exist a configuration c_{t_4} in E such that the color of all the communication registers and internal variables is 0 (1).

Proof:

Define a phase of a processor P_i in E, as the sequence of configuration starting with a configuration in which all the color of the internal variables of P_i are 0 (1) and ends in the first subsequent configuration in which they are 1 (0).

E is a fair execution, hence following c_{t_3} each processor is activated infinitely often. By corollary 4.6 and by the code of the processor every processor has infinite number of phases in E, specially the root processor. In the end

of $k \leq d$ (d is the depth of the tree) phases of the root processor all the internal variables and registers in the subtree, that includes the processor that are in distance k or less from the root, have an uniform color. We proved in Lemma 4.7 that after the first closing and opening of r_{f_s} the link values converges to one of the legitimate cycle values. Notice that from this stage of the execution whenever P_f reaches (0 0 → ? ?) the link values are (0 0 → 0 0) and P_s is waiting for farther instructions. When the root processor begins its first phase (following c_{t_1}) all its outgoing links are balanced and colored 0 (1). When it finishes this phase all its outgoing links are colored 1 (0). During the phase of the root processor all its sons change the color of their confirm register to fit its change. When a processor changes the color of its confirm register then the color of all its internal variables is equal to the color it is going to write. Moreover the processor must execute a serial write actions to its instruct registers so they fit this color. After one more phase of the root we can be sure that all the processors in distance two from the root did changed their confirm registers color to fit the root's color and so forth. □

We proceed by showing that in any configuration that follow c_{t_4} at most one processor is privileged. Recall that a privileged processor in our protocol is a processor that executes line 15 or 24 i.e. the critical section.

Lemma 4.9:

Let E be an arbitrary fair execution. Every configuration c_t that follows c_{t_4} ($t \geq t_4$) in E satisfies the following assertions:

(a) Exactly one active processor, P_k, exists.

(b) For any $P_i \neq P_k$ and every neighbor of P_i, P_j, the color of ir_{ji} is equal to the color of r_{ji}.

(c) For any neighbor of P_k, P_i, except may be one, the color of ir_{ik} is equal to the color of r_{ik}.

Proof:

First recall that by corollary 4.6 (2) and due to the fact that $t \geq t_4 \geq t_1$ for every i, j $r_{ij} = or_{ij}$. We proceed in the prove of the Lemma by induction over t.

Base Case ($t = t_4$):

Assertion (a): In c_{t_4} all the incoming links are balanced hence the root processor is the only active processor.

Assertion (b) and (c): By the definition of c_{t_4} the color of all registers and internal variables is 0.

Induction Step:

We assume those assertions hold for the configuration c_t and we prove, that they hold for the configuration that immediate follows it in E, c_{t+1}.

Case 1: Let P_i be an arbitrary non active processor of c_t. Assume that c_{t+1} is reached from c_t by a single step of P_i. During this atomic step P_i does not change the color of its communication registers or internal variables.

Proof: (Case 1)

By the definition of non active processor it holds that it has at least one unbalanced outgoing link and/or a balanced incoming link. By assertion (b) at least one of the predicates all_out_link_balanced or in_link_unbalanced is false. Therefore the only **write** actions, in a color field of a register, taken by P_i are those in lines 2,7,8. Since $t \geq t_1$ those actions do not change the color of any communication registers. Moreover by assertion (b) any **read** action does not change the corresponding internal variable. Hence in c_{t+1} the assertions (a) to (c) hold.

Case 2: Let P_k be the active processor of c_t. Assume that c_{t+1} is reached from c_t by a single step of P_k. then assertions (a) to (c) hold.

Proof: (Case 2)

consider the following cases:

Case 2.1: P_k **reads.** The **read** action assigns to one internal variable ir_{jk} the color of r_{jk}. If the assignment to ir_{jk} changed its color then by assertion (c) in c_{t+1} every internal variable has the same color as its register. Otherwise no changes occur in the value of a register or internal variable. Hence assertion (a) to (c) hold in c_{t+1}.

case 2.2: P_k **writes.** The only lines in the code that contain an atomic **write** that might change the color of a register are 15,24. A processor can execute lines 15 or 24 when the predicates *all_out_link_balanced* and *in_link_unbalanced* hold. By the definition of the active processor it holds that the incoming link of P_k is unbalanced and all outgoing links of P_k are balanced. Using assertion (c), in c_t the color of at most one internal variable is not equal to the color of its corresponding register. However if the color of one internal variable is not equal to the color of its corresponding register, one of the predicates *all_out_link_balanced* or *in_link_unbalanced* does not hold. Hence if P_k executes lines 15 or 24 the colors of any internal variable in c_t is equal to the color of its corresponding register. When P_k executes line 15 or 24 it either makes its incoming link balanced or one of its outgoing links unbalanced hence in c_{t+1} P_k is not active. By the observation of Lemma 4.2 there is always at least one active processor, hence

the processor, P_j, that is related to the register that was changed by P_k is the only active processor in c_{t+1} and has exactly one internal variable ir_{kj} that is different from r_{kj}. Hence assertion (a) to (c) hold in c_{t+1}.

□

Lemma 4.10:

In every configuration c_t that follows c_{t_4}, $t \geq t_4$, at most one privileged processor exist.

Proof:

By Lemma 4.9 in c_t assertions (a) to (c) hold. Using assertion (b) we prove that any non active processor is not privileged. Assume toward contradiction that there is a non active processor, P_i, that is privileged in c_t. By assertion (b) of lemma 4.9, in c_t, P_i holds the updated values of the register of its links. Since P_i is not active at least one of the predicates *all_out_link_balanced* or *in_link_unbalanced* does not hold. Hence when P_i executes line 13 or 23 it skips line 15 or line 24 respectively; this is contradiction to the assumption that P_i executes line 15 or 24 in c_t. By assertion (a) of lemma 4.9 exactly one active processor exist in c_t, hence at most one privileged processor exist in c_t.

□

Corollary 4.11:

The protocol is self stabilizing relative to the set L(Mutual Exclusion).

Proof:

By lemma 4.10 at most one privileged processor exist hence the **[exclusion]** property hold.

By corollary 4.6 each processor executes line 15 or 24 infinitely often hence the **[fairness]** requirement holds. □

5. Mutual-exclusion Protocol for dynamic networks

In this section we combine the protocols from the previous sections to obtain a mutual exclusion protocol for dynamic networks. Denote the spanning tree protocol by Pr_1. Denote the mutual exclusion, on tree structure, protocol by Pr_2. The communication registers in the system are divided into two main fields the first used by Pr_1 and the second by Pr_2. Any processor P_i execute the program of each protocol one after the other first a step in the program of Pr_1 then a step of Pr_2 and again Pr_1 and so forth. During execution of Pr_2 the processors uses the values of the field of the communication registers related to Pr_1 as a given parameter. Pr_2 uses these values as a parameter for the number and orientation of adjunct links in the spanning tree.

It is clear that in a fair schedule each processor executes each Pr_i infinitely often. Hence Pr_1 stabilizes and reaches a $lsc_i \in L$ (Spanning Tree). Whenever Pr_1 reaches $lsc_i \in L$ (Spanning Tree) it encodes by the communication registers value a spanning tree. From this moment Pr_2 uses the correct parameters for the number and orientation of its adjunct links in the spanning tree. Hence, Pr_2, the mutual-exclusion protocol on the spanning tree converges to $lsc_j \in L$ (Mutual Exclusion). It is clear that Pr achieves mutual-exclusion on arbitrary connected network.

NOTE: combined protocols are interesting since their correctness proof might be easier. In sequential research we will give a wield definition to self stabilizing combined protocols.

6. Concluding Remarks

Self stabilization was defined as an independent property of protocols. The viability of the new definitions was demonstrated by presenting self stabilizing protocol for mutual exclusion in dynamic systems with arbitrary communication graph. Our protocol is correct with respect to the most relaxed kind of atomicity.

The spanning tree protocol has the following property: The size of the registers of a processor is unbounded. This property matters only for dynamic systems where the final number of processors is unknown. If an apriori bound N on the number of processors in the system is known then the size of each register is $O(\log N)$ which looks very reasonable.

REFERENCES

[AHU74] A.V.Aho J.E. Hopcroft, J.D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974, pp. 5-33.

[BGW87] G.M. Brown, M.G. Gouda, and C.L. Wu: A Self-Stabilizing Token system, *Proc. of the Twentieth Annual Hawaii International Conference on System sciences*, 1987, pp.218-223.

[Bu87] J.E. Burns: Self-Stabilizing Rings without Demons, Technical Report GIT-ICS-87/36, Georgia Institute Of Technology.

[BP87] J.E. Burns and J. Pachl: Uniform Self-Stabilizing Rings, *Aegean Workshop On Computing, 1988, Lecture notes in computer science 319*, pp. 391-400.

[Di74] E.W. Dijkstra: self-stabilizing systems in spite of distributed control, Communications of the ACM 17,11 (1974), pp. 643-644.

[Di82] E.W. Dijkstra: self-stabilizing systems in spite of distributed control (EWD391), Reprinted in Selected Writing on Computing: A Personal Perspective, Springer-Verlag, Berlin, 1982, pp. 41-46.

[Di86] E.W. Dijkstra: A belated proof of self-stabilization. Distributed Computing 1, 1 (1986), pp. 5-6.

[Kr79] H.S.M. Kruijer: Self-stabilization (in spite of distributed control) in tree-structured systems, Information Processing Letters 8,2 (1979), pp. 91-95.

[La84] L. Lamport: Solved problems, unsolved problems, and non-problems in concurrency, *Proc. of the Third ACM symp. on Principles of Distributed Computing* (Vancouver, BC, Aug. 1984), 1984, pp.1-11.

[Tc81] M.Tchuente: Sur l'auto-stabilision dans un r'eseau d'ordinateurs, RAIRO Inf. Theor. 15 (1981), pp. 47-66.

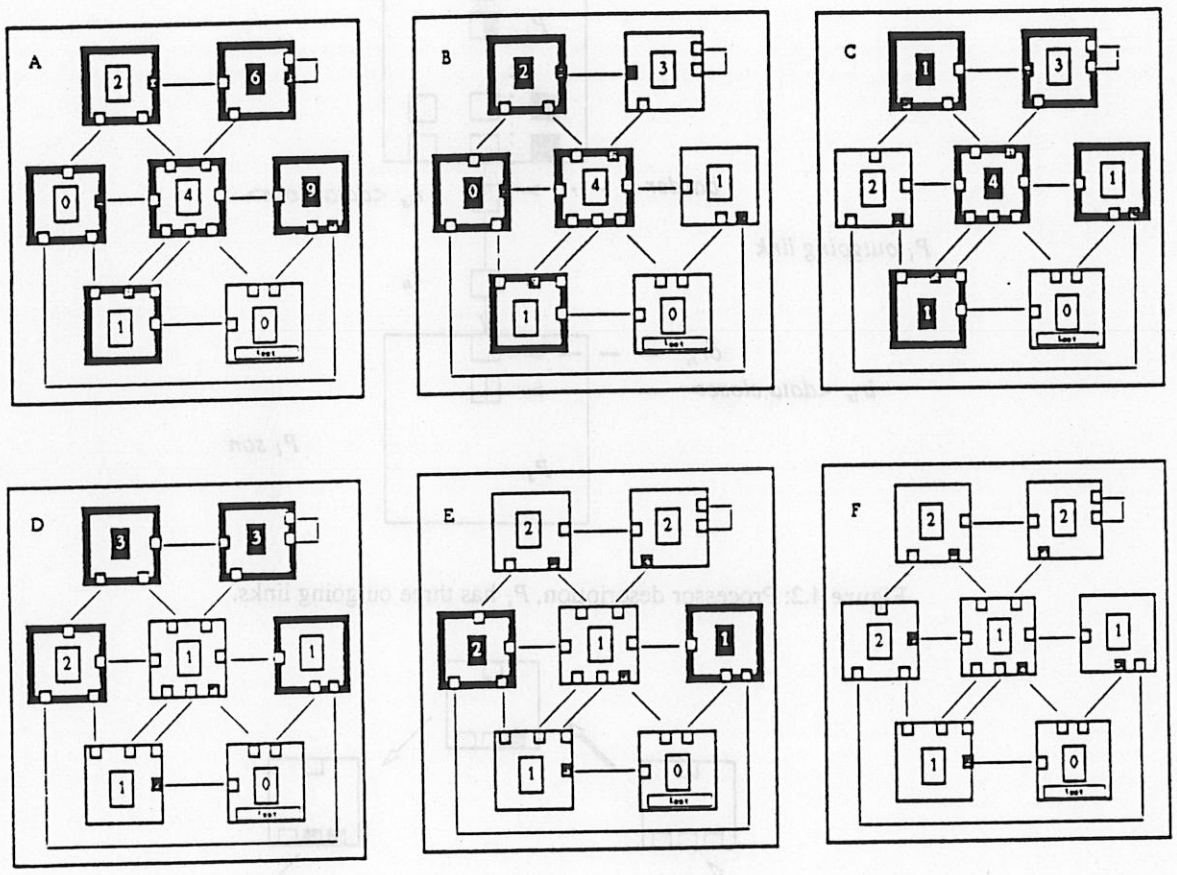


Figure 3.2: Execution sample- The frames of enabled processors are marked. The numbers in the center of each processor stand for the value of the *distance* fields of its internal registers. The number in the processors that are activeted each time are marked also. The order of execution is marked by the letter in the upper left corner.

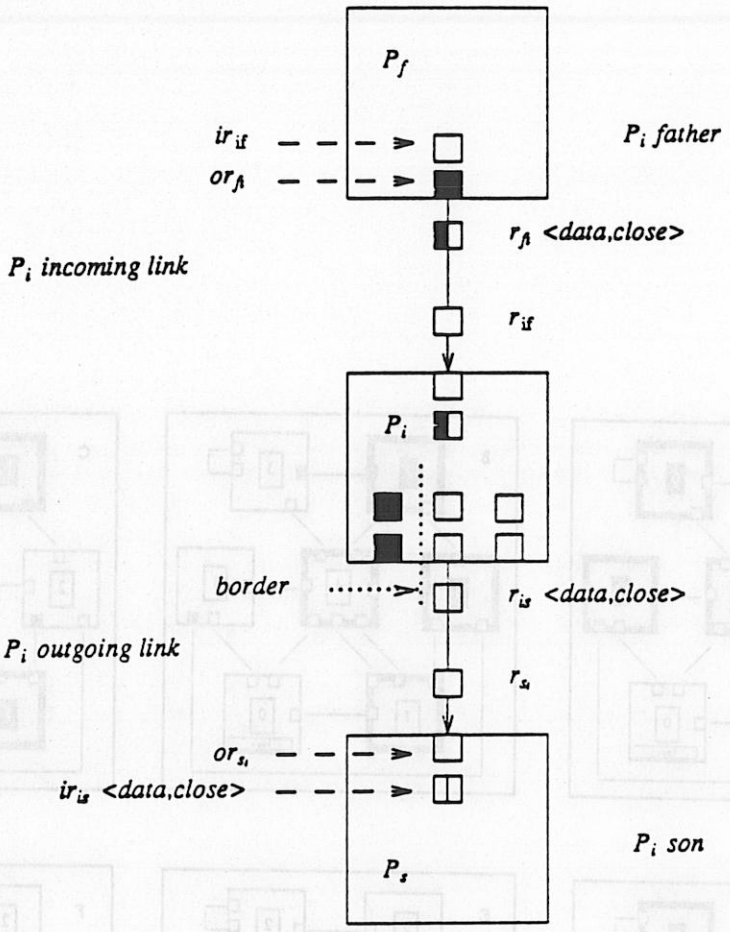


Figure 4.2: Processor description, P_i has three outgoing links.

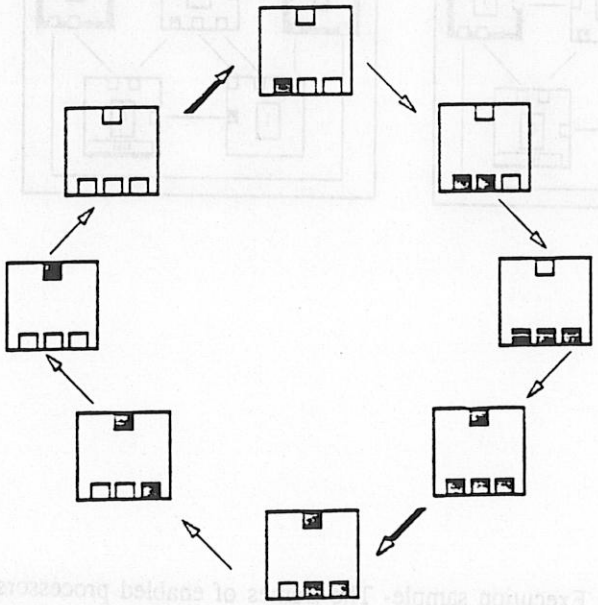


Figure 4.3: Transition function for a processor with k=3 outgoing links and father P_f. The internal variables or_{ij} $j \in (1,2..k,f)$ with their values (colors) are marked in every processor state. Every transition occurs when $or_{if} \neq ir_{fi}.data$ and $or_{ij} = ir_{ji}$ for every $j \in (1,2...k)$

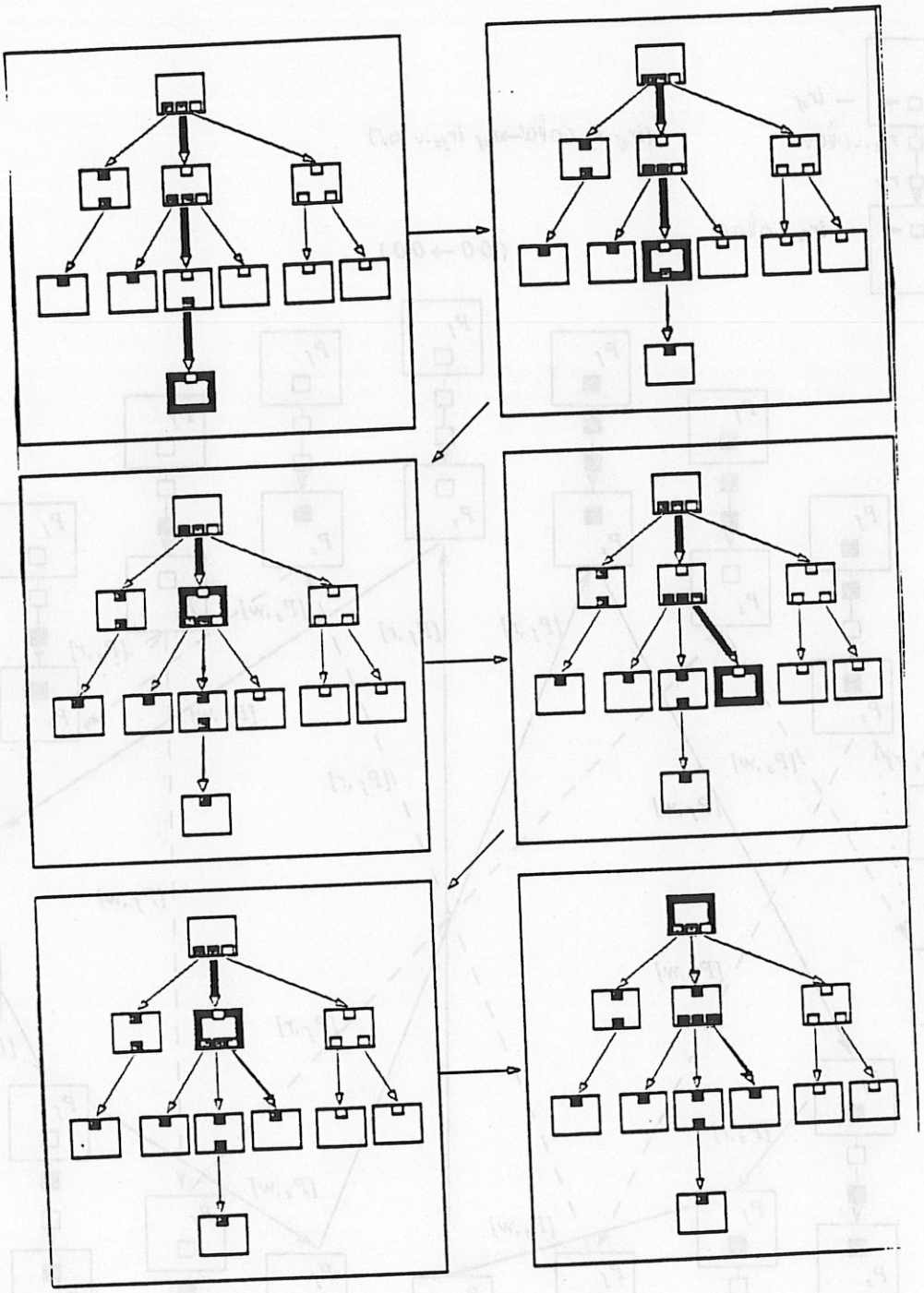


Figure 4.4: Execution sample: The active processor and the links on the separating path are marked (bold). All internal variables or_{ij} are shown with their value (color).

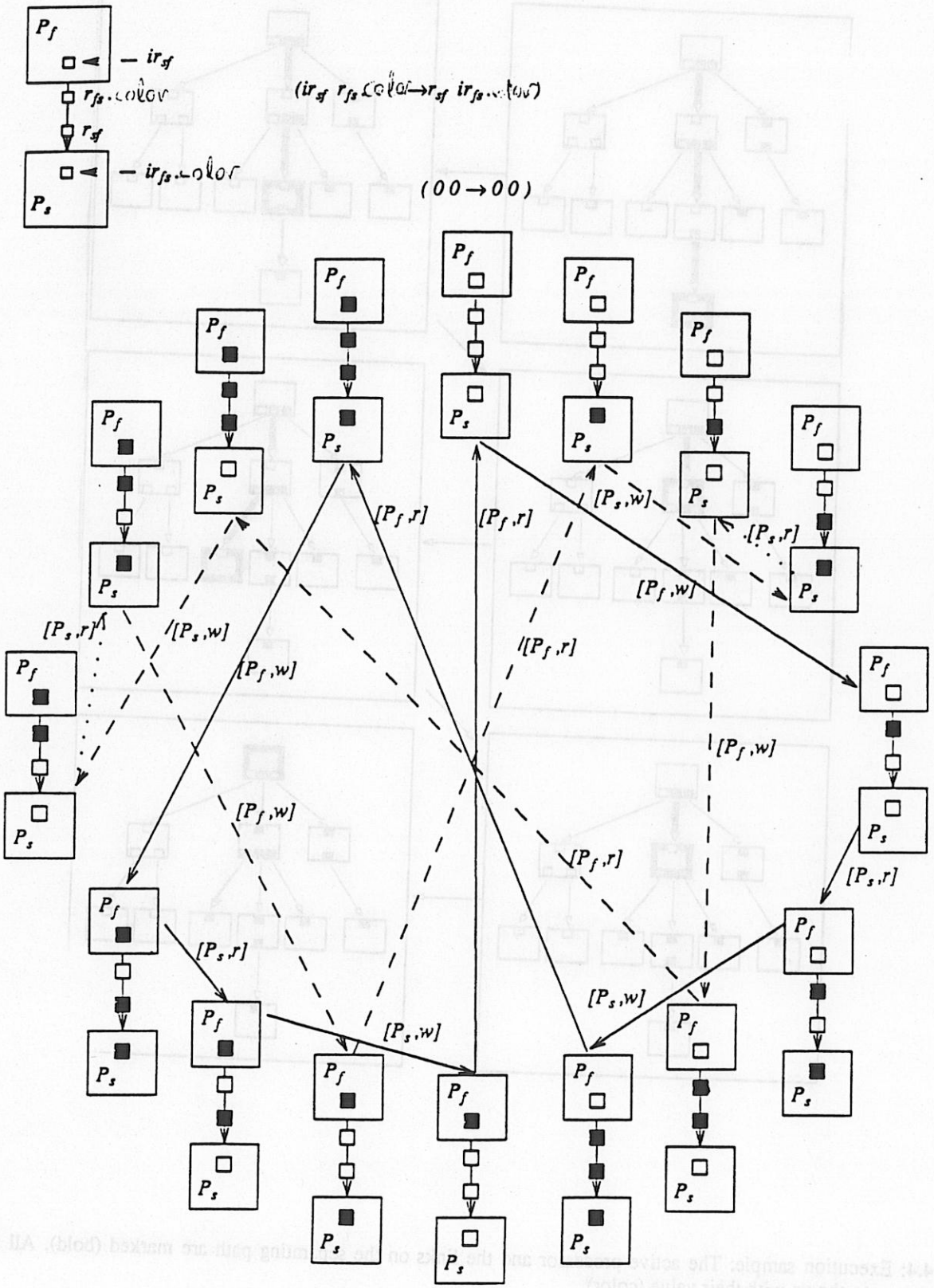


Figure 4.5: Link values transformation. The dashed lines are the transformations within the "illegal" link values. The use of the *close* field eliminates the transformations corresponding to the dotted lines.

Self-stabilizing Extensions for Message-passing Systems

Shmuel Katz*
Technion and IBM T.J. Watson Research Center

Kenneth J. Perry†
IBM T.J. Watson Research Center

September 27, 1989

Abstract

Self-stabilization is an abstraction of fault tolerance for transient malfunctions. Intuitively, a self-stabilizing program resumes normal behavior even if execution begins in an illegal initial state. Previously, self-stabilization was handcrafted into a program. Our goal is a superimposition by which a non-stabilized program can be mechanically augmented to yield a self-stabilizing one. A precise definition is given of a program being a self-stabilizing extension of a non-stabilizing program. We also clarify which properties are guaranteed to eventually hold in such an extension. The computational model used is that of an asynchronous distributed message-passing system. We contrast the difficulties of self-stabilization in this model with those of the more common shared-memory models. For processes connected in an arbitrary graph of FIFO channels, we demonstrate a superimposition based on repeated global snapshots that creates a self-stabilizing extension for a wide class of programs.

*Technion, Israel Institute of Technology, Haifa 32000, Israel. Bitnet: katz@techsel
†IBM Research, PO Box 704, Yorktown Heights, NY 10598. Internet: kjp@ibm.com

1 Introduction

The concept of a *self-stabilizing* distributed program was introduced by Dijkstra [Dij74]. Self-stabilization requires that a program executing with an arbitrary initial state (including arbitrary control locations) eventually must reach a *legitimate* state and thereafter remain in legitimate states. In other words, the property "the program is in a legitimate state" is stable (see [CL85]) and eventually true.

Self-stabilization is an abstraction of fault-tolerance for a model in which transient faults corrupt data, messages, and location counters (but not the program code). Each such fault is assumed to be followed by a long period without additional faults. The difficulty of self-stabilization lies mainly in the fact that a process has no way of distinguishing between an initial state and one that occurs during the computation. For example, the assertion $\{x = 0\}$ does not always hold following the code $x := 0$ since the initial control state may be just after this statement, without it having actually executed. Similarly, control of a process can be just after a **send** instruction in a message-passing model, without a message having actually been sent and with no indication of this fact.

Although a number of self-stabilizing programs have been published, careful and subtle reasoning must be employed to verify their correctness. The elegance of such custom-made self-stabilizing programs is often impressive. However, a more generally applicable approach in our view requires separating the self-stabilization from the original algorithm design. This separation reduces the complexity of design and encourages the reuse of valuable techniques that are otherwise hidden when encoded in particular algorithms.

In the following section, the computational model is described informally and compared with other approaches. In Section 3 we define the concept of a program being a *self-stabilizing extension* of a non-stabilizing program. Section 4 contains a theorem clarifying which properties are guaranteed to eventually hold in such an extension. Then, using these definitions, in the subsequent sections we give a methodology for creating a self-stabilizing program that is an extension of a non-stabilizing program. In particular, self-stabilizing versions of global snapshot and reset algorithms are superimposed on a non-stabilizing program to yield a self-stabilizing

extension.

2 Relation to other models

The computational model used in this paper is that of an *asynchronous message passing system.* A *message passing system* is a collection of *processes* that are connected by FIFO communication *channels.* Processes may exchange values only by transmitting *messages.* Most previous work in this area uses a shared-memory model. The system is *asynchronous* in that there are no bounds on either relative process speeds, message delivery time, or channel capacities. We do assume that every message sent is eventually received and that every statement whose guard remains true is eventually selected for execution. In the initial state of such a system, process states, location counters, and channels may have arbitrary values.

2.1 Previous results

Dijkstra[Dij74] demonstrated the concept of self-stabilization through an example that identified the legitimate states as those with exactly one enabled operation (called a *privilege*). His solutions thus are examples of self-stabilization for a form of mutual exclusion (or, equivalently, for a token ring with a single token).

As originally presented, self-stabilization is a property of a program, relative only to the definition of legitimate states. Most published algorithms [Dij74,Lam84,GE88,BP89,BGW89] follow Dijkstra in that they deal with mutual exclusion or token-passing. They present a program and then demonstrate that the self-stabilization is built-in. Lamport's mutual exclusion algorithm[Lam84] is the exception in that he creates a self-stabilizing program by inserting statements into a non-stabilizing program.

Since the original definition of Dijkstra, emphasis has been put on devising algorithms, rather than providing a precise semantics for self-stabilization. In [Lam84], Lamport defined a *transient malfunction* behavior of a shared-memory algorithm. This is an execution in which each process executes a single *malfunction* operation that arbitrarily assigns values to its variables, and then begins normal operation at any point in its code,

following the code of the algorithm. Then an algorithm is self-stabilizing for a property A if A eventually holds for every transient malfunction behavior of the algorithm. He leaves open the domain of the values in the malfunction operation.

2.2 Difficulties of the model

As will be demonstrated, there are several phenomena that must be overcome in order for a program to be self-stabilizing in an asynchronous message-passing environment. The primary one is *apparently sent messages*. In a local process state intended to follow the sending of a message, there is no way to determine whether (a) the desired message has actually been sent or (b) this is merely a false impression, and the local state is part of the initial global state. This could cause the system to deadlock with processes waiting for response messages that will never come because the request message was never actually sent. One approach to overcoming this difficulty is causing a message to be spontaneously sent "again" from time to time (which might be sending it for the first time after a false impression). Such repeated messages are called *prods* and are used in this model to ensure liveness. Prods can sometimes be avoided if some other messages are guaranteed to be sent without first requiring receipt of a message.

This need to spontaneously send messages is one symptom of the differences between a message-passing model and a shared-memory model, when self-stabilization is considered. In shared memory, the fact that a variable has an unexpected value can be determined by reading (testing) the variable and only writing to it when a problem is detected. In a message-passing context, a process cannot examine the contents of channels and thus must either use prodding or wait for another message, with the danger of deadlock in the system.

Another difficulty is of *infinite propagation* of false channel information. Since the channels can initially have arbitrary messages on them, the effects of these messages must not be to indefinitely generate new messages that are possible only in response to the misleading incoming messages. Otherwise an acceptable global state might never be reached. At some point, these injurious initial messages and injurious new messages generated due to them must be purged from the system.

2.3 Comparison with other fault models

In some respects self-stabilization seems easier than other forms of fault-tolerance: every process is guaranteed to participate in the algorithm, to execute only according to its code, and never to simply cease participating. This differs from, for example, Byzantine failures [LF82,LSP82], where some processes can ignore the intended program and take arbitrary (usually malicious) steps, or even crash failures, where some processes can cease participating. However, these other models allow only a subset of the processes to fail and the correctness criteria only relate to the processes that have not failed. In self-stabilization, all processes may have arbitrary initial states, yet every process must stabilize so that legitimate global states occur. Moreover, in this model no process can ever "depend" on the values in its local memory: even accurately counting the number of incoming messages is impossible since the number in the counter may have no relationship to the number of messages actually received.

A more precise statement of this last difficulty is that although self-stabilization is globally achievable, no process in the system can ever *know* that the system has self-stabilized. That is, there is no local state of a process for which it is guaranteed that all consistent global states are legitimate (since the initial state could be globally illegitimate, but have the "knowing" local state as its projection on some process). The practical implication of this inability to know when self-stabilization has occurred is that no process can ever switch from an inefficient but self-stabilizing version of an algorithm to a more efficient version that ignores self-stabilization. At the point of switching, the process would have to know that self-stabilization has already taken place, which is impossible. Many of the terms used here informally are defined more precisely in the following section.

3 Semantics and Definitions

Our goal is a means of creating a self-stabilizing program from a non-stabilizing one. Toward this end, we precisely define the notion of what it means for one program to be both self-stabilizing and "an extension" of another program. The definitions below are for the interleaving execution model of concurrent systems.

A *local state* of a process is an assignment of values to the local variables and the location counter. The *global state* of a system of processes is the cross product of the local states of its constituent processes, plus the contents of the channels, where here we assume FIFO queues of messages. The channels are not part of the local state of any process. Each local variable and location counter has a *domain* of values that it may assume. The semantics of individual operations are usually given by presenting the possible atomic steps (transitions) and the appropriate changes in the states. These are not described here. An *execution sequence* of program **P** is a (possibly infinite) sequence of global states in which each element follows from its predecessor by execution of a single atomic step of **P**. One state is a *successor* of another in an execution sequence if it appears later in the sequence. Finite execution sequences represent *terminated* computations. The set of all possible execution sequences of program **P** is denoted $sem(\mathbf{P})$ and defines the semantics of **P**. Note that no assumption is made about the initial state of an execution sequence, except that all values are from the appropriate domain.

The definition of self-stabilization depends on what are considered the "legitimate" states of a program. We could define the legitimate states of program **P** as those satisfying a predicate (called **P**'s *specification*). Alternatively, these states could be defined as those obtainable from a "normal" execution. For the purpose of defining self-stabilizing extensions of a program, we choose the latter. Of course, the normal initial states could also be defined using a predicate but, for the sake of concreteness, we choose a particular common possibility. Those initial states in which the location counter of each process is 0 and all channels are empty are said to be *normal*; the *legal* (i.e., intended) semantics of program **P** is the subset of $sem(\mathbf{P})$ containing only sequences with normal initial states and is denoted by $legsem(\mathbf{P})$. Every global state in a sequence from $legsem(\mathbf{P})$ is also defined to be legal.

Note that the set of legal global states is, in general, much smaller than the set of possible global states, since the latter includes many combinations of values that do not arise in any legal execution sequence. The *illegal execution sequences* are those that have initial illegal states. There is yet a third class of execution sequences: those with initial states that are legal, but not normal, e.g., with control not at the beginning of the code. These

are clearly suffixes of legal sequences. In the continuation, note that a suffix of a sequence can be the sequence itself.

Definition 1 *Program* **P** *is* self-stabilizing *if each sequence in* $sem(\mathbf{P})$ *has a non-empty suffix that is identical to a suffix of some sequence in* $legsem(\mathbf{P})$.

In other words, from some point on every computation is identical to a legal one. Now we turn to the relation between a program and an extension. A *projection* of a global state onto a subset of the variables and the messages on the channels is the value of the state for those variables and messages.

Definition 2 *Program* **Q** *is an* extension *of program* **P** *if for each global state in* $legsem(\mathbf{Q})$ *there is a projection onto all variables and messages of* **P** *such that the resulting set of sequences is identical to* $legsem(\mathbf{P})$, *up to stuttering*[1].

Definition 3 *Program* **Q** *is a* self-stabilizing extension *of program* **P** *if* **Q** *is self-stabilizing and also is an extension of* **P**.

That is, considering only those portions of **Q**'s global states that correspond to **P**'s variables and messages, the legal semantics of **P** and **Q** are identical if repetitions of states are ignored. Moreover, **Q** is self-stabilizing for all its computations. When begun in normal initial states, **P** and **Q** have the same possible executions (relative to **P**'s state, ignoring location counters) and **Q** resumes the intended semantics when begun in an illegal initial state. For legal but not normal initial states, **Q** merely executes the suffix of a legal computation, relative to **P**. Note that no correspondence is required among the illegal computations of **P** and **Q**, or among the location counters.

In particular, program **P** may terminate with control locations after all statements of its program (or, equivalently, at *halt* statements), but **Q** generally has no such halting locations. Otherwise, **Q** could have an initial state with its control halted, but with an illegal global state relative to the messages and variables—and thus not be self-stabilizing. The extension for a terminating computation of **P** has execution sequences that eventually repeat a final state of **P** forever, changing only variables not present in **P**.

[1] When comparing sequences, adjacent identical states are eliminated; this is sometimes called the elimination of stuttering.

4 Limits on Self-Stabilizing Extensions

Before demonstrating how a self-stabilizing extension of a program is created, we consider which properties can hold in the extension. A simple example illustrates the potential problems. Suppose that the specification of program **P** is "a 1 is eventually output" and that "1,0,0,..." is the only legal output sequence of **P**. Furthermore, suppose that program **Q** outputs only all-zero sequences when started in an illegal initial state and otherwise produces the same sequence as **P**. Then **Q** is a self-stabilizing extension of **P** but does not eventually satisfy **P**'s specification. The problem is that some sequences in $legsem(\mathbf{P})$ satisfy the specification but have suffixes that do not.

The following theorem characterizes properties that can hold in self-stabilizing extensions.

Theorem 1 *If* **Q** *is a self-stabilizing extension of* **P**, *and* A *is an assertion in (future) linear temporal logic over variables and messages of* **P**, *then:* A *holds for a suffix of every execution sequence of* **Q** *iff for each sequence in* $legsem(\mathbf{P})$ *either* A *is true in the final state (for finite sequences) or* A *is infinitely often true.*

Proof Since A is in the future fragment of linear temporal logic, for a suffix S of a sequence T, the truth of A for S is independent of the states in T but not S. That is, the states before S are irrelevant to the truth of A on S.

$\Rightarrow$. In $sem(\mathbf{Q})$, every suffix of an execution sequence is in itself an execution sequence. Since A is true for a suffix of every execution sequence of **Q**, if the sequence is infinite, A is true infinitely often along the sequence, because it only relates to the suffix, as noted above. Similarly, if the state relative to **P** repeats forever from some point on, A must hold in this "final" state, since it too is an execution sequence. Since **Q** is a self-stabilizing extension of **P**, by definition, for each sequence in $legsem(\mathbf{P})$ there is a sequence in $legsem(\mathbf{Q})$ that is identical w.r.t. **P**'s state. A must be true for these sequences of $legsem(\mathbf{Q})$ and their suffixes, since they are a subset of the sequences in $sem(\mathbf{Q})$, and thus it must also hold infinitely often (or in the final state) for the sequences in $legsem(\mathbf{P})$.

$\Leftarrow$. If A is infinitely often true for the infinite sequences in $legsem(\mathbf{P})$, then it is infinitely often true for the infinite sequences in $legsem(\mathbf{Q})$ that correspond to them. Similarly, for the finite sequences, if A is true in the final state, then $\mathbf{Q}$ will have a sequence in $legsem(\mathbf{Q})$ that eventually repeats the final state of $\mathbf{P}$ forever, changing only variables and messages not in $\mathbf{P}$, and thus not affecting the continued truth of A in that sequence. Every sequence in $legsem(\mathbf{Q})$ has a projection onto one in $legsem(\mathbf{P})$. Since $\mathbf{Q}$ is self-stabilizing, every sequence in $sem(\mathbf{Q})$ has a suffix identical to one in $legsem(\mathbf{Q})$, and thus satisfying A. ∎

Note that properties that cannot hold in a self-stabilizing extension can often be rephrased as ones that do. For example, if $legsem(\mathbf{P})$ contains only the infinite sequence of values (0,1,0,1,...), then an assertion that eventually the number of 1's and of 0's will be equal may never become true (e.g., if in $\mathbf{Q}$ there are sequences that begin with (1,1,1,0,1,0,1,...)). On the other hand, an assertion that infinitely often there is a state followed by a later state for which in the states between them there are equal numbers of 1's and 0's is true for every sequence in a self-stabilizing extension.

5 Superimposition of Self-stabilization

In this section, some general considerations on superimpositions are introduced. We also outline a superimposition to provide a self-stabilizing extension for a non-stabilizing program. A crucial component of this superimposition, a self-stabilizing global snapshot algorithm, is described in the following section and it is proven correct in Section 7. Another component, a reset algorithm, is described in Section 8. The complexity of the superimposition is analyzed in Section 9.

A *superimposition* is a collection of fragments of code and generic transformations that is to be merged with another algorithm (called the **basic** algorithm) in order to produce a combination satisfying properties not seen in the basic algorithm alone. The additional code (called the **imposed** statements) is interleaved with the result of transforming the code of the **basic** algorithm. Examples are termination detection algorithms, global snapshots, and deadlock detection algorithms that are superimposed onto basic programs that do not treat the issue handled by the superimposition.

Superimposition has been considered in [BF88] and [Kat87].

The idea of the superimposition that we present is to repeatedly (attempt to) take snapshots without interfering with the underlying basic computation. The results of these snapshots are analyzed in a distinguished initiator process, and, if necessary, all processes are reset to default values in a way that will eventually guarantee a legal global state. The properties of the extended program that guarantee self-stabilization are (1) eventually a snapshot that accurately reflects the state of the system will be successfully completed (even though the initiator process cannot know when this occurs), (2) eventually, if an illegal state is revealed in the accurate snapshot, a reset wave will succeed in establishing a legal global state, and (3) thereafter, the snapshots will be accurate, and will be of legal global states, so that no further reset operations will be done. The last condition guarantees that eventually the basic computation can proceed, so that a suffix of a legal computation will be computed, relative to the basic computation. The extension will necessarily continue forever taking snapshots interleaved with the basic computation, but these will not interfere with the "real" work in any way.

In defining a superimposition, the class of basic algorithms to which it is applicable should be specified. Under those assumptions about the nature of the basic algorithm, the combination of the superimposition with any algorithm in the class can be proven to satisfy the specification of the superimposition. Thus, in order to define a superimposition that will transform a given program $\mathbf{P}$ into a program $\mathbf{Q}$ so that $\mathbf{Q}$ is self-stabilizing w.r.t. $\mathbf{P}$, some basic restrictions on $\mathbf{P}$ can be articulated.

First, a predicate must exist that can determine whether or not a representation of a global state assembled in the variables of a single process is legal. Self-stabilization has been considered trivial for a single sequential process precisely because such an assumption is common. Second, it is assumed that if repeated accurate snapshots are made, and only legal states are detected, then the system is indeed in a legal state. This follows when the legal states are identified with those that could occur in $legsem(\mathbf{P})$. Just as in [CL], the snapshot taken in the algorithm is a possible ancestor of the present actual global state, and thus the actual state must also be a legal state of $legsem(\mathbf{P})$, as required.

6 A Self-Stabilizing Snapshot Algorithm

6.1 Overview

Our goal in taking snapshots is to eventually cause the variables of some process to contain a representation of a global state of **basic** program **P**, the program on which we perform our superimposition. In the algorithm introduced in this section, process 0 is given the special role of being both the initiator of snapshots and the process whose variables contain its results. The assignment of this role to any process in **P** is arbitrary.

We assume that we are given a system of n processes whose connectivity graph is described by a set **E** of ordered pairs such that $(i, j) \in \mathbf{E}$ if and only if there is a channel from process i to process j. So that snapshots may be taken and assembled in the variables of a single process, it is necessary that **E** describe a strongly-connected graph.

Definition 4 *At any global state σ, a process is said to have an* accurate snapshot for α *if local variables of the process contain a representation of a global state that is a possible successor of α and a possible predecessor of σ.*

Algorithm Snapshot, described in this section, is a self-stabilizing algorithm that permits process 0 to iteratively obtain accurate snapshots for the state that **P** had when each iteration began. The superimposition of this **imposed** algorithm onto **P** is a self-stablizing extension of **P**. A trivial assumption is that the **imposed** messages created by Algorithm Snapshot are syntactically distinguishable from the **basic** messages that are created by **P**. Before introducing the text of the algorithm, we attempt to give the reader an intuitive feel as to how it works.

We emphasize at the start that a self-stabilizing algorithm is only required to *eventually* establish the desired property, and not to establish it *immediately*. It is both acceptable and likely (e.g, when the initial state is not a normal initial state) that some number of inaccurate snapshots will initially occur. The sole requirement is that, from some point onwards, only accurate snapshots of **P** are obtained by process 0.

Chandy and Lamport[CL85] have described an algorithm that, when superimposed onto a **basic** program **P**, takes a single snapshot of **P**'s state.

macro $CL(j)$:
Process $i, i \geq 0$

$\neg initiated$	$\Rightarrow$	[$initiated$:= **true**; $eoc[j]$:= **true**; $eoc[k]$:= **false**, for all $k \neq j : (k,i) \in \mathbf{E}$;[a] $record[k]$:= *nil*, for all $k : (k,i) \in \mathbf{E}$; *basic_state* := state of $\mathbf{P}$ /* Send[b] a *marker* to all $k : (i,k) \in \mathbf{E}$ */]
$initiated \wedge \neg eoc[j]$	$\Rightarrow$	$eoc[j]$:= **true**
$initiated \wedge eoc[j]$	$\Rightarrow$	**skip** Illegal state

[a] A side effect not shown is that process i starts to record in variable $record[k]$ all basic messages of $\mathbf{P}$ that are received from each neighboring process k when $eoc[k]$ is **false**.

[b] In a legal state, the *marker* will be piggy-backed onto a *token* message of the algorithm that invokes this macro.

Variables:

- *basic_state* is state of the basic program $\mathbf{P}$.
- *initiated* is a Boolean indicating that initialization has been performed.
- $eoc[j]$ is a Boolean indicating that a *marker* message has been received from j.
- $record[k]$ is the sequence of **basic** messages received from k while $eoc[k]$ was **false**.

Figure 1: Chandy-Lamport snapshot algorithm.

This algorithm, which works in a general FIFO network, appears in Figure 1 as a macro[2] *CL* to be invoked by each process i upon receiving a message from process j. When i receives the very first message (called a *marker*) it saves **P**'s local state in a variable, begins to record the **basic** messages henceforth sent by **P**, and propagates the *marker* to its neighbors. Receipt of subsequent *markers* causes the algorithm to stop recording **basic** messages on the channel on which the *marker* was received. After each process has received one *marker* per incoming channel, it ceases to participate in the snapshot; after all processes cease to participate, the algorithm is terminated.

The fact that Chandy and Lamport's algorithm takes a single snapshot and terminates means that it alone cannot be self-stabilizing, as was pointed out is Section 3. Our solution is to impose a *control* program onto their algorithm; the resulting algorithm takes repeated snapshots of **basic** program **P** by iteratively invoking the single snapshot algorithm (see Figure 2). The control program regulates the iterative procedure by initiating waves of *token* messages. It also insulates the Chandy-Lamport algorithm from any message that would violate its assumptions. For example, an invariant of the single snapshot algorithm is that exactly one *marker* is received on each channel. This assumption can clearly be violated by *marker* messages that may be present in initial states that are not normal. Lastly, the control program establishes the initialization condition ($initiated$ = `false`) for each iteration of the single snapshot algorithm.

As was mentioned earlier, the process chosen to serve the role of process 0 behaves differently in that it both initiates and obtains the final results of each iteration (see Figure 3). A new single snapshot iteration is initiated by creating a new *marker* and its termination is detected when $reported[k]$ becomes true for all k. The *report* message sent by process i to signal local termination also contains a field PIECE that contains the state and channel information that it has recorded. Thus, at termination, process 0 has a representation of a prior global state of **P** in its *piece* variables. A crucial property (proved as Lemma 2) is that nothing may prevent process

[2]A macro, as opposed to a procedure, has no state of its own; it merely modifies the variables of the invoking program. Thus, we can look at macro invocation as an abbreviation for the statements contained in its body, after parameter substitution has been performed.

0 from advancing to the next iteration.

The difficulty in defining the control program is that the arbitrariness of the initial state means that no variable or message may be relied upon to always serve its intended purpose. For example, initially two processes may have different values in the local variable (*Current*) that defines the iteration in which each is participating. This introduces the potential problems of deadlock and the infinite propagation of messages. Deadlock is prevented by introducing a non-reactive statement (RECPROD) to creates the prods referred to in Section 2.2. These prods themselves introduce problems in that it is now possible for messages created in different iterations to simultaneously be in the system. To distinguish between them, we add to each *token* and *report* message an integer field VAL, which contains an iteration number. Infinite propagation is avoided by adding to each message a field PATH, which is a sequence of process identifiers.

6.2 Behavior of processes

At a high-level, the behavior of each process other than 0 in executing Algorithm Snapshot is to"react" to the receipt of a message m, containing the values v, p, and r in the VAL, PATH, and PIECE fields respectively. If predicate **IsNext** recognizes m as a *token* meant to start a new iteration, the process executes statement (NEXT), which changes its iteration counter and invokes macro *CL*. Should m be recognized as a *marker* message (i.e., predicate **IsMarker** holds), the process executes statement (RECSNAP) and invokes the *CL* macro. If neither of the above hold, the message is recognized as a prod (statement (RECPROD)) and is passed on to its neighbors. Additionally, the prod may cause a *report* message to be sent to process 0 if predicate **Finished**, which determines whether local termination has occurred, is satisfied. The correct implementation of these predicates is made non-trivial because of the multitude of global states (particularly the illegal ones) in which they may be evaluated.

Process 0's behavior is different in that it (a) initiates each new iteration of the single snapshot algorithm (by executing statement (START)) (b) non-reactively creates prods by executing statement (GENPROD). (c) saves the information contained in *report* messages by executing statement (RECREPT). In order to make the behavior of process 0 as similar to the

Process $i > 0$, upon receiving *token* message m with VAL = v, and PATH = p from process j:

(RECSNAP)	**IsMarker**	$\Rightarrow$	$[CL(j)$; **Finished** $\Rightarrow$ *Report*$]$
(DEJAVU)	**IsCntl** $\wedge$ **Seen**	$\Rightarrow$	`skip`
(NEXT)	**IsCntl** $\wedge \neg$**Seen** $\wedge$ **IsNext**	$\Rightarrow$	$[$*Current* := v; *Propagate*; *initiated* := `false`; $CL(j)$ $]$
(RECPROD)	**IsCntl** $\wedge \neg$**Seen** $\wedge \neg$**IsNext**	$\Rightarrow$	$[$*Propagate*; **Finished** $\Rightarrow$ *Report*$]$

Variables:

- *Current* is the "iteration number".

Definition of predicates:

- **IsMarker** $\equiv \neg eoc[j] \wedge (\mathrm{v} = Current)$
- **IsCntl** $\equiv \neg$**IsMarker** $\equiv eoc[j] \vee (\mathrm{v} \neq Current)$
- **Seen** $\equiv$ (i appears in sequence p) $\wedge((i \neq 0) \vee (\mathrm{p} \neq 0))$
- **IsNext** $\equiv (\mathrm{v} > Current)$
- **Finished** $\equiv (\mathrm{v} \neq Current) \vee (\bigwedge_{k:(k,i)\in \mathbf{E}} eoc[k])$

Figure 2: Snapshot algorithm for process $i > 0$.

Process 0, upon receiving *token* (or *report*) message m with VAL = v, and PATH = p, (and PIECE = r) from process j:

(SELFSNAP)	**IsMarker**	$\Rightarrow$	$[CL(j)$; **Finished** $\Rightarrow$ *Report*$]$
(SELFSENT)	**IsCntl** $\wedge$ **Seen**	$\Rightarrow$	`skip`
(RECREPT)	**IsCntl** $\wedge \neg$**Seen** $\wedge$ **IsReport**	$\Rightarrow$	$[k :=$ *first*(p); $\neg$*reported*$[k] \Rightarrow$ [*reported*$[k]$:= `true`; *piece*$[k]$:= r]]
(SELFPROD)	**IsCntl** $\wedge \neg$**Seen** $\wedge \neg$**IsReport**	$\Rightarrow$	[**Finished** $\Rightarrow$ *Report*]

Process 0, spontaneously:

(GENPROD)	`true`	$\Rightarrow$	*Start*
(START)	$\bigwedge_{k=0}^{n-1}$ *reported*$[k]$	$\Rightarrow$	[*reported*$[k]$:= `false`, for all $k \geq 0$; *Current* := *Current* + 1; *Start*; *initiated* := `false`; *CL*(0)]

Variables:

- *reported*$[k]$ is a Boolean indicating that process k has ended its participation in the current iteration and that *piece*$[k]$ is k's portion of the global state recorded.

Definition of predicates:

- **IsReport** $\equiv$ (v = *Current*) $\wedge$ **InReportForm**, where **InReportForm** is true only on *report* messages.

Figure 3: Snapshot algorithm for process 0

behavior of other processes as possible, we assume a channel from process 0 to itself so that process 0 may react to its own messages. For example, statement (SELFPROD) is executed when process 0 receives its own prod and statement (SELFSNAP) invokes the *CL* macro when a *marker* is recognized. This assumption is merely a convenience that avoids awkward coding.

A further convenience is the assumed existence of a few trivial macros whose specifications follow. All but *Start* are invoked after process i has received a message m whose PATH and VAL fields are equal to p and v respectively. All variables are local to process i:

- *Start* is executed by process 0 in order to send a *token* message with PATH = 0 and VAL = *Current* to each process k such that $(0, k) \in \mathbf{E}$.

- *Propagate* is executed by process $i > 0$ in order to send a *token* message with PATH = append(p, i) and VAL = v to each process k such that $(i, k) \in \mathbf{E}$.

- *Report* is executed by process $i \geq 0$ in order to send a *report* message with PATH = i, VAL = v, and

  $$\text{PIECE} = \langle basic\_state, record[k], \text{for each } k : (k, i) \in \mathbf{E} \rangle$$

  to process 0 according to some fixed strategy.

  One self-stabilizing way of implementing the necessary strategy is to define for each process a constant, loop-free path to process 0. Upon receiving a *report* message, a process forwards it on the next link in this path. Self-stabilization is achieved by encoding the path in the immutable code rather than mutable variables.

7 Correctness of the Snapshot Algorithm

In this section we formally demonstrate that Algorithm Snapshot is a self-stabilizing algorithm that eventually permits process 0 to iteratively obtain accurate snapshots for the state that **P** had when each iteration began. The proof proceeds by defining a certain class of *good* states, demonstrating that such states are eventually repeated (Section 7.2), and demonstrating that

an accurate snapshot is obtained when the single snapshot iteration begins in such a state (Section 7.3).

Prior to defining the good states, we prove the eventuality of a state with a certain simplifying property.

Definition 5 *Message m' is an* immediate copy *of message m if*

- *m' is sent by some process i as part of the action that it executes upon receiving message m, and*
- *m' is identical to m except that the* PATH *of m' is the concatenation of i to the* PATH *of m.*

"Copy" is the reflexive, transitive closure of the "immediate copy" relation.

Definition 6 *A global state σ in execution sequence s is said to be* purged *if every token and report message m in σ is a copy of a message sent in a prior state of s.*

Recall that the initial state may have messages in the channels; the definition excludes copies of such messages. Note that, without loss of generality, we assume that every process is evaluating the guard of one of the labelled statements in the initial state. We can do this since an initial state that does not satisfy the assumption (and any messages sent as a result) may always be transformed into one that does. We now show the eventuality of a purged state.

Lemma 1 (Eventually Purged) *In every execution of* Algorithm Snapshot *a purged state eventually occurs. The property of being purged is stable.*

Proof The eventuality of a purged state follows by the eventual message delivery assumption and observing that execution of Algorithm Snapshot results in only finitely many copies of any message. The finiteness follows from the fact that *report* messages are sent on loop-free paths to process 0 and that the act of copying a *token* message is guarded by predicate ¬**Seen**, which insures that a process does not copy its own copies. ∎

Thus, we henceforth restrict our attention to execution sequences whose initial state is purged.

7.1 Liveness of Iteration

A fundamental property of our algorithm is now established: no matter what the state, eventually process 0 may execute a (START) statement. Therefore, by fairness, a new iteration always begins.

Lemma 2 (Iteration Liveness) *In any execution of* Algorithm Snapshot, *for any purged state α, there is some successor state σ in which property "reported[k] is true for all k" holds. Moreover, this property holds for each successor of σ until the next* (START) *statement is executed.*

Proof That the property remains stable until the next (START) statement is executed may be seen by observing that *reported*[k] may be falsified only by executing the next (START) statement.

Let v_α denote the value of process 0's *Current* variable in state α. We claim that eventually, there is some successor state of α in which either "*reported*[k] is true for all k" already holds (in which case the lemma holds) or in which every message is a copy of a message created by executing statement (GENPROD) in a successor state of α. In the latter case, each of these messages has field VAL = v_α because process 0 cannot change its *Current* variable until *reported*[k] becomes true for all k. By the continued enabling and execution of statement (GENPROD) and the assumptions of FIFO channels and eventual message delivery, there occurs a state in which every message is a copy of a message created by execution of a (GENPROD) statement in a successor state of α. By inspection of Algorithm Snapshot, we see that repeated reception of these messages results in each process eventually sending a *report* message with VAL = v_α. (Process i must send a *report* because it executes either (a) statement (RECPROD), (b) statement (RECSNAP) or (SELFSNAP), or (c) statement (NEXT) followed by one of the above.). As each *report* is received by process 0, *reported*[k] become true for each k in turn (and remains true until the next (START) statement is executed) until finally a state σ results in which *reported*[k] is true for all k. ∎

7.2 Good states and their eventuality

We now define when a state is good. Intuitively, the good states are the ones in which an iteration is guaranteed to terminate with an accurate

snapshot.

Definition 7 *A global state σ is* good *if and only if*

1. *The value of the Current variable in each process is identical.*
2. *No message in σ can henceforth cause* **IsNext** *to become true, for any process i.*
3. *reported[k] is true for all $0 \leq k < n$.*

In the next lemma, we establish that reaching a good state is unavoidable, no matter what the initial state.

Lemma 3 (Eventuality of good) *In every execution of* Algorithm Snapshot, *every purged state eventually has a good successor state. Moreover, each successor of a good state remains good until the subsequent* (START) *statement is executed.*

Proof Let α be a purged state. We must show that eventually some successor state σ is good. That successor states of a good state remain good until a subsequent (START) statement is executed may be seen by inspection of Algorithm Snapshot.

Let V denote the finite set such that integer $v \in V$ if and only if, in state α, v is the value of either the *Current* variable of some process or the VAL field of some message. By Lemma 2, inspection of the guards, and the assumptions of eventual message delivery and fairness, a state β occurs in which process 0 has just executed a (START) statement and for which v_β, the value of *Current* for process 0, exceeds the maximum value in V. Eventually, for each $i > 0$, there is some successor state of β in which process i receives its first copy of the *token* created at β. By construction of v_β, the *Current* variable of process i is set to v_β as a result of executing statement (NEXT) in this state. Since no (START) statement creating a value greater than v_β may execute until process 0 has received a *report* from each $i \geq 0$, and since process i may send such a *report* only after receiving the first copy of the *token* created at β, eventually some state γ occurs in which the *Current* variable of each process is equal to v_β. The construction of v_β also insures that no message in state γ can henceforth cause **IsNext** to become true when it is received by any process.

Since both these properties are stable until the subsequent (START) statement is executed, let σ be the successor of γ (whose eventuality is guaranteed by Lemma 2) in which *reported*[k] is also true for all $0 \leq k < n$. ∎

We may now combine several previous results to demonstrate the good states repeatedly occur.

Lemma 4 (Recurrence of good) *In any execution of* Algorithm Snapshot, *good states repeatedly occur.*

Proof The proof is by induction. The eventuality of a first good state is assured by Lemmas 1 and 3. Since purged is a stable property, eventually every good state has a good successor, by Lemma 3. ∎

7.3 Good states result in accurate snapshots

In this section, we finally establish the correctness of Algorithm Snapshot, drawing upon the work of previous sections. We show that an iteration that begins in a good state will terminate with process 0 having an accurate snapshot for the state that **P** had when the iteration began. Once we show that Algorithm Snapshot invokes macro *CL* at the proper moments, this will follow from the correctness of the Chandy-Lamport algorithm.

Theorem 2 (Good Snapshot) *In any execution of* Algorithm Snapshot, *eventually process* 0 *repeatedly obtains accurate snapshots for the state in which each* (START) *statement is executed.*

Proof

We have already shown (by Lemma 4) that good states repeat. So we need only show that accurate snapshots are obtained.

Let σ be the good state in which a (START) statement is executed and let v be the value of process 0's *Current* variable in this state. Clearly the immediate successor of σ is not good. We must show that in state γ, the next good state (whose eventuality is insured by Lemma 4), the value of process 0's *Current* variable is equal to $v + 1$ and that the concatenation of *piece*[k] for $0 \leq k < n$ is an accurate snapshot for σ. The first fact is trivially seen by inspecting statement (START). The latter is shown by demonstrating that the (START) statement executed at σ results

in each process receiving the same messages at it would were it executing the Chandy-Lamport algorithm in isolation.

In executing the (START) statement at σ, process 0 performs the same statements as an initiator of the Chandy-Lamport algorithm would, namely, it invokes $CL(0)$ with *initiated* = `false`, creates a new token (with VAL = v + 1), and sends it to each neighbor. When each process receives a copy of this token, we show that its behavior is just as it would be if it too were executing the Chandy-Lamport algorithm in isolation.

First, examine how copies of the *tokens* created at σ are treated by each process i. When process $i > 0$ receives its very first copy of this token, statement (NEXT) is executed and it behaves just as a non-initiator process does in the Chandy-Lamport algorithm, namely it invokes $CL(j)$ with *initiated* = `false`, and sends a copy of the *token* to each neighboring process. (For $i = 0$, these actions were previously done at σ.) When process $i \geq 0$ receives subsequent copies of this *token*, statement (RECSNAP) is executed ((SELFSNAP) for $i = 0$), causing process i to invoke $CL(j)$ with *initiated* = `true`, which is again exactly the way that a process behaves when receiving *markers* in the Chandy-Lamport algorithm.

Next, consider how copies of any *other token* are treated. At every process, predicates **IsMarker** and **IsNext** are false for each *token* created before σ, by the goodness of σ and the monotonicity of values created by process 0. The predicates are also false for each *token* created between σ and γ, since the FIFO channel assumption guarantees that $eoc[j]$ holds when received by i from each j and since no value greater than v is created before γ. Thus, the only possible effect of these *tokens* is to trigger execution of statement (RECPROD) or (SELFPROD). But since creation of *report* messages in this statement are guarded by predicate **Finished**, any such message is created by i *after* the *report* created by executing statement (RECSNAP) ((SELFSNAP) for process 0) and therefore, by the FIFO channel assumption (and the assumption that all *reports* from i are sent on identical routes) arrives at process 0 when $reported[i]$ is already true. Therefore, only copies of the *token* created at σ may affect the snapshot for σ. ∎

8 The reset algorithm

By prior assumption, along with the text of **basic** program **P**, we are given a predicate that recognizes representations of the legal global states of **P**. Once process 0 has obtained each snapshot, it applies this predicate. Should an illegal state be indicated, process 0 initiates execution of a *reset* algorithm, which changes the global state of **P** to some default normal initial state α.

The reset algorithm may be imposed onto Algorithm Snapshot very easily. Process 0 simply associates "flavor" with the VAL field of each *token* and *report* message. To begin a normal snapshot, "vanilla" *token*s are created; to begin a reset, "reset" tokens are created. Upon receiving a "reset" token, in addition to the actions described in Algorithm Snapshot, each process suspends execution of **P** (discarding any **basic** messages that subsequently arrive) and changes its local part of **P**'s state to the values it would have in α. Vanilla tokens are handled exactly as in Algorithm Snapshot, with the addition that receiving such a token causes a process to resume execution of **P** if it were suspended.

Because flavoring the tokens only modifies the state of **P**, it is trivial to show that our previous proof, that accurate snapshots are eventually repeatedly obtained, still holds. To see that executing the reset algorithm in a good state really causes **P** to resume execution in the default normal initial state α, consider how each process $i \geq 0$ behaves when it receives a "reset" token. By the FIFO property of channels and the fact that i suspends execution of **P** before sending a "reset" token, it is easy to show that, when process i's $eoc[j]$ variable becomes true the channel from process j to process i is devoid of **basic** messages and remains that way until the reset iteration ends. Thus, when process 0 begins the next vanilla iteration, **P** resumes execution from a normal initial state. Basic program **P** thenceforth remains in legal states and no further invocations of the reset algorithm may occur.

9 Improvements to Algorithm Snapshot

Note that the snapshots do not interfere in any way with the **basic** computation: it can always proceed, except during a reset phase. Although above the correctness has been emphasized, some simple techniques can be used to drastically reduce the expected number of steps to self-stabilization, under reasonable assumptions.

Gouda and Evangelist have defined the *convergence span*[GE88] of a self-stabilizing system to be the maximum number of "critical" steps that must be executed before a legal state is reached. For Algorithm Snapshot, consider statement (START) to be a critical step. The set V, previously used in the proof of Lemma 3, is defined by $v \in V$ if, in the initial state, v is the value of the *Current* variable of some process or the VAL of some message. Because statement (START) increments process 0's *Current* variable by 1, the convergence span of Algorithm Snapshot may be as much as the difference between the initial value of process 0's *Current* variable and the maximum of V.

A simple trick to reduce the convergence span is to have each process insert its *Current* value in each *report* message that it creates. When executing statement (START), process 0 need only choose a value that exceeds the maximum value received in the last set of *report* messages. If by chance there are no initial *token* or *report* messages (only arbitrary *Current* values), the modified version will converge on the second iteration to an accurate snapshot, followed by an accurate reset, if necessary. In the general case, define the set W as the subset of V with values greater than or equal to the initial value of process 0's *Current* variable. The modified version will have a convergence span of at most the cardinality of W.

Due to the completely asynchronous model, and the need for prod messages to avoid deadlock, under the assumptions seen here there is no way to bound the number of messages used for snapshots. The initiating process could generate arbitrarily many prodding messages before any other process completes its snapshot. A more realistic model would use absolute time intervals and time-outs to control the frequency of snapshots and prodding.

There is also a possible trade-off on the frequency of snapshot operations, relative to operations of the **basic** computation. If snapshots are

taken frequently, the system will self-stabilize when necessary, without wasting significant **basic** computation time. In this case, the number of steps needed afterwards to keep taking snapshots is large relative to the **basic** computation. On the other hand, if snapshots are only taken after some large number of **basic** computation steps, the self-stabilization will be slower, but the subsequent cost will be lower. Of course, this assumes that the **basic** computation cannot deadlock, even from illegal initial states.

10 Discussion

In this paper we have explored the limits of self-stabilization: we have shown the impossibility of attaining certain properties in self-stabilizing extensions and demonstrated the possibility of mechanically creating self-stabilizing extensions.

We have shown that it is impossible for certain properties to hold in self-stabilizing extensions. However, we note that the impossibility is sensitive to the exact statement of the property.

We have also demonstrated a superimposition that takes a **basic** program **P** and creates a self-stabilizing extension of **P**. An alternate means of creating this extension is to create a new program from scratch. The advantage of the superimposition is its generality: its correctness does not in any way depend on the behavior of **P** and it may therefore be imposed in an automated fashion. But this is also its weakness: by starting anew, we may take advantage of properties specific to **P** and create an extension that is perhaps more efficient and elegant. One should view the superimposition as demonstrating the possibility of a universal methodology for creating self-stabilizing extensions. The existence of other universal methodologies should be pursued. For example, our superimposition imposed Algorithm Snapshot on top of **P**. But one may look at Algorithm Snapshot itself as being the superimposition of a "control" algorithm on top of Chandy and Lamport's snapshot algorithm. A key feature of the "control" is that it invoked its "basic" program (i.e., the Chandy and Lamport algorithm) only when doing so would not violate the invariants of the "basic" program (e.g., exactly one *marker* received per channel per iteration). Thus, in some sense, it acted as a filter. Perhaps this technique

can be extended to a wider class of programs.

Some of the other techniques used may also be generally applicable, in particular, prodding and message numbering. The prodding ensured that the system would not deadlock, while the numbering was used to distinguish new rounds of activity from old ones. In addition, the existence of values larger than those initially in the system was crucial to the proof of eventual self-stabilization.

References

[BF88] L. Bouge and N. Francez. A compositional approach to superimposition. In *Principles of Programming Languages*, ACM, 1988.

[BGW89] G. M. Brown, M. G. Gouda, and C. L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 1989. to appear.

[BP89] J. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

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

[GE88] M. Gouda and M. Evangelist. *Convergence/Response tradeoffs in concurrent systems.* Technical Report TR88-39, University of Texas at Austin, 1988.

[Kat87] S. Katz. *A superimposition control construct for distributed systems.* Technical Report STP-268-87, MCC, 1987.

[Lam84] L. Lamport. The mutual exclusion problem: part ii - statement and solutions. *Journal of the ACM*, 33(2):327–348, 1984.

[LF82] L. Lamport and M. Fischer. *Byzantine Generals and Transaction Commit Protocols.* Technical Report Opus 62, SRI, 1982.

[LSP82] L. Lamport, R.E. SHostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

UNITY Proofs of Two Self-Stabilizing Termination Detection Algorithms

Charles Richter
MCC Software Technology Program
Austin, Texas

Abstract

A self-stabilizing system is a system that, when in a "bad" state, will eventually place itself in a "good" state, and once in a good state, will not return to a bad state. In an earlier paper, we give UNITY proofs of Dijkstra's original three self-stabilizing systems. In this paper, we present UNITY proofs of the self-stability of the termination detection algorithms of Gouda and Evangelist. We assume the reader is familiar with UNITY.

1.0 Self Stabilization

The self-stabilization problem, for a program F, may be stated as:

F *stabilizes-to* g ≡
stable g in F ∧
true ↦ g in F

where g is some state formula of F. Thus, F stabilizes to a set of states g if g is stable in F — that is, once in g, F remains in g — and if each state leads to g in F. The two properties, *stable* g and true ↦ g, are the safety and progress properties, respectively, of self-stability.

Gouda and Evangelist described a family of self-stabilizing termination detection algorithms, and gave examples of two such algorithms [3]. These algorithms assume a logical ring of N processes, P.0 through P.N-1. In each algorithm, P.i may examine only its own state and that of its left neighbor, P.i-1 (mod N). In other words, in each of the systems, any process P.i must alter its own state based only on that state and the state of its left neighbor.

We give the two algorithms given in [3] and proofs of their self-stability using UNITY [1]. For the UNITY formulations, we ignore the notion of process altogether, dealing simply with the changes to vectors of state variables. Note that we subscript the values of state variables using a "dot notation," with a blank terminating the subscript. For example, if s is a vector, then s.i refers to the i-th element of the vector, s.i+1 is the (i+1)'st element of s, and s.i + 1 is the result of adding 1 to the value of s.i. Also, where appropriate, modulo arithmetic is assumed.

2.0 The Slow Convergence, Fast Response System

In the first algorithm, GE1, each process has two states, s and v. The values for s are *idle* and *busy*, while v ranges from 0 to N-1. The program for GE1 is:

Note: This paper was also released as MCC technical report STP-230-89.

Program GE1

assign

s.0, v.0 := idle, (v.0 + 1) *mod* N	*if* s.0 = busy ∧ v.0 = v.N–1
[] s.0 := busy	*if* s.0 = idle ∧ s.N–1 = busy
[] s.0 := busy	*if* s.0 = idle ∧ v.0 = v.N–1
[] < [] i: 0<i≤N–1:: s.i, v.i := idle, v.i–1	*if* s.i = busy >
[] < [] i: 0<i≤N–1:: s.i := busy	*if* s.i = idle ∧ s.i–1 = busy >
[] < [] i: 0<i≤N–1:: s.i, v.i := idle, v.i–1	*if* s.i = idle ∧ v.i ≠ v.i–1 >

Intuitively, the propagation of a value from v.0 through v.N–1 represents the circulation of a token around the ring of processes. The token is used to detect termination: if v.0 = v.N–1, the token has traveled once around the ring, and if v.N–1 becomes equal to v.0 while s.0 = idle, then termination has occurred (i.e., all s.i = idle and v.i = v.0, for 0≤i≤N–1), and a new job may be initiated. The first transition in GE1 initiates token circulation, while the third transition detects termination and initiates a new job.

If we allow GE1 to start in a random state, however, the above may not be true. For example, we could have v.N–1 = v.0 and s.0 = idle (indicating termination has occurred), yet s.i = busy for some i, 0<i<N–1 (indicating that termination has not occurred). Therefore, we will partition the system into "good states" and "bad states;" a good state in any state in GE1 from which, if GE1 reaches a state where v.0 = v.N–1 and s.0 = idle, then all s.i = idle and v.i = v.0, for 0≤i≤N–1. Put another way, if GE1, once in a good state, subsequently detects termination, then termination has actually occurred.

We now define the set of good states for GE1. They are:

$$g \equiv \langle \exists m: 0 \le m < N:: g.m \rangle,$$

where:

$$g.m \equiv (s.0 = idle \wedge \langle \forall i: 0 \le i \le m:: s.i = idle \wedge v.i = v.0 \rangle \wedge \langle \forall i: m < i < N:: v.i = (v.0 - 1) \bmod N \rangle) \vee (s.0 = busy \wedge \langle \forall i: 0 \le i \le m:: v.i = v.0 \rangle \wedge \langle \forall i: m < i < N:: v.i = (v.0 - 1) \bmod N \rangle),$$

for 0≤m<N.

It is obvious that, if GE1 is in state g while v.0 = v.N–1 and s.0 = busy, then termination has actually occurred. It remains to prove that GE1 *stabilizes–to* g; that is:

stable g in GE1 , shown in section 2.2
∧ true ↦ g in GE1 , shown in section 2.3.

2.1 Properties of GE1

For notational convenience, we employ the following abbreviations in the proofs of GE1. Define q.i, for 0≤i≤N–1, to hold iff the values of all v.0 through v.i are equal. Define p.i, for 0≤i≤N–1, to hold iff q.i holds and the values of all v.j for j>i differ from the value of v.i. That is:

$$q.i \equiv \langle \forall j: 0 \le j \le i:: v.0 = v.j \rangle$$
$$p.i \equiv q.i \wedge \langle \forall j: i < j \le N-1:: v.0 \ne v.j \rangle$$

We first list some properties which can be proved directly from the program code for GE1. We will use the numbered properties in later proofs. These properties are:

g.N–1 *unless* g.0 (GE1.1)
g.i *unless* g.i+1 , for 0≤i≤N–2 (GE1.2)

$v.i \neq k$ *unless* $v.i \neq k \wedge v.i-1 = k$, for $0<i<N$ (GE1.3)

$v.0 = k$ *unless* $v.0 = k \wedge v.N-1 = k$ (GE1.4)

$v.0 = k$ *unless* $v.0 = k+1$ (GE1.5)

$v.i = k$ *unless* $v.i-1 \neq k$, , for $0<i<N$ (GE1.6)

$v.i \neq k \wedge v.i-1 = k$ *unless* $v.i-1 \neq k \vee v.i = k$ (GE1.7)

$v.0 = k \wedge \langle \exists i: 0<i \leq N-1:: v.i = k \rangle \Rightarrow$
$\langle \exists k: 0 \leq k<N:: \langle \forall i: 0 \leq i<N:: v.i \neq k \rangle \rangle$ (GE1.8)

To see that GE1.1 holds, observe that if g.N–1 holds, then either the first or the third statement will be executed; if the first is executed, g.N–1 will still hold, while after the third is executed, g.0 will hold. GE1.2 holds because, if g.i–1 (for $0<i \leq N-1$) holds before the second or fifth statement, g.i–1 will hold after that statement, while if g.i–1 is true before the fourth or sixth statement, g.i will hold afterward. Note that GE1.8 follows from the fact that the number of values for v.i is N, the number of processes. We can now derive other properties from the above using the theorems and definitions of UNITY:

$v.i \neq k \wedge v.i-1 = k$ *ensures* $v.i = k \vee v.i-1 \neq k$, for $0<i<N$ (GE1.9)
, from GE1.7 and program text

$\langle \forall i: 0<i<N:: v.i \neq k \rangle$ *unless* $v.0 = k$ (GE1.10)
, from the conjunction of GE1.3 for all i, $0<i<N$

$v.0 = k \wedge v.N-1 \neq k \wedge \ldots \wedge v.i+2 \neq k$ *unless* $v.0 = k \wedge v.N-1 \neq k \wedge \ldots \wedge v.i+2 \neq k \wedge v.i+1 = k$ (GE1.11)
, from the conjunction of GE1.3 and GE1.4

$v.0 = v.1 = \ldots = v.i = k$ *unless* $v.0 \neq v.1 = \ldots = v.i = v.N-1 = k$ (GE1.12)
, from the conjunction of GE1.6 and GE1.4

$v.0 = v.1 = \ldots = v.i = k \wedge v.N-1 \neq k \wedge \ldots \wedge v.i+2 \neq k$ *unless* $v.0 = v.1 = \ldots = v.i = v.i+1 = k \wedge v.N-1 \neq k \wedge \ldots \wedge v.i+2 \neq k$ (GE1.13)
, from the conjunction of the GE1.11 and GE1.12

$p.i \mapsto p.i+1$, for $0 \leq i \leq N-2$ (GE1.14)
, GE1.13, GE1.9 $(v.i+1 \neq k \wedge v.i = k \mapsto v.i+1 = k \vee v.i \neq k)$, and PSP

$v.0 = v.1 = \ldots = v.i = k$ *unless* $v.0 = k+1$ (GE1.15)
, from the conjunction of GE1.5, GE1.6

$v.i \neq k \wedge v.i-1 = k$ *ensures* $v.i-1 \neq k \vee v.i = k$ (GE1.16)
, from GE1.7 and program text

$v.0 = k \wedge q.i$ *ensures* $v.0 = k+1 \vee q.i+1$ (GE1.17)
, from PSP applied to GE1.15 and GE1.16

$\langle \forall i: 0 \leq i \leq N-1:: v.i = k \rangle$ *ensures* $v.0 = k+1 \wedge \langle \forall i: 1 \leq i \leq N-1:: v.i = k \rangle$ (GE1.18)
, from GE1.10 (v.0 will change first) and program text.

The actual derivations of properties GE1.10, GE1.11, and GE1.12 are shown below. The derivation of GE1.15 is similar to that of GE1.12. GE1.10 is derived from GE1.3 as follows:

$v.1 \neq k \wedge v.2 \neq k$ *unless* $(v.1 \neq k \wedge v.2 \neq k \wedge v.1 = k) \vee (v.2 \neq k \wedge v.1 \neq k \wedge v.0 = k) \vee (v.0 = k \wedge v.1 \neq k \wedge v.1 = k \wedge v.2 \neq k)$
, from the conjunction of GE1.3 and for i=1 and i=2

$v.1 \neq k \wedge v.2 \neq k$ *unless* $v.0 = k$
, removing false terms and consequence weakening

$v.1 \neq k \wedge v.2 \neq k \wedge v.3 \neq k$ *unless* $v.0 = k$
, from the conjunction of the above and GE1.3 for i=3, simplifying the right-hand side.

Repeated applications of the conjunction rule with GE1.3 will generate property GE1.10. The derivation of GE1.11 is:

$v.0 = k \wedge v.N-1 \neq k$ *unless* $v.0 = k \wedge v.N-1 \neq k \wedge v.N-2 = k$
, from conjunction of GE1.4 and GE1.3 for i=N–1, omitting false terms

$v.0 = k \wedge v.N-1 \neq k \wedge v.N-2 \neq k$ *unless* $v.0 = k \wedge v.N-1 \neq k \wedge v.N-2 \neq k \wedge v.N-3 = k$
, from conjunction of the above and GE1.3 for i=N–2
$v.0 = k \wedge v.N-1 \neq k \wedge v.N-2 \neq k \wedge v.N-3 \neq k$ *unless*
$v.0 = k \wedge v.N-1 \neq k \wedge v.N-2 \neq k \wedge v.N-3 \neq k \wedge v.N-4 = k$
, from conjunction of the above and GE1.3 for i=N–3.

Repeated applications of the conjunction rule using GE1.3 will generate property GE1.11. The derivation of GE1.12 is:

$v.1 = k$ *unless* $v.0 \neq k$
, from property GE1.6
$v.2 = k$ *unless* $v.1 \neq k$
, from property GE1.6
$v.1 = k \wedge v.2 = k$ *unless* $v.0 \neq k$
, from the conjunction of the above two, plus consequence weakening
$v.3 = k$ *unless* $v.2 \neq k$
, from property GE1.6
$v.1 = k \wedge v.2 = k \wedge v.3 = k$ *unless* $v.0 \neq k$
, from the conjunction of the above two, plus consequence weakening
$v.1 = \ldots = v.i = k$ *unless* $v.0 \neq k$
, from the conjunction of GE1.6 for 1 through i
$v.0 = k$ *unless* $v.0 = k \wedge v.N-1 = k$
, property GE1.4
$v.0 = v.1 = \ldots = v.i = k$ *unless* $v.0 = v.1 = \ldots = v.i = v.N-1 = k$
, from the conjunction of the above two

2.2 Safety Proof of GE1

We now prove the essential safety property of GE1, that g is stable. The proof that *stable* g in GE1 is:

g.i *unless* g.i+1 , for $0 \leq i \leq N-1$ (assuming addition modulo N)
, from the simple disjunction of GE1.1 and GE1.2
$\langle \exists i: 0 \leq i \leq N-1:: g.i \rangle$ *unless* $\langle \forall i: 0 \leq i \leq N-1:: \neg g.i \vee g.i+1 \rangle \wedge \langle \exists i: 0 \leq i \leq N-1:: g.i \rangle$
, from general disjunction for *unless* applied to the above
$\langle \exists i: 0 \leq i \leq N-1:: g.i \rangle$ *unless* false
, from the definition of g
g *unless* false
, because $\langle \exists i: 0 \leq i \leq N-1:: g.i \rangle \equiv g$, from the definition of g.

Hence, *stable* g in GE1. Note that $\langle \forall i: 0 \leq i \leq N-1:: \neg g.i \vee g.i+1 \rangle \wedge \langle \exists i: 0 \leq i \leq N-1:: g.i \rangle$ must be false, because g.i can hold for at most one i, $0 \leq i \leq N-1$. $\langle \exists i: 0 \leq i \leq N-1:: g.i \rangle$ states that g.i holds for at least one i, and $\langle \forall i: 0 \leq i \leq N-1:: \neg g.i \vee g.i+1 \rangle$ states that if g.i holds, then g.i+1 also holds.

2.2 Progress Proof of GE1

The progress proof obligation for GE1 is to show that true $\mapsto$ g. The proof that true $\mapsto$ g in GE1 is:

true $\mapsto$ q.N–1 $\vee$ g
, proved below
q.N–1 $\mapsto$ g
, proved below
true $\mapsto$ g
, from cancellation applied to the above two properties.

The first assertion merely states that, from any state, GE1 will reach a state at which either all the v.i values, for i between 0 and N–1, will be the same, or else g holds. The second property asserts that, once GE1 has reached the state just described, it will eventually reach a state at which g holds.

The proof of the second intermediate result, $q.N-1 \mapsto g$, is:

$q.N-1 \equiv \langle \forall i: 0 \leq i \leq N-1:: v.i = k \rangle$
, from some k, by the definition of q
$\langle \forall i: 0 \leq i \leq N-1:: v.i = k \rangle \mapsto v.0 = k+1 \wedge \langle \forall i: 1 \leq i \leq N-1:: v.i = k \rangle$
, from GE1.18
$v.0 = k+1 \wedge \langle \forall i: 1 \leq i \leq N-1:: v.i = k \rangle \Rightarrow g$
, from the definition of g
$q.N-1 \mapsto g$
, from the above, the implication rule, and the transitivity of $\mapsto$.

The proof of the first intermediate result, $true \mapsto q.N-1 \vee g$, is:

$true \Rightarrow v.0 = k \wedge (\langle \forall i: 0 < i \leq N-1:: v.i \neq k \rangle \vee \langle \exists i: 0 < i \leq N-1:: v.i = k \rangle)$
, for some k, $0 \leq k \leq N-1$
$true \mapsto v.0 = k \wedge (\langle \forall i: 0 < i \leq N-1:: v.i \neq k \rangle \vee \langle \exists i: 0 < i \leq N-1:: v.i = k \rangle)$
, from the implication rule for $\mapsto$
$v.0 = k \wedge \langle \forall i: 0 < i \leq N-1:: v.i \neq k \rangle \mapsto q.N-1 \vee g$
, shown below
$v.0 = k \wedge \langle \exists i: 0 < i \leq N-1:: v.i = k \rangle \mapsto q.N-1 \vee g$
, shown below
$true \mapsto q.N-1 \vee g$
, from the cancellation rule for $\mapsto$.

We now have two new intermediate results to show. The proof that the first, $v.0 = k \wedge \langle \forall i: 0 < i \leq N-1:: v.i \neq k \rangle \mapsto q.N-1 \vee g$, is:

$v.0 = k \wedge \langle \forall i: 0 < i \leq N-1:: v.i \neq k \rangle \mapsto \langle \forall i: 0 < i \leq N-1:: v.i \neq k \rangle$
, from the implication rule for $\mapsto$
$\langle \forall i: 0 < i \leq N-1:: v.i \neq v.0 \rangle \mapsto q.N-1 \vee g$
, shown below
$v.0 = k \wedge \langle \forall i: 0 < i \leq N-1:: v.i \neq k \rangle \mapsto q.N-1 \vee g$
, from the transitivity of $\mapsto$.

We now prove the second intermediate result from above, $v.0 = k \wedge \langle \exists i: 0 < i \leq N-1:: v.i = k \rangle \mapsto q.N-1 \vee g$. Note that, while proving this, we will also prove the intermediate result just introduced, $\langle \forall i: 0 < i \leq N-1:: v.i \neq v.0 \rangle \mapsto q.N-1 \vee g$. The proof that $v.0 = k \wedge \langle \exists i: 0 < i \leq N-1:: v.i = k \rangle \mapsto q.N-1 \vee g$ is:

$v.0 = k \wedge \langle \exists i: 0 < i \leq N-1:: v.i = k \rangle \mapsto \langle \forall i: 0 < i \leq N-1:: v.i \neq v.0 \rangle \vee q.N-1$
, shown below
$\langle \forall i: 0 < i \leq N-1:: v.i \neq v.0 \rangle \mapsto q.N-1 \vee g$
, shown below
$v.0 = k \wedge \langle \exists i: 0 < i \leq N-1:: v.i = k \rangle \mapsto q.N-1 \vee g$
, from the cancellation rule for $\mapsto$.

Again, we have new intermediate results to prove. We will first prove the second one, $\langle \forall i: 0 < i \leq N-1:: v.i \neq v.0 \rangle \mapsto q.N-1 \vee g$. The proof is by induction: the value of v.0 will be propagated first to v.1, then to v.2, and so on, while v.0 remains unchanged. Define M to be a vector of N bits, b.0 through b.N–1, where:

b.0 = 0, and
b.i = 1 iff $v.i \neq v.0$, for $0 < i < N$.

Let the orderings among values of M be lexicographic, but such that b.0 is the high-order bit. The values of M obviously form a well-founded set. The proof that $\langle\forall i: 0<i\leq N-1:: v.i \neq v.0\rangle \mapsto q.N-1 \vee g$ is:

$\langle\forall i: 0<i\leq N-1:: v.i \neq v.0\rangle \equiv p.0$
, from the definition of p
$p.i \mapsto p.i+1$, for $0\leq i<N-1$
, from GE1.14
$p.i \wedge M=m \mapsto p.i+1 \wedge M<m$
, from the definition of M
$\langle\exists i: 0\leq i\leq N-2:: p.i\rangle \wedge M=m \mapsto (\langle\exists i: 0\leq i\leq N-1:: p.i\rangle \wedge M<m) \vee g$
, rewriting the above
$M=0 \Rightarrow p.N-1 \Rightarrow q.N-1$
, from the definitions of M, p
$\langle\exists i: 0\leq i\leq N-2:: p.i\rangle \mapsto q.N-1 \vee g$
, from the induction rule for $\mapsto$
$\langle\forall i: 0<i\leq N-1:: v.i \neq v.0\rangle \mapsto q.N-1 \vee g$
, from the transitivity of $\mapsto$.

We now prove the first intermediate result above, that $v.0 = k \wedge \langle\exists i: 0<i\leq N-1:: v.i = k\rangle \mapsto \langle\forall i: 0<i\leq N-1:: v.i \neq v.0\rangle \vee q.N-1$. Again, we will use an inductive argument, employing the fact (shown below) that v.0 will eventually be incremented, and thus will eventually reach a value to which no other v.i's are equal. Define a metric D to be the difference between the current value of v.0 and the "next higher" possible value j such that $\langle\forall i: 0\leq i<N:: v.i \neq j\rangle$. (Property GE1.8 assures us of such a j, and GE1.10 guarantees $\langle\forall i: 0\leq i<N:: v.i \neq j\rangle$ until $v.0 = j$.) That is, D is defined as:

D = minimum $\{(j - v.0)$ *mod* N, for j such that $\langle\forall i: 0\leq i<N:: v.i \neq j\rangle\}$.

(There may be several such j; we want the smallest one greater, mod N, than v.0's value.) Proof that $v.0 = k \wedge \langle\exists i: 0<i\leq N-1:: v.i = k\rangle \mapsto \langle\forall i: 0<i\leq N-1:: v.i \neq v.0\rangle \vee q.N-1$:

$v.0 = k \mapsto v.0 = k+1 \vee q.N-1$
, shown below
$v.0 = k \wedge D=d \mapsto (v.0 = k+1 \wedge D<d) \vee q.N-1$
, from the definition of D
$\langle\forall k: 0\leq k<N:: v.0 = k\rangle \wedge D=d \mapsto (\langle\forall k: 0\leq k<N:: v.0 = k\rangle \wedge D<d) \vee q.N-1$
, rewriting the above
$D=0 \Rightarrow v.0 = j$
, from the definition of D
$\langle\forall k: 0\leq k<N:: v.0 = k\rangle \mapsto v.0 = j \vee q.N-1$
, from the induction rule for $\mapsto$
$\langle\forall i: 0\leq i<N:: v.i \neq j\rangle$ *unless* $v.0 = j \wedge \langle\forall i: 0<i<N:: v.i \neq j\rangle$
; property GE1.18
$\langle\forall k: 0\leq k<N:: v.0 = k\rangle \mapsto (v.0 = j \wedge \langle\forall i: 0<i<N:: v.i \neq j\rangle) \vee q.N-1$
, from PSP applied to the above two properties
$v.0 = k \wedge \langle\exists i: 0<i\leq N-1:: v.i = k\rangle \mapsto \langle\forall i: 0<i\leq N-1:: v.i \neq v.0\rangle \vee q.N-1$
, rewriting the above.

We must now show that $v.0 = k \mapsto v.0 = k+1 \vee q.N-1$. Once again, we must employ an inductive proof. This time, we use induction to show that the value k will eventually be propagated to every v.i, after which v.0 must change. (Of course, v.0 can change whenever $v.N-1 = v.0$; k need not be propagated to all v.i's.) Let M be as defined before, a vector of N bits, b.0 through b.N-1, where:

b.0 = 0, and
$b.i = 1$ iff $v.i \neq v.0$, for $0<i<N$.

Let $0 \le k < K$. Proof that $v.0 = k \mapsto v.0 = k+1 \vee q.N-1$:

$v.0 = k \wedge q.i \mapsto q.i+1 \vee v.0 = k+1$
, property GE1.17
$v.0 = k \wedge q.i \wedge M=m \mapsto (q.i+1 \wedge M<m) \vee v.0 = k+1$
, from the definition of M
$v.0 = k \wedge \langle \exists i: 0 \le i < N-1:: q.i \rangle \wedge M=m \mapsto$
$(v.0 = k \wedge \langle \exists i: 0 < i \le N-1:: q.i \rangle \wedge M<m) \vee v.0 = k+1$
, rewriting the above
$M=0 \Rightarrow q.N-1$
, from the definition of M
$v.0 = k \wedge \langle \exists i: 0 \le i < N-1:: q.i \rangle \mapsto (q.N-1 \wedge v.0 = k) \vee v.0 = k+1$
, from the induction rule for $\mapsto$
$v.0 = k \wedge \langle \exists i: 0 \le i < N-1:: q.i \rangle \mapsto q.N-1 \vee v.0 = k+1$
, from consequence weakening.

This completes the progress proof for GE1.

3.0 The Fast Convergence, Slow Response System

The second algorithm, GE2, is similar to GE1. GE2, however, includes an additional variable, z, indicating the number of times the token has circulated around the ring. Termination is not detected until the token has circulated through the ring N times. The program for GE2 is:

Program GE2
assign

s.0, v.0, z := idle, (v.0 + 1) *mod* 2, 0	*if* s.0 = busy $\wedge$ v.0 = v.N–1
[] v.0, z := (v.0 + 1) *mod* 2, z+1	*if* s.0 = idle $\wedge$ v.0 = v.N–1 $\wedge$ z < N–1
[] s.0 := busy	*if* s.0 = idle $\wedge$ s.N–1 = busy
[] s.0 := busy	*if* s.0 = idle $\wedge$ v.0 = v.N–1 $\wedge$ z = N–1
[] < [] i: 0<i≤N–1:: s.i, v.i := idle, v.i–1	*if* s.i = busy >
[] < [] i: 0<i≤N–1:: s.i := busy	*if* s.i = idle $\wedge$ s.i–1 = busy >
[] < [] i: 0<i≤N–1:: v.i := v.i–1	*if* s.i = idle $\wedge$ v.i $\neq$ v.i–1 >

The good states for GE2 are:

$g \equiv \langle \exists m: 0 \le m < N:: g.m \rangle$,

where:

$g.0 \equiv s.0 = busy \vee (s.0 = idle \wedge s.1 = busy \wedge [v.0 \neq v.N-1 \wedge K \le (N - z.0 - 1)])$

$g.m \equiv \langle \forall i: 0 \le i \le m:: s.i = idle \rangle \wedge s.m+1 = busy$
$\wedge ([v.0 = v.N-1 \wedge K < (N - z.0 - 1)] \vee [v.0 \neq v.N-1 \wedge K \le (N - z.0 - 1)])$
,for $0 < m < N-1$, and

$g.N-1 \equiv \langle \forall i: 0 \le i \le N-1:: s.i = idle \rangle$,

and where K is the number of k's such that $m < k < N-1$ and $v.k \neq v.k+1$.

3.1 Properties of GE2

For notational convenience, define p.i, for $0 \le i \le N-1$, to hold if and only if all s.0 through s.i are idle. That is,

$p.i \equiv \langle \forall j: 0 \le j \le i :: s.j = idle \rangle$, for $0 \le i \le N-1$.

The following properties of GE2 are obvious from program text:

$g.0$ *unless* $\langle \exists i: 0 < i \le N-1 :: g.i \rangle$ (GE2.1)

$g.N-1$ *unless* $g.0$ (GE2.2)

$g.i$ *unless* $g.0 \vee \langle \exists j: i < j \le N-1 :: g.j \rangle$, for $1 \le i \le N-2$ (GE2.3)

$s.0 = idle$ *unless* $s.0 = busy$ (GE2.4)

$s.i = idle$ *unless* $s.i-1 \neq idle$, for $1 \le i \le N-1$ (GE2.5)

GE2.1 holds because g.0 must hold before the first statement is executed (because s.0 = busy); after it is executed, g.i, for some i, $0 \le i \le N-1$, will hold. Property GE2.2 holds because g.N–1 holds and therefore all s.i's are idle, and so the only change which could invalidate g.N–1 is if s.0 becomes busy. GE2.3 holds because, if g.i holds (for $1 \le i \le N-2$), then after the third or fourth statement is executed, g.0 will hold, while after the fifth statement is executed, g.j will hold for $j \ge i$, and after the second, sixth, or seventh statement is executed, g.i will still hold.

The following properties can be derived from the above:

$p.i$ *unless* $p.i+1 \vee s.0 = busy$ (GE2.6)
, from disjunction of GE2.4 and GE2.5

$p.i$ *ensures* $p.i+1 \vee s.0 = busy$ (GE2.7)
, from GE2.6 and program text.

Note that the actual derivation of GE2.6 is similar to that for GE1.15, given in section 2.1.

3.2 Safety Proof of GE2

We now prove that g is stable in GE2. Properties GE2.1, GE2.2, and GE2.3 tell us that if g.i holds for some i, $0 \le i \le N-1$, then it will continue to hold until g.j holds for some j, $0 \le j \le N-1$, and so, intuitively, g is stable. We can derive *stable* g in GE2 formally from GE2.1, GE2.2, and GE2.3 as follows:

$g.0$ *unless* $\langle \exists i: i \neq 0 :: g.i \rangle$
, property GE2.1

$g.N-1$ *unless* $\langle \exists i: i \neq N-1 :: g.i \rangle$
, from consequence weakening and GE2.2

$g.i$ *unless* $\langle \exists j: j \neq i :: g.j \rangle$,for $1 \le i \le N-2$
, from consequence weakening and GE2.3

$g.i$ *unless* $\langle \exists j: j \neq i :: g.j \rangle$,for $0 \le i \le N-1$
, from simple disjunction of the above

$\langle \exists i: 0 \le i \le N-1 :: g.i \rangle$ *unless*
$\langle \forall i: 0 \le i \le N-1 :: \neg g.i \vee \langle \exists j: 0 \le j \le N-1 \wedge j \neq i :: g.j \rangle \rangle \wedge \langle \exists i: 0 \le i \le N-1 :: g.i \rangle$
, from general disjunction for *unless* applied to the above

$\langle \exists i: 0 \le i \le N-1 :: g.i \rangle$ *unless* false
, from the definition of g

g *unless* false
, because $\langle \exists i: 0 \le i \le N-1 :: g.i \rangle \equiv g$, from the definition of g.

Therefore, g *unless* false in GE2; that is, *stable* g in GE2. Note that an explanation analogous to that at the end of section 2.2 shows why $\langle \forall i: 0 \le i \le N-1 :: \neg g.i \vee \langle \exists j: 0 \le j \le N-1 \wedge j \neq i :: g.j \rangle \rangle \wedge \langle \exists i: 0 \le i \le N-1 :: g.i \rangle$ is false.

3.3 Progress Proof of GE2

We now show that true $\mapsto$ g in GE2. We first sketch the proof informally; we then derive it formally. Either s.0 = busy or s.0 = idle. If s.0 = busy, g holds by definition. If s.0 =

idle, then either idle will be propagated to all s.i, or else s.0 will become idle. In either case, g holds.

The proof that true $\mapsto$ g in GE2 is:

true $\mapsto$ s.0 = busy $\vee$ s.0 = idle
, from the implication rule for $\mapsto$
s.0 = busy $\mapsto$ g
, from the implication rule for $\mapsto$
s.0 = idle $\mapsto$ g
, shown below
true $\mapsto$ g
, from the cancellation rule for $\mapsto$.

We now show the intermediate result, that s.0 = idle $\mapsto$ g. To do this, we use an inductive proof to show that (if g doesn't hold first) idle is propagated up through s.N–1. For the induction, we use metric function M whose value is defined to be a vector of bits b.0 through b.N–1, where each bit is defined as follows:

b.i = 1 iff s.i = busy , for $0 \leq i \leq N-1$.

Let the orderings among values of M be lexicographic, but such that b.0 is the high–order bit. (That is, for $m_1, m_2 \in M$, $m_1 > m_2$ if b.0 = 1 in m_1 while b.0 = 0 in m_2.) The values of M obviously form a well–founded set. The proof that s.0 = idle $\mapsto$ g is:

s.0 = idle $\Rightarrow$ $\langle \exists i: 0 \leq i \leq N-1:: p.i \rangle$
, from the definition of p
s.0 = idle $\mapsto$ $\langle \exists i: 0 \leq i \leq N-1:: p.i \rangle$
, from the implication rule for $\mapsto$
$\langle \exists i: 0 \leq i \leq N-1:: p.i \rangle \wedge M=m \mapsto (\langle \exists i: 0 \leq i \leq N-1:: p.i \rangle \wedge M<m) \vee g$
, from GE2.7, the definition of M, and $M=0 \Rightarrow p.N-1 \Rightarrow g$
$\langle \exists i: 0 \leq i \leq N-1:: p.i \rangle \mapsto g$
, from the induction rule for $\mapsto$
s.0 = idle $\mapsto$ g
, from the transitivity of $\mapsto$.

This completes the progress proof for GE2.

Remarks

When they presented these two termination detection algorithms, Gouda and Evangelist identified the rate at which each algorithm converges to a safe state and the kinds of transitions that must be executed during that convergence [3]. We ignore those aspects here, concentrating instead on the proofs of self–stability.

In related work, the self–stability of Dijkstra's original three systems [2] are proved using UNITY in [4]. An interesting proof (again using UNITY) of the self–stability of Hopfield nets is given in [5].

References

[1] M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison–Wesley, 1988.

[2] E. Dijkstra, "Self–Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM*, Volume 17, Number 11, November 1974, pp. 643–4.

[3] M. Gouda and M. Evangelist, "Convergence/Response Tradeoffs in Concurrent Systems," MCC Technical Report STP-124-89, submitted to *Transactions on Programming Languages and Systems*.

[4] C. Richter, "Proofs of Some Self-Stabilizing Systems," MCC Technical Report STP-212-89.

[5] N. Soparkar, "Hopfield Nets — More Formally," Department of Computer Sciences Technical Report TR-88-38, University of Texas at Austin, October 1988.

Appendix A: Alternative Safety Proofs for GE1 and GE2

In the safety proofs for GE1 and GE2, we used the general disjunction rule for *unless*. This rule was discovered after the publication of [1] and so is not included in that book. Here we give safety proofs using only rules included in [1].

The essential safety property of GE1, that *stable* g in GE1, is proved using only properties GE1.1 and GE 1.2. One straightforward derivation of g *unless* false is:

g.0 $\vee$ g.1 *unless* g.2
, from the disjunction of g.0 *unless* g.1 and g.1 *unless* g.2
g.0 $\vee$ g.1 $\vee$ g.2 *unless* g.3
, from the disjunction of the above and g.2 *unless* g.3
g.0 $\vee$ g.1 $\vee$ ... $\vee$ g.N–2 *unless* g.N–1
, from the disjunction of all g.i *unless* g.i+1, for $0 \leq i \leq N-2$
g.0 $\vee$ g.1 $\vee$ ... $\vee$ g.N–2 $\vee$ g.N–1 *unless* false
, from the disjunction of the above and g.N–1 *unless* g.0
g *unless* false
, from the above and g $\equiv$ g.0 $\vee$ g.1 $\vee$ ... $\vee$ g.N–2 $\vee$ g.N–1.

Proof that *stable* g in GE2:

g.N–2 *unless* g.0 $\vee$ g.N–2 $\vee$ g.N–1
, from GE2.3
g.N–2 $\wedge$ $\neg$(g.0 $\vee$ g.N–1) *unless* g.0 $\vee$ g.N–2 $\vee$ g.N–1
, from the above
g.N–2 *unless* $\neg$g.N–2
, from the anti–reflexivity rule for *unless*
g.N–2 *unless* g.0 $\vee$ g.N–1
, from disjunction of the above two properties
g.N–1 *unless* $\neg$g.N–1 $\wedge$ g.0
, from consequence weakening of GE2.2
g.N–1 $\vee$ g.N–2 *unless* g.0
, from simple disjunction of the above two
g.N–3 *unless* g.0 $\vee$ g.N–1 $\vee$ g.N–2
, from property GE2.3
g.N–1 $\vee$ g.N–2 *unless* $\neg$(g.N–1 $\vee$ g.N–2) $\wedge$ g.0
, from consequence weakening of an above property
g.N–1 $\vee$ g.N–2 $\vee$ g.N–3 *unless* g.0
; from simple disjunction of the above two
g.N–1 $\vee$ g.N–2 $\vee$ ... $\vee$ g.1 *unless* g.0
, from repeated applications of the above rules
g.0 $\wedge$ $\neg$(g.1 $\vee$ ... $\vee$ g.N–1) *unless* g.0 $\vee$ g.1 $\vee$ ... $\vee$ g.N–1
, from property GE2.1
g.0 *unless* $\neg$g.0
, from the anti–reflexivity rule for *unless*
g.0 *unless* g.1 $\vee$ g.2 $\vee$ ... $\vee$ g.N–1
, from disjunction of the above two
g.N–1 $\vee$ g.N–2 $\vee$ ... $\vee$ g.1 $\vee$ g.0 *unless* false
, from disjunction of above and g.N–1 $\vee$ g.N–2 $\vee$ ... $\vee$ g.1 *unless* g.0
g *unless* false
, because g $\equiv$ g.N–1 $\vee$ g.N–2 $\vee$ ... $\vee$ g.1 $\vee$ g.0.

Appendix B: UNITY Definitions and Theorems

We now list the UNITY definitions and theorems used in this paper. These concepts are explained in more detail in [1]. The relevant definitions, where F is a program and s is a program statement, are:

p *unless* q in F $\equiv$ $\langle \forall s:\ s \text{ in } F ::\ \{ p \wedge \neg q \}\ \ s\ \ \{ p \vee q \} \rangle$

p *ensures* q in F $\equiv$ $(p \text{ } \textit{unless}\ q \wedge \langle \exists s:\ s \text{ in } F ::\ \{ p \wedge \neg q \}\ \ s\ \ \{ q \} \rangle)$

$p \mapsto q$ holds in F iff $p \mapsto q$ can be derived by a finite number of applications of the following rules:

$$\frac{p \text{ } \textit{ensures}\ q}{p \mapsto q}$$

$$\frac{p \mapsto r,\ \ r \mapsto q}{p \mapsto q}$$ (the transitivity rule)

$$\frac{\langle \forall m:\ m \in W ::\ p(m) \mapsto q \rangle}{\langle \exists m:\ m \in W ::\ p(m) \rangle \mapsto q}$$ (the disjunction rule, for any set W)

The theorems for *unless* are:

anti–reflexivity:

$p \text{ } \textit{unless}\ \neg p$

consequence weakening:

$$\frac{p \text{ } \textit{unless}\ q,\ \ q \Rightarrow r}{p \text{ } \textit{unless}\ r}$$

conjunction:

$$\frac{p_1 \text{ } \textit{unless}\ q_1,\ \ p_2 \text{ } \textit{unless}\ q_2}{p_1 \wedge p_2 \text{ } \textit{unless}\ (p_1 \wedge q_2) \vee (p_2 \wedge q_1) \vee (q_1 \wedge q_2)}$$

disjunction:

$$\frac{p_1 \text{ } \textit{unless}\ q_1,\ \ p_2 \text{ } \textit{unless}\ q_2}{p_1 \vee p_2 \text{ } \textit{unless}\ (\neg p_1 \wedge q_2) \vee (\neg p_2 \wedge q_1) \vee (q_1 \wedge q_2)}$$

simple disjunction:

$$\frac{p_1 \text{ } \textit{unless}\ q_1,\ \ p_2 \text{ } \textit{unless}\ q_2}{p_1 \vee p_2 \text{ } \textit{unless}\ q_1 \vee q_2}$$

general disjunction:

$$\frac{\langle \forall i ::\ p.i \text{ } \textit{unless}\ q.i \rangle}{\langle \exists i :: p.i \rangle \text{ } \textit{unless}\ \langle \forall i ::\ \neg p.i \vee q.i \rangle \wedge \langle \exists i ::\ q.i \rangle}$$, for sets of predicates p, q

The theorems for $\mapsto$ (aside from transitivity, given above) are:

implication:

$$\frac{p \Rightarrow q}{p \mapsto q}$$

cancellation:

$$\frac{p \mapsto q \vee b,\ \ b \mapsto r}{p \mapsto q \vee r}$$

consequence weakening:

$$\frac{p \mapsto q,\ \ q \Rightarrow r}{p \mapsto r}$$

induction:

$$\frac{\langle \forall m:\ m \in M ::\ p \wedge M{=}m \mapsto (p \wedge M \prec m) \vee q \rangle}{p \mapsto q}$$, for any well-founded set W

PSP (Progress-Safety-Progress):

$$\frac{p \mapsto q,\ \ r \ \textit{unless}\ b}{p \wedge r \mapsto (q \wedge r) \vee b}$$

On Relaxing Interleaving Assumptions

James E. Burns
School of Information and Computer Science
Georgia Institute of Technology

Mohamed G. Gouda
Department of Computer Sciences
The University of Texas at Austin

Raymond E. Miller
Department of Computer Science
University of Maryland

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 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.

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:

$$V_i \leftarrow V_{i-1 \,(\mathrm{mod}\ 3)} + V_i + V_{i+1 \,(\mathrm{mod}\ 3)}$$

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

$$\textbf{if } loc_i = 0 \textbf{ then } L_i \leftarrow V_{i-1 \,(\mathrm{mod}\ 3)} + V_{i+1 \,(\mathrm{mod}\ 3)};\ loc_i \leftarrow 1 \textbf{ fi}$$

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

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 [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 [Lam84] 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 concurrent 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 the V_j here stand (ambiguously) for the set of values taken on by the variables. 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 $\alpha = \{i\}$, 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, $0 < i < \mathit{length}(C)$.[1] 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 $\mathit{Own}_0, \mathit{Own}_1, \ldots, \mathit{Own}_n$ such that if $\gamma \xrightarrow{i} \gamma'$ and $V_j(\gamma) \neq V_j(\gamma')$ then $V_j \in \mathit{Own}_i$. Let $\mathit{Own}_i[\gamma]$ 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 $\mathit{Own}_i[\gamma'] = \mathit{Own}_i[\gamma_i]$, and for every $j \in V - \cup_{i \in \alpha} \mathit{Own}_i$, $V_j(\gamma') = V_j(\gamma)$.

(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

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

unambiguous because there are no multi-writer variables.

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 $parallel(\gamma_0)$ to be the subset of Γ that is reachable from γ_0. We define $serial(\gamma_0)$ analogously for serial reachability.

Reachability is often called a *safety* property. Another kind of property of interest is a *progress* 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 $eventually(\Lambda, \gamma_0)$ holds. The analogous predicate for serial progress is $serial\text{-}eventually(\Lambda, \gamma_0)$.

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(γ_0) = parallel(γ_0) and serial-eventually(Λ, γ_0) if and only if eventually(Λ, γ_0).*

Proof: Since serial computations are also parallel computations, trivially *serial*$(\gamma_0) \subseteq$ *parallel*(γ_0) and *eventually*(Λ, γ_0) implies *serial-eventually*(Λ, γ_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*(γ_0).

Now suppose *eventually*(Λ, γ_0) does not hold, so there is an infinite computation, C, from γ_0 that avoids Λ. Let C' be the corresponding serial computation, formed (inductively) by chaining together serial computations between the adjacent configurations in C. Since Λ is serially closed, if C' ever reaches a configuration of Λ, then all remaining configurations of C' are in Λ, which implies the next occurring configuration of C is in Λ. Therefore, (not *eventually*(Λ, γ_0)) implies (not *serial-eventually*(Λ, γ_0)), or, equivalently, *serial-eventually*$(\Lambda, \gamma_0) \Rightarrow$ *eventually*(Λ, γ_0), as required. □

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. 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 $D_\alpha(\gamma)$.

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.

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(γ_0) is acyclic, then serial(γ_0) = parallel(γ_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 infinite 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 [BGW, 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[2]. It seems likely that this property would have been observed before if the results of the previous section were known.

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 enabled 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

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

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}.) The 'mod' operator below is a binary operator returning the remainder of the first operand when divided by the second.

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.

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, $V_0(\gamma_1) = s + 1 \bmod K$, $V_0(\gamma_2) = s + 2 \bmod K$, ..., $V_0(\gamma_n) = +n \bmod K$. 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

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

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

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

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

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

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

[Lam84] L. Lamport. Solved problems, unsolved problems, and non-problems in concurrency. Invited address at the 2nd Symp. on Principles of Distributed

Computing (Montreal, Aug. 1983), included in *Proc. of the Third Symp. on Principles of Distributed Computing* (Vancouver, BC, Aug. 1984), ACM, 1984, pp. 1–11.

[Lam86] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing I*, 2 (Dec. 1986), 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. Theor. 15* (1981) 47–66.

The Instability of Self-Stabilization*

Mohamed G. Gouda
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Rodney R. Howell
Dept. of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506

Louis E. Rosier
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

June 22, 1989

Abstract

We argue that the important property of self-stabilization is, in principle, unstable across system classes. In particular, we show that, for a wide variety of system classes, there is no simulation that "preserves" or "enforces" self-stabilization.

1 Introduction

A system is said to be self-stabilizing if starting from any configuration the system is guaranteed to reach a configuration that is reachable from the initial configuration. The motivation behind this concept is that if, due to some unpredictable error, the system were to reach an "unreachable" configuration, it would eventually correct itself, returning to some "reachable" configuration. Thus, self-stabilizing systems are in some sense more robust than those that are not self-stabilizing. The notion of self-stabilization has been utilized in the fields of mathematics and control theory for many years (see, e.g., [CK80, OWA89]). Consider, for example, the Newton-Raphson method for finding roots of functions. For many functions, the Newton-Raphson method is self-stabilizing; i.e., no matter what initial estimate is made for the root, eventually the iteration will converge to a root. The concept of self-stabilization was introduced to the field of computer science by Dijkstra in [Dij73], and subsequently has gained much attention in computer science research, particularly in the area of distributed computing; see, e.g., [Dij74, Lam86, BP89, Gou87, BYC88, BGW89, Mul89].

The main purpose of this paper is to demonstrate how the potential for self-stabilization changes radically when one class of systems is simulated by another. As an example, consider a result of Dijkstra's given in [Dij73]. He gave a problem that can be solved by a self-stabilizing asymmetric ring of processes, and showed that no such solution exists if the ring is required to be symmetric (see also [LR81]). This result can be interpreted in the following way: there is no simulation of an asymmetric ring by a symmetric ring that preserves self-stabilization. This interpretation suggests that the *simulation paradigm*, which is often used in analyzing and designing systems, may not be very robust when the issue of self-stabilization is involved. A

*This work was supported in part by U. S. Office of Naval Research Grant No. N00014-86-K-0763 and National Science Foundation Grant No. CCR-8711579.

simulation of a class A of systems by a class B of systems — the paradigm in action — is simply a function $f : A \rightarrow B$ such that for each system $M \in A$, the computations of $f(M)$ mimic in some well-defined manner the computations of M. An example of a simulation is the simulation of shared memory programs by CSP systems [Hoa78]. In order to examine the effect of self-stabilization on the simulation paradigm, we will demonstrate the presence or absence of various types of simulations on a wide variety of system classes. To make our negative results as strong as possible, our formal definition of simulation will be very weak. Thus, we are able to show that even for a very liberal notion of simulation (in particular, the simulations are not even required to be recursive) there are many cases in which simulations preserving or enforcing self-stabilization cannot exist. Furthermore, we do not even need to consider the effect of inputs to a system in order to achieve these negative results.

We will now discuss in some detail how the simulation paradigm is used as both an analysis and a design tool. We then discuss how self-stabilization can affect both these aspects. The simulation paradigm is often used in the analysis of systems to determine whether a particular property, such as self-stabilization, is present in a given system. A common scenario is that we have an algorithm or methodology for deciding the presence of the property in one class of systems, say systems of communicating finite-state machines (CFSMs), but do not yet have one for some similar class, such as Boolean CSP systems. In order to achieve such a methodology in a Boolean CSP system, we might simulate the CSP system by a system of CFSMs, analyze the CFSMs, and finally, conclude properties about the CSP system. More formally, let A and B be two system classes. The simulation paradigm is then applied to the analysis of systems in the following manner:

1. Find a simulation $f : A \rightarrow B$.
2. Given a machine $M \in A$, analyze $f(M)$.
3. Conclude properties about M.

In order for this procedure to work, the simulation clearly must preserve (the existence or absence of) the property being studied. This is usually not a problem because the standard simulations nearly always preserve most useful properties (e.g., deadlock freedom, reachability, liveness, fair nontermination, etc. [KM69, OL82, EL87, Car87, HRY88]). However, we will show that such is often not the case with respect to self-stabilization.

The simulation paradigm is also used as a tool for designing systems. In this case, we are interested in determining whether a particular property, such as self-stabilization, can be forced upon a given system. More generally, we would like to find a simulation of a class A by a class B such that all simulating systems have the desired property. For example, suppose we wish to design some self-stabilizing system in Ada to be implemented on a variety of different architectures. If there is some efficiently computable simulation of Ada programs that forces self-stabilization on each of the target architectures, we do not need to worry about self-stabilization; we simply design the system in Ada, and use the simulation to force self-stabilization on the target architecture. Note that in general the classes A and B do not need to be different classes, in which case we are simply interested in whether the property can be forced on any given system in A. Again, we will show that self-stabilization cannot be forced for many classes of systems.

Hence, we specifically examine two questions, one related to analysis and the other related to design, with respect to a number of classes of systems. Let A and B be two system classes. The two questions we

ask are:

1. Does there exist a simulation of A by B that preserves self-stabilization?
2. Does there exist a simulation of A by B that forces self-stabilization?

The classes of concurrent systems we consider include cellular arrays, communicating finite-state machines, CSP systems, and systems of Boolean programs communicating via 1 reader / 1 writer shared variables. In order to demonstrate that the difficulties are not simply products of concurrency, we also consider finite-state machines, Petri nets, Turing machines, and vector addition systems with states.

The main message here is that self-stabilization, as important a property as it is, is very sensitive to changes in the system classes under consideration. Understanding this sensitivity is the first step for a system designer who seeks self-stabilization in systems. It is also important to keep the results of this paper in perspective. In particular, the definition of simulation is intentionally chosen to be very weak to support the proofs of very strong negative results. Thus, even though a number of positive results are included for completeness, they are not entirely satisfactory. In fact, by taking full advantage of the weakness of our definition, one might show the existence of some type of simulation by exhibiting a nonrecursive simulation. Such a simulation would obviously be unusable in the simulation paradigm. Therefore, in any study focusing on positive results, a stronger definition of simulation should be considered.

The remainder of the paper is organized as follows. In Section 2, we define much of the terminology used throughout the paper. In Section 3, we give an overview of the specific problems we examine and summarize the main factors that tend to disrupt self-stabilization. In Section 4, we examine the role of halting on self-stabilization. In Sections 5 and 6, we examine two more subtle phenomena, isolation and look-alike configurations. Finally, in the Appendix we give detailed proofs of several theorems omitted from the body of the paper to improve readability.

2 Definitions

We include in this paper the examination of a wide variety of models of parallel computation. Although a particular model may have a more "natural" definition, for the sake of consistency, we define a *system of n concurrent processes* as a triple (Q, q_0, Δ), where Q is a (possibly infinite) set of *system configurations*, $q_0 \in Q$ is the *initial configuration*, and $\Delta = \{\delta_1, ..., \delta_n\}$, where each δ_i is a finite set of *transitions* and all δ_i's are pairwise disjoint. For systems of only one process, we also use the notation (Q, q_0, δ), where δ is a set of transitions. Intuitively, each δ_i represents the transitions of a process M_i. Each transition $t \in \delta_i$ is a partial function[1] $t : Q \rightarrow Q$. If $t(q_1) = q_2$, we also write $q_1 \xrightarrow{t} q_2$ or $q_1 \rightarrow q_2$. If $q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} \cdots \xrightarrow{t_n} q_{n+1}$ and $\sigma = t_1 ... t_n, n \geq 0$, then we also write $q_1 \xrightarrow{\sigma} q_{n+1}$ or $q_1 \xrightarrow{*} q_{n+1}$. We define the *reachability set* of (Q, q_0, Δ) as the set $R(Q, q_0, \Delta) = \{q \mid q_0 \xrightarrow{*} q\}$. The set $R(Q, q_0, \Delta)$ is distinguished from Q in that Q is the set of *legal* configurations (i.e., those allowed by the syntactic definition of the system), whereas $R(Q, q_0, \Delta)$ is that subset of Q consisting of those configurations that can actually be reached by the system. A *halting configuration* is a configuration $q \in Q$ such that $t(q)$ is undefined for all $t \in \bigcup_{i=1}^{n} \delta_i$. A *finite computation* of

[1] In order to allow multiple transitions to cause the same action, we should technically define a transition as a symbol denoting a partial function. However, our definition allows us to avoid introducing additional notation that may lead to confusion. In any case, we do allow multiple transitions to cause the same action.

(Q, q_0, Δ) is a finite sequence σ of transitions in δ such that $q_0 \xrightarrow{\sigma} q$ for some $q \in Q$, and $t(q)$ is undefined for all $t \in \bigcup_{i=1}^{n} \delta_i$. An *infinite computation* of (Q, q_0, Δ) is an infinite sequence σ of transitions in $\bigcup_{i=1}^{n} \delta_i$ such that for each finite prefix σ' of σ, there is a $q \in Q$ for which $q_0 \xrightarrow{\sigma'} q$. A *computation* of (Q, q_0, Δ) is either a finite computation or an infinite computation. (Note that according to this definition, a proper prefix of a computation is not a computation.) Let $C(Q, q_0, \Delta)$ denote the set of all computations of (Q, q_0, Δ). A system (Q, q_0, Δ) is said to be *self-stabilizing* iff for every $q \in Q$, every computation σ of (Q, q, Δ) has a finite prefix σ' such that $q \xrightarrow{\sigma'} q'$ for some $q' \in R(Q, q_0, \Delta)$. (This definition of self-stabilization is simply a reformulation of the definition given in [Dij73].)

All of the classes of systems we discuss in this section can be defined by restricting the above definition of a concurrent system. (A sequential system is just a concurrent system of 1 process.) Usually, it is a straightforward matter to translate the standard definition of some particular system class to some restriction of a general concurrent system. In such cases, we will not give the translation, and we will use whichever characterization is more convenient.

Much of this paper involves the simulation of one system class by another. In order to conclude that some type of simulation does not exist, we need to formalize the concept of simulation. In order to require a simulation to preserve the behavior of each individual process, we first make the following definitions. Let Δ^{∞} denote the set of all finite and infinite sequences of transitions in $\bigcup_{i=1}^{n} \delta_i$. For a sequence $\sigma \in \Delta^{\infty}$, let the *projection* of σ onto process i, denoted $\pi_i(\sigma), 1 \leq i \leq n$, be the homomorphism defined by

- $\pi_i(\epsilon) = \epsilon$ (where ϵ denotes the empty string);
- $\pi_i(t) = \epsilon$ if $t \notin \delta_i$;
- $\pi_i(t) = t$ otherwise; and
- $\pi_i(t\sigma) = \pi_i(t)\pi_i(\sigma)$.

Let $C_i(Q, q_0, \Delta) = \{\pi_i(\sigma) \mid \sigma \in C(Q, q_0, \Delta)\}, 1 \leq i \leq n$. We say that a concurrent system (Q', q_0', Δ') of n processes *simulates* a concurrent system (Q, q_0, Δ) of n processes iff

- there is a homomorphism $h : \Delta'^{\infty} \rightarrow \Delta^{\infty}$ such that
- $h(C(Q', q_0', \Delta')) = C(Q, q_0, \Delta)$,
- $h(C_i(Q', q_0', \Delta')) = C_i(Q, q_0, \Delta), 1 \leq i \leq n$, and
- for any computation σ' of (Q', q_0', Δ'), $\pi_i(h(\sigma'))$ is finite iff $\pi_i(\sigma')$ is finite, $1 \leq i \leq n$.

We then call h the *simulation homomorphism*. In order to motivate this definition of simulation, let $M = (Q, q_0, \Delta)$ and $M' = (Q', q_0', \Delta')$ be arbitrary systems. Intuitively, if M' simulates M, any computation σ' of M' "generates" some computation σ of M. Thus, we can assume that with each transition in Δ' there is associated some (possibly empty) finite sequence of transitions from Δ; i.e., we have a homomorphism $h : \Delta'^{\infty} \rightarrow \Delta^{\infty}$. Since each computation of M must be simulated by some computation of M', and each computation of M' should simulate some computation of M, $h(C(M')) = C(M)$. Also, since the behavior of individual processes should be preserved, $h(C_i(M')) = C_i(M)$. Finally, only finite (infinite) computations

of individual processes of M should be simulated by finite (infinite, respectively) computations of individual processes of M', so for any computation σ' of (Q', q_0', Δ'), $\pi_i(h(\sigma'))$ is finite iff $\pi_i(\sigma')$ is finite.

If $\mathcal{M}_1$ and $\mathcal{M}_2$ are two system classes, and there is a function $f : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ such that for all $M \in \mathcal{M}_1$, $f(M)$ simulates M, we say $\mathcal{M}_2$ *simulates* $\mathcal{M}_1$. We then call f the *simulation*. If for all $M \in \mathcal{M}_1$, $f(M)$ is self-stabilizing iff M is self-stabilizing, we say f is a *self-stabilization preserving simulation*. If for all $M \in \mathcal{M}_1$, $f(M)$ is self-stabilizing, we say f is a *self-stabilization forcing simulation*.

The reader should note two important points from the above definitions. First, we make no requirements as to how easy it is to either find or compute each f and h. In particular, even though f must be computable in order to be used in the simulation paradigm, we do not make this requirement of a simulation in general. This generality provides for a very weak definition of simulation, yielding some very strong negative results. The second point is that our systems have no input. If inputs were to be considered, we would need to define self-stabilization so that for any input, from any configuration containing that input, all computations eventually reach a configuration reachable from the initial configuration having that input; i.e., we would assume the input to be incorruptible. Ignoring inputs serves to make our negative results even stronger (i.e., no simulation is possible even when inputs are not considered).

One of the factors that tends to disrupt self-stabilization concerns what we call "isolation." We say an *isolation* occurs at a configuration q of M if there exist a computation σ from q and distinct computations σ_1 and σ_2 from distinct configurations q_1 and q_2, respectively, such that for all $1 \leq i \leq n$, either $\pi_i(\sigma) = \pi_i(\sigma_1)$ or $\pi_i(\sigma) = \pi_i(\sigma_2)$, but σ is not enabled at either q_1 or q_2. Intuitively, the processes of M become partitioned into two nonempty sets S_1 and S_2 such that each process in S_i behaves as if it were executing σ_i from q_i, and any communication between the two sets is insufficient to correct this behavior. Another factor we examine concerns what we call "look-alike configurations." We say that configurations q_1 and q_2 are *look-alike configurations* if there is some computation σ enabled at both q_1 and q_2. Intuitively, the system does not have the power to differentiate between q_1 and q_2 because it may behave in exactly the same way upon entering either configuration.

Throughout this paper, we will use the notation σ^n to indicate the sequence σ iterated n times. Likewise, σ^ω will indicate σ iterated infinitely many times. Iteration will take precedence over concatenation, so that, for example, $\sigma\tau^3 = \sigma\tau\tau\tau$. We will also use the notation σ^* to indicate the set $\{\sigma^n \mid n \geq 0\}$, and σ^+ to indicate the set $\sigma^* - \epsilon$. We can then construct expressions denoting sets in the following manner:

- t denotes $\{t\}$;
- $\sigma\tau$ denotes $\{\sigma_i\tau_j \mid \sigma_i \in \sigma \text{ and } \tau_i \in \tau\}$; and
- σ^* denotes $\{\sigma_1\sigma_2...\sigma_n \mid n \geq 0 \text{ and } \sigma_i \in \sigma \text{ for } 1 \leq i \leq n\}$.

It should always be clear from the context whether a given expression indicates a single sequence or a set of sequences.

3 Summary of Results

Throughout the remainder of the paper, we present our results concerning various classes of concurrent systems. These results are summarized in Tables 1 and 2. The proofs of many of these results have the

System Class	Forced	
	With Halting	Without Halting
Cellular Arrays	no	yes
Linear Cellular Arrays	no	yes.
Turing machines	no	yes
Finite-state machines	yes	yes
CSP	no	no
1 reader / 1 writer shared memory programs	no	?
Communicating finite-state machines	no	?
Bounded communicating finite-state machines	no	no
Petri nets	no	no
Petri nets with capacities	no	no

Table 1 — Results involving simulations of one class by itself.

Simulated Class	Simulating Class	With Halting		Without Halting	
		Preserved	Forced	Preserved	Forced
Cellular Arrays	Linear Cellular Arrays	—	—	yes	yes
Finite-state machines	Turing machines	no	no	yes	yes
1 reader / 1 writer shared memory programs	Boolean CSP	no	no	no	no
Boolean CSP	Infinite CSP	yes	no	yes	no
Boolean CSP	Communicating finite-state machines	no	no	?	?
1 reader / 1 writer shared memory programs	Communicating finite-state machines	no	no	?	?
Vector addition systems with states	Petri nets	no	no	no	no
Vector addition systems with states	Petri nets with capacities	yes	no	yes	no

Table 2 — Results involving simulations of one class by another.

advantage of being rather short and fairly easy to follow. Unfortunately, proofs of this sort tend to give the (sometimes false) impression that the theorems are somewhat obvious. To the contrary, it has been our experience that problems involving self-stabilization are so different from other problems in distributed computing that the intuition developed by studying other problems is often misleading in the study of self-stabilization. To lend support to this claim, we reproduce the following comment of Dijkstra's concerning one of his related proofs in [Dij73]:

> Again I beg my intrigued readers to stop reading here and to try to solve the stated problem themselves, for only then will they (slowly!) build up some sympathy with my difficulties: the problem has been with me for many months, while I was oscillating between trying to find a solution — and many an at first sight plausible construction turned out to be wrong! — and trying to prove the non-existence of a solution. And all the time I had no indication in which of the two directions to aim, nor of the simplicity or complexity of the argument — if any! — that would settle the question.

In order to focus on the meaning of our results, rather than on the technical details of their proofs, we have omitted many of the proofs from the main body of this paper. The omitted proofs appear in the Appendix.

We have uncovered three main factors that tend to disrupt self-stabilization: halting, isolation, and look-alike configurations. Of these three, halting is the most familiar (as far as we know, isolation and look-alike configurations are new concepts). Furthermore, it is not hard to see that halting is very likely to interfere with self-stabilization, since the "bad" configuration the system might enter could conceivably be a halting configuration. The system would then have no way of recovering, since it would have already halted. Thus, it follows immediately from the definition of self-stabilization that in a self-stabilizing system, all halting configurations must be reachable. For most system classes, this restriction causes a loss of computational power (see Table 1). A more interesting observation from Table 1, however, is that for some system classes, self-stabilization cannot be forced even when halting is disallowed. Hence, there must be other more subtle factors, such as isolation or look-alike configurations, interfering with self-stabilization. A look at Table 2 shows how halting can also interfere with simulations of one system class by another. First of all, as is shown in the first entry of Table 2, when individual processes are allowed to halt, there can be no simulation of arbitrary cellular arrays by linear cellular arrays (regardless of whether self-stabilization is preserved). The reason for this is that the communication connections in a linear cellular array form a linear chain, whereas in an arbitrary cellular array, arbitrary connections are allowed. Thus, if one process in a linear cellular array were to halt, it would split the system into two isolated components. However, Table 2 again shows entries which are unaffected by the presence of halting. Thus, it seems necessary to examine the extent to which halting affects self-stabilization before studying the more subtle issues of isolation and look-alike configurations.

4 Problems Involving Halting

Table 1 gives three system classes for which halting is the sole factor in preventing the forcing of self-stabilization: cellular arrays, linear cellular arrays, and Turing machines. (For standard definitions of cellular arrays and Turing machines, see, e.g., [IKM85, Kos74, Smi71] and [HU79], respectively.) Because celluar arrays are synchronous systems with multiple transitions executing simultaneously, they are rather awkward to define formally in terms of our definition of a system given in Section 2. For this reason, we omit in this section any formal discussion of cellular arrays, leaving their discussion to the Appendix (Theorems A.1, A.2, and their corollaries). Regarding Turing machines, we will now show that any self-stabilizing Turing machine must have an infinite computation; thus self-stabilization cannot be forced if we allow halting.

Theorem 4.1: There is no self-stabilization forcing simulation of Turing machines by Turing machines.

Proof. Consider a Turing machine M that starts in a halting configuration; i.e., its only computation is the empty computation. Suppose some self-stabilizing Turing machine M' simulates M. From the definition of simulation, all computations of M' must be finite; hence, M' has a halting configuration. Since Turing machines have infinitely long tapes, we can modify the tape contents[2] of any halting configuration of M' to generate infinitely many new halting configurations for M'. Because M' is self-stabilizing, all of these halting configurations must be reachable. From König's Infinity Lemma [Kon36], M' must have an infinite computation — a contradiction. □

[2] We assume that all TMs have at least one nonblank symbol in their respective tape alphablets.

In order to demonstrate the full effect of halting on the possibility of forcing self-stabilization on Turing machines, we now show that self-stabilization can be forced if no halting configurations are present. We should keep this result in its proper perspective. It is not a very strong result due to our weak definition of simulation. In particular, for any infinite computation, there is no bound on the maximum number of moves needed to simulate any transition in the computation. Furthermore, there is no bound on the number of moves made from an arbitrary configuration before a reachable configuration is reached. Nonetheless, it does show the existence of a self-stabilization forcing simulation.

Theorem 4.2: There is a self-stabilization forcing simulation of Turing machines with no halting configurations by Turing machines.

Proof. Let M be an arbitrary Turing machine with no halting configurations and k worktapes. We construct a self-stabilizing Turing machine M' that simulates M. M' contains $2k+1$ worktapes and operates as follows. M' simulates M on k tapes in a straightforward manner. After each simulated move of M, M' scans from left to right a special tape containing a list of transitions simulated so far. When M' encounters a symbol other than a transition, it overwrites that symbol with the last transition executed and overwrites the next symbol with a blank. Let n be the number of transitions in the list. M' then blanks the first n cells of the remaining k tapes and simulates the listed transitions on these k tapes. After the simulation of the list of transitions is completed, M' compares the first n symbols on each of the two sets of k tapes, verifying that the corresponding tapes match. Once this is verified, the next move of M is simulated, and the process continues. If at any time an unexpected symbol is encountered, all tapes are erased to a length equal to the number of transitions in the list, and the entire simulation is restarted (note that this restart is never done in a computation from the initial state). It is not hard to see that M' is self-stabilizing and simulates M. □

The only entry in Table 1 for which self-stabilization can be forced even in the presence of halting is for finite-state machines; such a simulation simply consists of removing all unreachable configurations. Concerning simulations of one system class by another, Table 2 shows that the only simulation for which we can show that halting interferes with the preservation or forcing of self-stabilization is the simulation of finite-state machines by Turing machines. If the tape is removed from machine M in the proof of Theorem 4.1, then this proof shows that there is no self-stabilization preserving (or forcing) simulation of finite-state machines by Turing machines. On the other hand, let M be an arbitrary finite-state machine with no halting configurations. By adding a storage tape (that is always ignored) to M, we have a Turing machine M' with no halting configurations that simulates M. From Theorem 4.2, there is a self-stabilizing Turing machine M'' that simulates M' (and hence M). To show that there is a self-stabilization preserving simulation of M by a Turing machine, note that if M is self-stabilizing, M'' preserves self-stabilization; otherwise, by adding a new state q to the finite-state control of M'' so that q can never be entered from the outside and can never be left, we have a Turing machine that simulates M and preserves (the absence of) self-stabilization. (Note that since it is decidable whether M is self-stabilizing, this construction is effective; in general, however, constructions need not be effective to show the existence of a simulation, which is simply a function.) Besides Turing machines and finite-state machines, halting seems to affect self-stabilization to some degree on Boolean programs in which communication takes place exclusively via shared variables having exactly one reader and one writer, on communicating finite-state machines, and on Boolean CSP, although at this time we do not know the full extent of these effects. In particular, self-stabilization cannot be forced on either 1 reader / 1 writer shared memory programs or communicating finite-state machines if halting is allowed. Furthermore,

there is no self-stabilization preserving (forcing) simulation of either Boolean CSP or 1 reader / 1 writer shared memory programs by communicating finite-state machines if halting is allowed. However, we do not know whether any of these results hold in the absence of halting. The proofs of all of these results involve isolation, which we discuss in more detail in the next section.

It can be seen from the proofs in this section that when halting affects self-stabilization, it tends to do so in a straightforward manner. In the next two sections, we examine factors interfering with self-stabilization in more subtle ways.

5 Problems Involving Isolation

In this section, we examine the effects of isolation on self-stabilization. The primary system class we discuss in this section is the class of CSP systems [Hoa78]. We first illustrate the effect of isolation by showing that self-stabilization cannot be forced on CSP systems. By using a similar strategy, we can also show that self-stabilization cannot be preserved by simulations of shared memory programs by CSP systems. We also show that when Boolean CSP systems are simulated by infinite-state CSP systems, self-stabilization can be preserved, but not forced. All of these results hold regardless of whether halting is allowed. By using halting, we can extend these techniques to obtain other results shown in Tables 1 and 2 concerning communicating finite-state machines [BZ83] and shared memory programs; however, at this time we do not know whether these results hold when halting is not allowed. It might also be noted that a special case of one of Dijkstra's proofs in [Dij73] may actually be viewed as a proof via isolation (the astute reader might wish to verify this claim).

We now define the class of CSP systems [Hoa78]. CSP processes communicate with each other via message passing. The command "$P\ !\ a$" is the **send** command, interpreted as "**send** to process P message a." Likewise, the command "$P\ ?\ x$" is the **receive** command, interpreted as "**receive** from process P a message to be stored in variable x." The communication takes place in a synchronous fashion; i.e., if M_1 sends a message to M_2, neither process may continue until the communication is complete. In terms of our formal definition of a system of concurrent processes, no transition $t_i \in \delta_i$ representing a **send** to M_j may take place unless it enables a transition $t_j \in \delta_j$ representing a **receive** from M_i. After t_i takes place, t_j cannot be disabled until it takes place, and no other transitions from $\delta_i \cup \delta_j$ can occur until t_j takes place. It is also possible to use **receive** commands in the guards of guarded commands. In this case, the value of the **receive** command is true when input is received and false when the other process (i.e., the one from which the message is to be received) has terminated. The guard remains unevaluated until one of these two events occurs. An alternative command in which none of the guards is evaluated is suspended until some guard is evaluated. The variables in a CSP system are potentially unbounded. A *Boolean CSP system* is a CSP system in which Boolean variables are used instead of unbounded variables. For a detailed description of CSP, see [Hoa78]. We will now show that self-stabilization cannot be forced on CSP systems.

Theorem 5.1: There is no self-stabilization forcing simulation of CSP systems by CSP systems.

Proof. Let M be the following CSP system:

M_1:: [true → [M_2 ! 0; s_1
*[true → skip]] s_2
□ true → [M_2 ! 1; s_3
*[true → skip]]] s_4

M_2:: [M_1 ? a; t_1
[$a = 0$ → *[true → skip] t_2
□ $a = 1$ → *[true → skip]]] t_3

Suppose some CSP system $M' = (M_1', M_2')$ simulates M, and let h be the simulation homomorphism. Since M' can simulate each of the computations having a prefix $s_1 t_1 s_2^n t_2$, from König's Infinity Lemma [Kon36], M' has an infinite computation σ' such that $h(\sigma') = s_1 t_1 s_2^\omega$ in which M_2' does not terminate. Along σ', M_1' reaches a local configuration C_1, after which M_1' progresses indefinitely (to simulate s_2^ω) without any communication with M_2'. (Note that if M_2' had terminated, M_1' would have been able to detect this via a receive command as a guard of a guarded command.) By similar reasoning, M_2' can reach a local configuration C_2 after which M_2' progresses indefinitely (to simulate t_3^ω) without any communication with M_1'. Now consider the computation in M' where M_1' and M_2' start at C_1 and C_2, respectively, and each of them progresses infinitely often. This computation simulates a computation in M that has infinitely many s_2s and infinitely many t_3s. Since such a computation can never be executed in M, M' is not self-stabilizing. □

This proof is valid for both infinite-state CSP systems and Boolean CSP systems. Furthermore, the above proof shows that there is no self-stabilization forcing simulation of Boolean CSP systems by infinite-state CSP systems. However, it is easily seen that there is a self-stabilization preserving simulation of Boolean CSP systems by infinite-state CSP systems: given a Boolean CSP system M, we construct a system M' by interleaving the statements of M with statements that force a restart if any variable has a value greater than 1. M' is clearly self-stabilizing iff M is.

The key feature of the above proof technique is that one process may execute arbitrarily many transitions while the other process is executing none. Such a situation is quite common in shared-memory programs — particularly if no wait statement is available. We can therefore extend this technique to show that there is no self-stabilization preserving (or forcing) simulation of Boolean programs communicating exclusively through shared variables by Boolean CSP programs; in fact, this result holds even when the shared variables are required to have at most one reader and one writer (see the Appendix, Theorem A.4).

Other results involving isolation include the fact that self-stabilization cannot be forced on 1 reader / 1 writer shared memory programs. Also, there is no self-stabilization preserving (forcing) simulation of either Boolean CSP systems or 1 reader / 1 writer shared memory programs by communicating finite-state machines. However, we do not know at this time whether any of these results hold in the absence of halting. See the Appendix, Theorems A.3, A.7, and A.8, for proofs of these results.

6 Problems Involving Look-Alike Configurations

A system class that illustrates very nicely the problems with look-alike configurations is that of Petri nets [Pet81, Rei85]. The set of configurations Q for a Petri net is the set of nonnegative integer vectors of a specified dimension k. Each transition in a Petri net may be defined by a k-dimensional nonnegative integer vector u and a k-dimensional integer vector v, $u + v \geq 0$, in the following manner: $t_{u,v}(w) = w + v$ for all $w \geq u$. This notation for a Petri net closely parallels the vector replacement system notation; see, e.g., [Kel72]. In terms of more conventional definitions of Petri nets (e.g., [Pet81, Rei85]) k is the number of

places, the configuration vectors give the number of tokens on each place, the vector u above describes the number of incoming arcs from each place to the transition $t_{u,v}$, and the vector v above describes the net effect of firing $t_{u,v}$. It is easily seen that any infinite computation in the Petri net (N^k, w, δ) is also a computation in (N^k, w', δ) if $w < w'$; hence, if there is an infinite computation from w, w and w' are look-alike configurations. The next lemma and its corollary illustrate how look-alike configurations affect self-stabilization in Petri nets.

Lemma 6.1: If the self-stabilizing Petri net (N^k, v_0, δ) has an infinite computation, then for any $w \in N^k$, there is a $w' \geq w$ such that $w' \in R(N^k, v_0, \delta)$.

Proof. Suppose (N^k, v_0, δ) has an infinite computation σ and that there is a $w \in N^k$ such that for all $w' \geq w$, $w' \notin R(N^k, v_0, \delta)$. Let $w' = v_0 + w$, and consider the computation σ in (N^k, w', δ). The set of vectors reached in the computation from w' is simply the set of vectors reached in the computation from v_0 with w added to each. Hence, each vector reached in the computation from w' is $\geq w$, and is therefore not in $R(N^k, v_0, \delta)$. Therefore, (N^k, v_0, δ) is not self-stabilizing — a contradiction. □

Corollary 6.1: If the self-stabilizing Petri net (N^k, v_0, δ) has an infinite computation, then there is a $w \in R(N^k, v_0, \delta)$ such that for all $t \in \delta, t(w)$ is defined.

We can now show that self-stabilization cannot be forced in Petri nets.

Theorem 6.1: There is no self-stabilization forcing simulation of Petri nets by Petri nets.

Proof. Consider the Petri net P shown in Figure 1. Suppose the self-stabilizing Petri net $P' = (N^k, v_0, \delta)$ simulates P. Since all computations of P are infinite, all computations of P' are infinite. Thus, from Corollary 6.1, there is a $w \in R(P')$ such that for all $t \in \delta$, $t(w)$ is defined. Since t_1^ω is a computation of P, there must be some transition $t_1' \in \delta$ simulating a finite, nonempty sequence of t_1s. Likewise, since $t_2 t_3^\omega$ is a computation of P, there must be some transition $t_3' \in \delta$ simulating a finite, nonempty sequence of t_3s. Thus, t_1' and t_3' are both enabled at $w \in R(P')$. However, there is no reachable configuration of P at which both t_1 and t_3 are enabled. Thus, P' does not simulate P — a contradiction. Therefore, no simulation forces self-stabilization. □

Under the above definition of Petri nets, self-stabilization cannot be forced on the Petri net in Figure 1. On the other hand, if we allow explicit capacities to be given for the number of tokens on certain places, self-stabilization can be forced on this particular Petri net using the construction given in the proof of Theorem A.11. However, by using more careful arguments, we can show that Theorem 6.1 holds even for this more general definition. Theorem 6.1 can also be extended in a very natural manner to show that there is no self-stabilization preserving (or forcing) simulation of vector addition systems with states (VASSs) [HP79] by Petri nets. This result is of particular interest because Petri nets and VASSs are often considered equivalent formalisms (see, e.g., [HP79, Pet81]). On the other hand, there is a self-stabilization preserving simulation of VASSs by Petri nets with capacities. Another extension of the techniques given in Theorem 6.1 yields the result that self-stabilization cannot be forced upon either general systems of communicating finite-state machines [BZ83] or systems of communicating finite-state machines whose channel contents never exceed some maximum number of messages. These results are formally shown in the Appendix, Theorems A.5, A.6, A.9, A.10, and A.11.

A Appendix

In this Appendix, we present the proofs omitted in the main body of the paper. The first proofs we give are those concerning cellular arrays. Since it is rather awkward to define cellular arrays in the terminology of our formal definition of a concurrent system, after first giving a standard formal definition, we will explicitly show how this standard definition may be translated into our terminology. A *cellular array* is a finite set $\{M_1, ..., M_n\}$ of finite-state machines. The machines operate in a synchronous fashion controlled by a clock; i.e., each time the clock fires, all machines that have not yet halted change states nondeterministically according to their respective *next-move relations*. The particular format of the next-move relation of a given machine depends upon the *topology* associated with the system. The topology is a mapping $g : \{1, ..., n\} \rightarrow 2^{\{1,...,n\}}$ such that $i \in g(i)$ for $1 \leq i \leq n$. Informally, the topology specifies which processes the machine may reference in determining its next state. More formally, let $g(i) = \{i_1, ..., i_k\}$, and let the state set of each M_j be given by Q_j. Then the next-move relation δ_i of M_i is a subset of $Q_{i_1} \times \cdots \times Q_{i_k} \times Q_i$. The next-move relation is interpreted to mean that if M_{i_j} is in the state q_{i_j} for $1 \leq j \leq k$, M_i may move to state q_i iff $(q_{i_1}, ..., q_{i_k}, q_i) \in \delta_i$. There are two types of states in each machine: halting states and nonhalting states. If q_{i_j} is a halting state, then δ_{i_j} contains no tuples having q_{i_j} as a component; otherwise, for all $q_{i_1}, ..., q_{i_{j-1}}, q_{i_{j+1}}, ..., q_{i_k}$, there is at least one q_i such that $(q_{i_1}, ..., q_{i_k}, q_i) \in \delta_{i_j}$. Thus, in a nonhalting state, there is always some move available, regardless of the states of the other machines.

We will now show how the cellular array $(M_1, ..., M_n)$ defined above may be defined as a concurrent system (Q', q_0', Δ'). The main problem is that in a cellular array, transitions from different machines execute simultaneously. Our concurrent system (Q', q_0', Δ') will mimic this behavior by executing a "simultaneous" collection of transitions sequentially in the order of their machine subscripts. Since each machine in a cellular array executes a transition at each clock cycle until it halts, such a serial representation will give an unambiguous description of the actual computation. It will also be clear that (Q', q_0', Δ') is self-stabilizing iff $(M_1, ..., M_n)$ is.

Let $(q_1, ..., q_n)$ be a configuration of $(M_1, ..., M_n)$ (i.e., each M_i is in state q_i). For $1 \leq i \leq n$, let $P_i(q_1, ..., q_n)$ be the set of machines M_j such that $j < i$ and q_j is not a halting state; i.e., $P_i(q_1, ..., q_n)$ will be the set of machines whose moves will be simulated prior to the move of M_i. Let $\delta_i(q_1, ..., q_n)$ be the set of transitions in δ_i enabled at $(q_1, ..., q_n)$. Let $D_i(q_1, ..., q_n) = \delta_{i_1}(q_1, ..., q_n) \times \cdots \times \delta_{i_k}(q_1, ..., q_n)$ if $P_i(q_1, ..., q_n) = \{M_{i_1}, ..., M_{i_k}\} \neq \emptyset$, and $D_i(q_1, ..., q_n) = \{\emptyset\}$ if $P_i(q_1, ..., q_n) = \emptyset$. The set of configurations Q' will be composed of a set NHC of nonhalting configurations and a set HC of halting configurations. We define $NHC = \{(q_1, ..., q_n, T) \mid q_j \in Q_j$ for $1 \leq j \leq n$, and for some i, q_i is a nonhalting state and $T \in D_i(q_1, ..., q_n)\}$. The components $q_1, ..., q_n$ give the current states of each of the machines, and T gives a prefix of some sequence of transitions simulating one step of the cellular array. We define $HC = \{(q_1, ..., q_n, \emptyset) \mid q_j$ is a halting state of M_j for $1 \leq j \leq n\}$, and $Q' = NHC \cup HC$. Let $t \in \delta_i$ for some $1 \leq i \leq n$. We define the transition $t' \in \delta_i'$ by

- $t'(q_1, ..., q_n, T) = (q_1, ..., q_n, T \cup \{t\})$ if $t \notin T$ and $(q_1, ..., q_n, T), (q_1, ..., q_n, T \cup \{t\}) \in Q'$; and
- $t'(q_1, ..., q_n, T) = (p_1, ..., p_n, \emptyset)$ if $(q_1, ..., q_n, T) \in Q'$, $(q_1, ..., q_n, T \cup \{t\}) \notin Q'$, t is enabled in $(q_1, ..., q_n)$, and $(M_1, ..., M_n)$ reaches $(p_1, ..., p_n)$ upon simultaneously executing the transitions in $T \cup \{t\}$.

Thus, if from configuration $(q_1, ..., q_n)$ of $(M_1, ..., M_n)$ the set of transitions $\{t_{i_1}, ..., t_{i_k}\}$ simultaneously

fire to produce $(p_1, ..., p_n)$, this action is simulated by $(q_1, ..., q_n, \emptyset) \xrightarrow{t'_{i_1}} (q_1, ..., q_n, \{t_{i_1}\}) \xrightarrow{t'_{i_2}} \cdots \xrightarrow{t'_{i_{k-1}}} (q_1, ..., q_n, \{t_{i_1}, ..., t_{i_{k-1}}\}) \xrightarrow{t'_{i_k}} (p_1, ..., p_n, \emptyset)$. We now define $\Delta' = \{\delta'_1, ..., \delta'_n\}$, where each $\delta'_i = \{t' \mid t \in \delta_i\}$. Finally, we let $q'_0 = (q_1, ..., q_n, \emptyset)$, where q_i is the initial state of M_i for $1 \leq i \leq n$. It is now a straightforward matter to verify that (Q', q'_0, Δ') exhibits the desired behavior.

A class of cellular arrays of particular interest is the class of linear cellular arrays. A *linear cellular array* is a cellular array $(M_1, ..., M_n)$ whose topology g is defined by

- $g(1) = \{1, 2\}$;
- $g(i) = \{i-1, i, i+1\}$ for $2 \leq i \leq n-1$; and
- $g(n) = \{n-1, n\}$.

We will first show that self-stabilization cannot, in general, be forced upon either cellular arrays or linear cellular arrays. In order to show the importance of halting to the proof of this theorem, we will then show that self-stabilization can be forced upon both cellular arrays with no halting states and linear cellular arrays with no halting states.

Theorem A.1: There is no self-stabilization forcing simulation of cellular arrays (linear cellular arrays) by cellular arrays (linear cellular arrays).

Proof. Let $M = (M_1, M_2)$ be the cellular array shown in Figure 2. Two computations of M are possible: $\sigma_1 = s_1 t_1 s_3 t_2 s_3^\omega$, and $\sigma_2 = s_2 t_1 t_3 t_4^\omega$. (Recall that the transitions in each of the pairs $s_i t_j$ are executed simultaneously in the actual cellular array.) Let $M' = (M'_1, M'_2)$ be any cellular array that simulates M. Since M' must be able to simulate both σ_1 and σ_2, both M'_1 and M'_2 must have halting states. Thus, M' has a halting configuration, and is therefore not self-stabilizing. Since any cellular array of two machines can be viewed as a linear cellular array, the theorem follows. □

Theorem A.2: There is a self-stabilization forcing simulation of cellular arrays with no halting states by linear cellular arrays.

Proof. Let $M = (M_1, ..., M_n)$ be an arbitrary cellular array. Since each M_i is a finite-state machine, we can construct one finite-state machine A describing the entire system M; i.e., each state of A is a tuple $(q_1, ..., q_n)$, where each q_i is a state of M_i. We may then remove all unreachable states from A. We will now describe a self-stabilizing linear cellular array $M' = (M'_1, ..., M'_n)$ that simulates M. All of the work will actually be done by the machine M'_1. This machine will select an infinite sequence of transitions from A representing a computation of M. After each transition t of A is selected, M'_1 will wait long enough for all the other machines to determine that t has been selected. At this time, all the machines simulate t. M'_1 then selects the next transition, and the simulation continues in the same manner.

More formally, let Q_A be the state set of A, and let δ_A be the transition relation of A. The state set of M_i, $1 \leq i \leq n$, will be $\{(q, i) \mid q \in Q_A\} \cup \{(t, j, i) \mid t \in \delta_A \text{ and } i \leq j \leq n\}$. The next-move relation of M'_1 is defined solely in terms of the current state of M'_1. If M'_1 is in state $(q, 1)$, it may move to any state $(t, 1, 1)$ such that t is enabled at q in A. If M'_1 is in any state $(t, j-1, 1)$, $2 \leq j \leq n$, it moves to state $(t, j, 1)$. If M'_1 is in any state $(t, n, 1)$, it moves to the state $(q', 1)$ such that transition t places A in state q'; moves of this last type will simulate moves of M_1. For $2 \leq i \leq n$, the next-move relation of M'_i is defined solely in terms of the state of M'_{i-1}. If M'_{i-1} is in some state $(q, i-1)$, M'_i moves to state (q, i). If M'_{i-1} is in some

state $(t, j-1, i-1)$, $i \leq j \leq n$, M_i' moves to state (t, j, i). If M_{i-1}' is in some state $(t, n, i-1)$, M_i' moves to the state (q', i) such that transition t places A in state q'; moves of this last type will simulate moves of M_i. The initial state of M' has each M_i' in the state (q, i) such that q is the initial state of A. It is not hard to see that in any computation from the initial state, M' simulates the next move of M every $n+1$ moves; thus, M' simulates M. Furthermore, it is not too difficult to see that from any configuration, after at most $i-1$ moves, M_i' is in some state consistent with M_1'. Since all states of M_1' are clearly reachable from the initial configuration, M' must be self-stabilizing. □

The above simulation is not very satisfying because the entire computation is actually being done by one machine, M_1'; the machines $M_2', ..., M_n'$ simply execute the transitions that M_1' tells them to execute.

The following corollaries follow immediately from Theorem A.2.

Corollary A.1: There is a self-stabilization forcing simulation of celluar arrays without halting states by cellular arrays.

Corollary A.2: There is a self-stabilization forcing simulation of linear celluar arrays without halting states by linear cellular arrays.

We now introduce another class of concurrent systems, the class of Boolean programs in which each variable may be read by at most one process and written by at most one process. All communication is therefore performed via shared variables to which one process writes and from which another process reads. The syntax we use to describe these systems is similar to CSP without the communication commands (see [Hoa78]). We will now show that self-stabilization cannot be forced on this type of system. The proof will use both halting and isolation.

Theorem A.3: There is no self-stabilization forcing simulation of 1 reader / 1 writer shared memory programs by 1 reader / 1 writer shared memory programs.

Proof. Consider the following system $M = (M_1, M_2)$, where all variables are initially zero:[3]

M_1:: [true → $a := 1$ s_1
□ true → [$b := 1$; s_2
*[true → skip]]] s_3

M_2:: [*[$a = 0 \wedge b = 0$ → skip]; t_1
$a = 1$ → *[true → skip]] t_2

The only variables in M, a and b, are read by M_2 and written by M_1. M_1 first nondeterministically chooses either s_1 or s_2. If it chooses s_1, it writes 1 to a and halts. If it chooses s_2, it writes 1 to b and repeatedly executes s_3. Meanwhile, M_2 repeatedly executes t_1 until M_1 executes its first transition. If M_1's first transition is s_1, M_2 then repeatedly executes t_2. Otherwise, M_2 halts. Thus, all computations of M are infinite, but in any computation, one of the processes executes only finitely many times. Therefore, in any system $M' = (M_1', M_2')$ that simulates M, both M_1' and M_2' must have the ability to halt. Since all variables can be read by at most one process, there must be a configuration of M' in which both M_1' and M_2' have halted. Thus, M' is not self-stabilizing. □

We will now show that there is no self-stabilization preserving simulation of shared memory programs by Boolean CSP systems. It is not immediately clear, however, that there is any simulation between these system classes, whether or not self-stabilization is preserved. Hence, we first sketch an example of a simulation that does not preserve self-stabilization. Let $M = (M_1, ..., M_n)$ be an arbitrary system of 1 reader / 1 writer

[3] Throughout the remainder of the paper, we will always assume the variables to be initially zero.

shared memory programs. $M' = (M'_1, ..., M'_n)$ will be a Boolean CSP system that behaves as follows. M'_1 first creates a status vector containing the values of all shared variables of M and all processes which have not halted. It then nondeterministically decides which process M_i will execute first. If $i \neq 1$, M sends the status vector to M'_i. From this point on, there will be exactly one "active" process at any given time when no communication is taking place; the other processes will be waiting for messages from all other processes. The active process, say M'_i, will nondeterministically simulate a nonempty (but possibly infinite) sequence of transitions from M_i, updating the status vector accordingly. If it has chosen to simulate a finite sequence of transitions, it then nondeterministically selects some process from the list of processes which have not halted. It then sends the updated status vector to that process, which then becomes active. If at any time a process M'_i simulates the termination of M_i, it removes its own name from the list of processes which have not halted, sends the status vector to some process that has not halted, then halts. If at any time the list of processes that have not halted contains only one process, that process remains active until it simulates a termination. The details of this simulation are left to the reader. The following theorem, shown using isolation, now shows that neither this simulation nor any other simulation can be guaranteed to either preserve or force self-stabilization.

Theorem A.4: There is no self-stabilization preserving (forcing) simulation of 1 reader / 1 writer shared memory programs by Boolean CSP systems.

Proof. Let $M = (M_1, M_2)$ be the following shared memory system:

M_1:: * [$a = 0 \rightarrow b := 1$ s_1 □ $a = 1 \rightarrow b := 0$] s_2

M_2:: * [$b = 0 \rightarrow a := 0$ t_1 □ $b = 1 \rightarrow a := 1$] t_2

M has no halting configurations, and M is clearly self-stabilizing, since all configurations are reachable. Suppose some CSP system $M' = (M'_1, M'_2)$ simulates M with simulation homomorphism h. Let $S = \{\sigma' \mid \sigma'$ is a finite prefix of some computation $\sigma'\sigma''$ of M', M'_2 has not terminated in σ', and $h(\sigma') \in s_1{}^*\}$. Since M can execute any computation beginning with $(s_1)^n t_2$ for any n, S is infinite. Hence, from König's Infinity Lemma [Kon36], there is an infinite string σ' such that any finite prefix of σ' is in S. Clearly, σ' must be a computation of M' in which M'_2 does not terminate. However, since $h(\sigma')$ contains no transitions from M_2, σ' must contain only finitely many transitions from M'_2. Thus, $\pi_1(\sigma')$ and $h(\pi_1(\sigma'))$ are both infinite. It must therefore be the case that $h(\sigma') = s_1^\omega$. Hence, there is some state of M'_1 from which there is an infinite computation simulating s_1^ω and containing no communication commands. By similar reasoning, there is a state of M'_2 from which there is an infinite computation simulating t_1^ω and containing no communication commands. There must therefore be a configuration of M' from which a computation containing infinitely many s_1s and infinitely many t_1s, but no s_2s nor t_2s, can be simulated. Since such a computation can never be executed in M, M' is not self-stabilizing. □

Note that the above theorem holds even when halting states are disallowed.

The next class we examine is that of communicating finite-state machines (CFSMs) [BZ83]. Informally, a system of CFSMs is a finite set of finite-state machines that communicate via unbounded FIFO channels. No empty channel detection is possible, and either a read or a write to some channel is performed by each transition. (See, e.g., [BZ83] for a formal definition of CFSMs.) We will first show that self-stabilization cannot be forced on systems of CFSMs. The proof uses both halting and look-alike configurations.

Theorem A.5: There is no self-stabilization forcing simulation of CFSMs by CFSMs.

Proof. Let $M = (M_1, M_2)$ be the system of two CFSMs defined as follows. M_1 contains only one state and no transitions. M_2 contains only one state and one transition, which writes some symbol to the output channel of M_2. Suppose there is a self-stabilizing system $M' = (M_1', M_2')$ of CFSMs that simulates M. Then for any computation σ' of M', $\pi_1(\sigma')$ is finite and $\pi_2(\sigma')$ is infinite. Since M_2' is finite-state and may only read finitely many symbols from its input channel, in any computation σ', M_2' must enter the same state twice without executing any reads in between. Thus, M_2' has a state q from which there is an infinite computation containing no reads. It follows from König's Infinity Lemma [Kon36] that the input channel to M_2' is bounded, say, by m. If we therefore start M_1' in some arbitrary state, M_2' in q, and the input channel to M_2' with some string longer than m symbols, there is an infinite computation in which the contents of the input channel to M_2' are unchanged. Thus, M' is not self-stabilizing — a contradiction. □

Note that in the above proof both halting and isolation are involved. At this time, we are unable to show whether the elimination of halting states might allow self-stabilization to be forced.

A system of CFSMs is said to be *bounded* if its reachability set is finite. Consider any bounded system of CFSMs having an infinite computation σ. Let q be some system configuration that is reached more than once by σ, and let α be the string written to some channel c between the first two occurrences of q. Let q' be q modified by appending α to the contents of channel c enough times so that q' is not reachable. It is not hard to see that there is an infinite computation σ' from q' in which q' is reached infinitely often. Therefore, no bounded system of CFSMs having an infinite computation is self-stabilizing. This fact implies the following theorem.

Theorem A.6: There is no self-stabilization forcing simulation of bounded CFSMs by bounded CFSMs.

We now show that there is no self-stabilization preserving simulation of Boolean CSP systems by CFSMs.

Theorem A.7: There is no self-stabilization preserving (forcing) simulation of Boolean CSP systems by CFSMs.

Proof. Consider the following CSP system M:

M_1:: [**skip**] $\qquad\qquad$ M_2:: *[**true** → **skip**]

M is clearly self-stabilizing. Furthermore, M simulates the system of CFSMs given in the proof of Theorem A.5. Thus, if there were a simulation of M by a system of CFSMs, we would have a simulation of the system of CFSMs given in the proof of Theorem A.5 by CFSMs — a simulation we have already shown does not exist. □

The same technique may be used to show the following theorem.

Theorem A.8: There is no self-stabilization preserving (forcing) simulation of 1 reader / 1 writer shared memory programs by CFSMs.

We now examine the class of vector addition systems with states (VASSs). The set of configurations of a VASS is of the form $Q \times N^k$, where Q is a finite set of machine states. The transitions are defined in terms of two machine states, q and q', and a vector $v \in N^k$ in the following manner: $t_{q,q',v}(q, w) = (q', w + v)$ for $w + v \geq 0$. Thus, a VASS may be viewed as a Petri net augmented with a finite-state control (see, e.g., [HP79]). There is therefore a straightforward simulation of Petri nets by VASSs, and Hopcroft and Pansiot [HP79] have shown how to simulate a k-dimensional VASS by a $k+3$-dimensional Petri net. However, we

show in the next theorem that in general there is no simulation of a VASS by a Petri net that preserves self-stabilization. This proof uses look-alike configurations.

Theorem A.9: There is no self-stabilization forcing (preserving) simulation of VASSs by Petri nets.

Proof. Consider the VASS M shown in Figure 3. Since every configuration of M is reachable, M is self-stabilizing. Furthermore, M simulates the Petri net in Figure 1, which we have already shown in Theorem 6.1 cannot be simulated by a self-stabilizing Petri net. □

Finally, we examine a slightly more general class of Petri nets. We define a *Petri net with capacities* in the same way as a Petri net with the exception that the set of configurations may be restricted so that certain vector coordinates (i.e., places) may not exceed specified bounds. In what follows, we first use look-alike configurations to show that self-stabilization cannot be forced upon Petri nets with capacities; we then give a self-stabilization-preserving simulation of VASSs by Petri nets with capacities.

Theorem A.10: There is no self-stabilization forcing simulation of Petri nets with capacities by Peitr nets with capacities.

Proof. Let M be the Petri net shown in Figure 4. M behaves as follows. First, t_1 may fire arbitrarily many times (possibly infinitely many times). At any time t_2 may fire once, permanently disabling t_1. After t_2 fires, M executes an infinite computation containing only t_3s and t_4s; however, t_4 may never have fired more times than t_3. Consider the marking v in which $p_1 = 1$, $p_2 = 0$, and $p_3 = 1$. (Note that v and v_0 are look-alike configurations). Clearly, v is not reachable. However, the computation $\sigma = t_1^\omega$ from v enters v infinitely often; therefore, M is not self-stabilizing. In the execution of σ from v, infinitely often some sequence of transitions $t_2 t_3^k t_4^m$, $k < m$, is enabled. Since such a sequence is never enabled in any computation of M from its initial marking v_0, this fact is enough to show that M is not self-stabilizing. We will now show that such a phenomenon occurs in any Petri net with capacities that simulates M.

Let $M' = (Q', v_0', \delta')$ be a Petri net with capacities that simulates M, and let h be the simulation homomorphism. Let S be the following set of suffixes of computations of M: $S = \{t_2 t_3^i t_4^i t_3^\omega \mid i > 0\}$. For each $i \geq 0$, there are only finitely many sequences σ of transitions from v_0' such that $h(\sigma) = t_1^i$; otherwise, from König's Infinity Lemma, M' would have an infinite computation simulating t_1^j for some $j \geq 0$. Hence, for each $i \geq 0$, there is a σ_i such that

1. $v_0' \xrightarrow{\sigma_i} w_i$ in M';
2. $h(\sigma_i) = t_1^i$; and
3. infinitely many computations in S can be simulated from w_i.

From König's Infinity Lemma, there is a computation σ of M' such that $h(\sigma) = t_1^\omega$ and for any finite prefix σ_1 of σ, there exist σ_2 and w such that $v_0' \xrightarrow{\sigma_1\sigma_2} w$ and infinitely many computations in S can be simulated from w. From [KM69], $\sigma = \sigma_1\sigma_2\sigma_3$ where $v_0' \xrightarrow{\sigma_1} v_1' \xrightarrow{\sigma_2} v_2'$, $\sigma_2 \neq \epsilon$, $v_1' \leq v_2'$, and on all bounded coordinates, v_1' and v_2' are equal (see Figure 5).

Let σ_4 be such that $v_0' \xrightarrow{\sigma_1\sigma_4} v_3'$ and infinitely many computations in S can be simulated from v_3'. Let S' be the infinite subset of S that can be simulated from v_3'. From König's Infinity Lemma, there is a τ such that $h(\tau) \in t_2 t_3^\omega$ and any finite prefix of τ simulates a prefix of some computation in S'. From [KM69], $\tau = \tau_1\tau_2\tau_3$ such that $v_3' \xrightarrow{\tau_1} v_4' \xrightarrow{\tau_2} v_5'$, $\tau_2 \neq \epsilon$, $v_4' \leq v_5'$, and in all bounded coordinates, v_4' and v_5' are equal.

Let m be such that $h(\tau_1\tau_2\tau_4) = t_2t_3^mt_4^mt_3^\omega$ for some computation τ_4 from v_5'. Clearly, $h(\tau_2) \neq \epsilon$ (otherwise $h(\sigma_1\sigma_4\tau_1\tau_2^\omega)$ is finite) and $t_2 \notin h(\tau_2)$; thus, $h(\tau_2) \in t_3^+$.

Let $v' = v_0' + v_5' - v_4'$, $\sigma' = \sigma_1\sigma_2^\omega$, and $\tau' = \sigma_4\tau_1\tau_4$. We claim that σ' is a computation from v' and that infinitely often in σ', τ' is enabled. To see this, first note that since $v_5' \geq v_4'$ and v_5' and v_4' are equal on all bounded coordinates, any infinite computation from v_0' is also a computation from v' (v_0' and v' are look-alike configurations). Clearly, $\sigma_1\sigma_2^\omega$ is a computation from v_0', and hence from v'. Let $v' \xrightarrow{\sigma_1\sigma_2^i} w_i$, and let $x = v_5' - v_4'$ and $y = v_2' - v_1'$. (Note that both x and y are nonnegative and are zero in all bounded coordinates of M'.) Since $w_i = v_1' + x + iy$, $w_i \xrightarrow{\sigma_4} v_3' + x + iy \xrightarrow{\tau_1} v_4' + x + iy = v_5' + iy$, and τ_4 is a computation from $v_5' + iy$. But $h(\tau') = t_1^jt_2t_3^kt_4^mt_3^\omega$ where $k < m$ — a suffix that can never be executed in M. Hence, w_i is unreachable for all $i > 0$, so from v', σ' never reaches a reachable configuration. Therefore, M' is not self-stabilizing. □

Since Petri nets with capacities can clearly be simulated by VASSs, we have the following corollary.

Corollary A.3: There is no self-stabilization forcing simulation of VASSs by Petri nets with capacities.

We conclude by showing that there is a self-stabilization preserving simulation of VASSs by Petri nets with capacities. The obvious strategy to use in trying to prove this theorm is to use 1-bounded places to represent each of the states in the VASS. The difficulty in this approach is dealing with the extra configurations introduced while not allowing transitions to be enabled at the wrong times. However, we are able to overcome this difficulty in the proof that follows.

Theorem A.11: There is a self-stabilization preserving simulation of VASSs by Petri nets with capacities.

Proof. For ease of explanation, we will only give a self-stabilization preserving simulation of a finite-state machine by a Petri net with capacities. It should be clear how to extend this simulation to VASSs. Let M be an arbitrary finite-state machine. Without loss of generality, assume that each transition of M changes the state of M; otherwise, we can clearly add states as necessary to enforce this condition without affecting self-stabilization. Let the state set of M be $Q = \{q_1, ..., q_n\}$. We construct a Petri net M' with n 1-bounded coordinates as follows. We represent each state q_i of M by a vector v_i in which coordinate i is 0 and all other coordinates are 1. We simulate a transition t from state q_i to state q_j by transition t' shown in Figure 6. (Note that t' is not enabled in any other configuration.) Clearly, the Petri net so constructed simulates M. We must now deal with the configurations of M' that do not correspond to any state of M; i.e., those configurations in which the number of coordinates having a value of 0 is not exactly one. The first of these configurations is the vector of all 1s. We introduce a new transition, only enabled at this vector, which brings M' to the configuration representing q_1 (see Figure 7). Finally, we can bring all other vectors to the vector of all 1s as shown in Figure 8 (note again that each of these transitions is only enabled at one vector). It should be clear that M' is self-stabilizing iff M is self-stabilizing. □

The simulation given in the above proof is much better than most of the other simulations we have given in this paper. First of all, it is a real-time simulation. Also, if we have some bound on the number of moves needed for the VASS to stabilize, the bound for the Petri net with capacities is only two greater. Unfortunately, the size of the description is exponential in the size of the description of the VASS due to the large number of transitions from "bad" configurations.

References

[BGW89] G. Brown, M. Gouda, and C. Wu. Token systems that self-stabilize. 1989. *IEEE Trans. on Computers*, to appear.

[BP89] J. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Trans. on Programming Languages and Systems*, 11:330–344, 1989.

[BYC88] F. Bastani, I. Yen, and I. Chen. A class of inherently fault tolerant distributed programs. *IEEE Trans. on Software Engineering*, 14:1432–1442, 1988.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30:323–342, 1983.

[Car87] H. Carstensen. Decidability questions for fairness in Petri nets. In *Proceedings of the 4th Symposium on Theoretical Aspects of Computer Science*, pages 396–407, 1987. LNCS 247.

[CK80] W. Cheney and D. Kincaid. *Numerical Mathematics and Computing.* Brooks/Cole, Monterey, Cal., 1980.

[Dij73] E. Dijkstra. EWD391 Self-stabilization in spite of distributed control. 1973. Reprinted in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, Berlin, 1982, pp. 41-46.

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

[EL87] E. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[Gou87] M. Gouda. *The Stabilizing Philosopher: Asymmetry by Memory and by Action.* Technical Report TR-87-12, Dept. of Computer Sciences, University of Texas at Austin, 1987.

[Hoa78] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[HP79] J. Hopcroft and J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoret. Comp. Sci.*, 8:135–159, 1979.

[HRY88] R. Howell, L. Rosier, and H. Yen. A taxonomy of fairness and temporal logic problems for Petri nets. In *Proceedings of the 13th Symposium on Mathematical Foundations of Computer Science*, pages 351–359, 1988. To appear in *Theoret. Comp. Sci.*

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.

[IKM85] O. Ibarra, S. Kim, and S. Moran. Sequential machine characterizations of trellis and cellular automata and applications. *SIAM J. Computing*, 14:426–447, 1985.

[Kel72] R.M. Keller. *Vector Replacement Systems: A Formalism for Modelling Asynchronous Systems.* TR 117, Princeton University, CSL, 1972.

[KM69] R. Karp and R. Miller. Parallel program schemata. *J. of Computer and System Sciences*, 3:147–195, 1969.

[Kon36] D. König. *Theorie der Endlichen und Unendlichen Graphen.* Akademische Verlagsgesellschaft, Leipzig, 1936.

[Kos74] S. Kosaraju. On some open problems in the theory of cellular automata. *IEEE Trans. on Computers*, C-23:561–565, 1974.

[Lam86] L. Lamport. The mutual exclusion problem: Part II — Statement and solutions. *JACM*, 33:327–348, 1986.

[LR81] D. Lehman and M. Rabin. On the advantages of free choice: A symmetric and fully distributed solution of the dining philosophers problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 133–138, 1981.

[Mul89] N. Multari. *Self-stabilizing Protocols.* PhD thesis, Dept. of Computer Sciences, University of Texas at Austin, 1989. In preparation.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. on Programming Languages and Syst.*, 4:455–495, 1982.

[OWA89] C. Özveren, A. Willsky, and P. Antsaklis. Stability and stabilizability of discrete event dynamic systems. MIT LIDS Publication, LIDS-P-1853, 1989.

[Pet81] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag, Heidelberg, 1985.

[Smi71] A. Smith. Cellular automata complexity tradeoffs. *Information and Control*, 18:466–482, 1971.

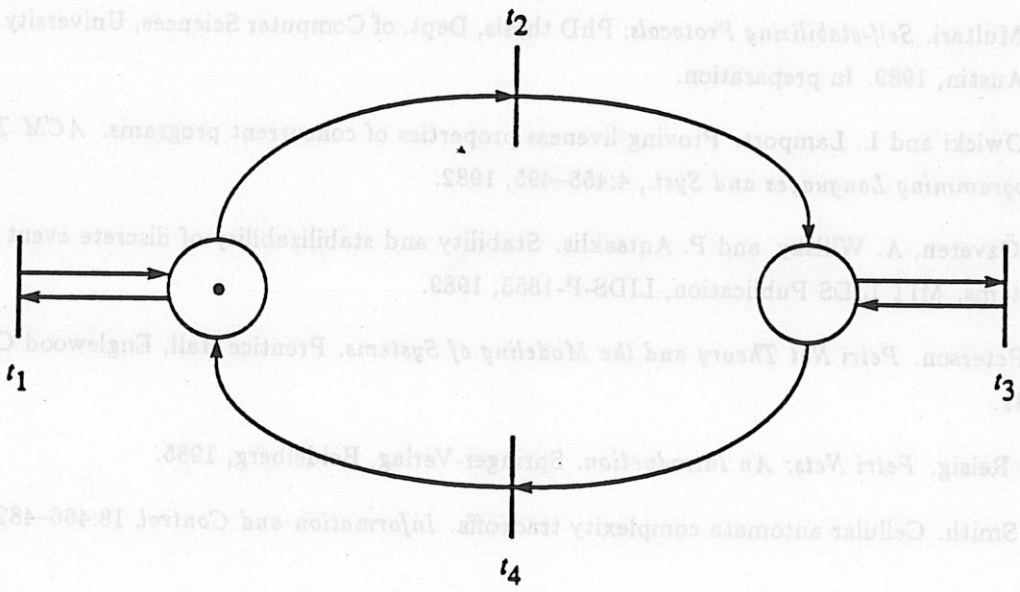
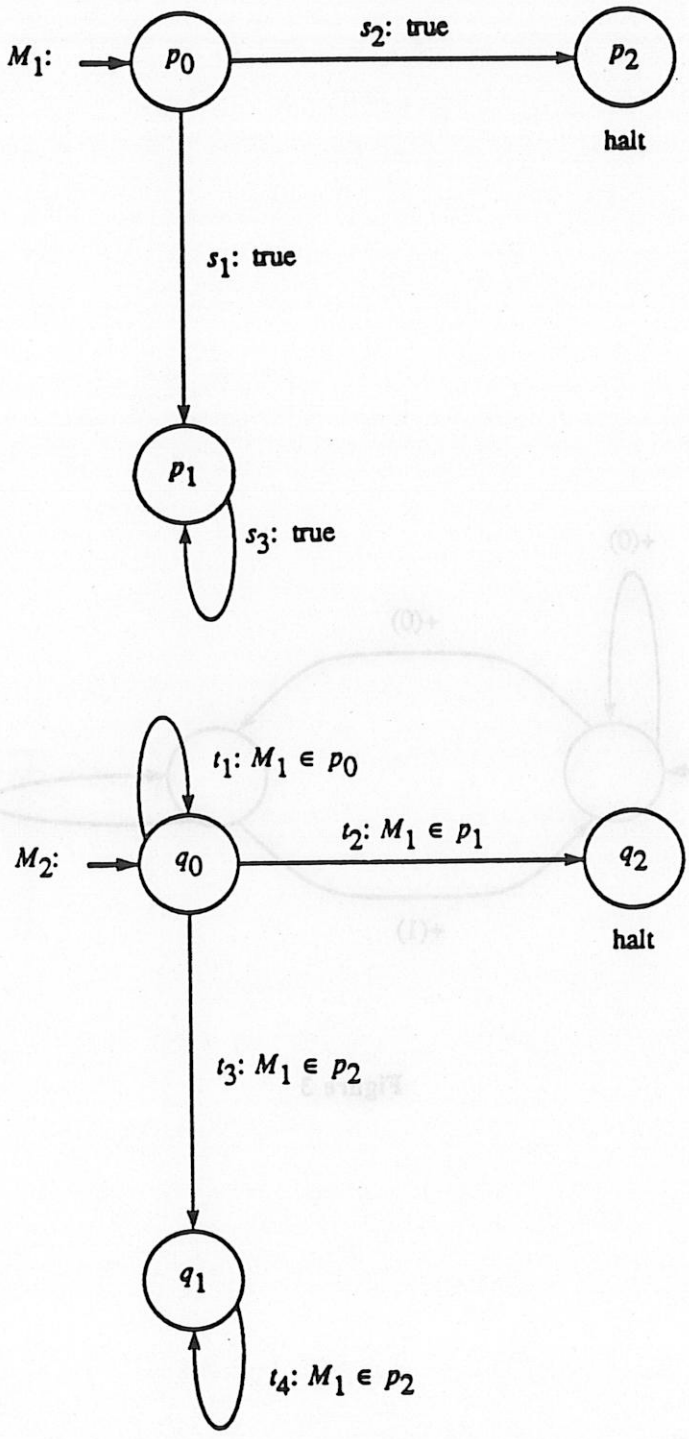


Figure 1



Arc labels of the form *t*: *cond* mean the arc represents transition *t*, and *t* is enabled only if *cond* is true.

Figure 2

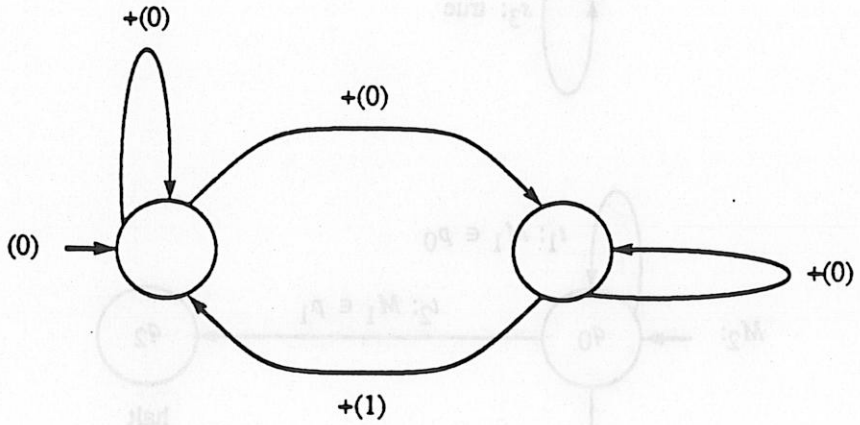


Figure 3

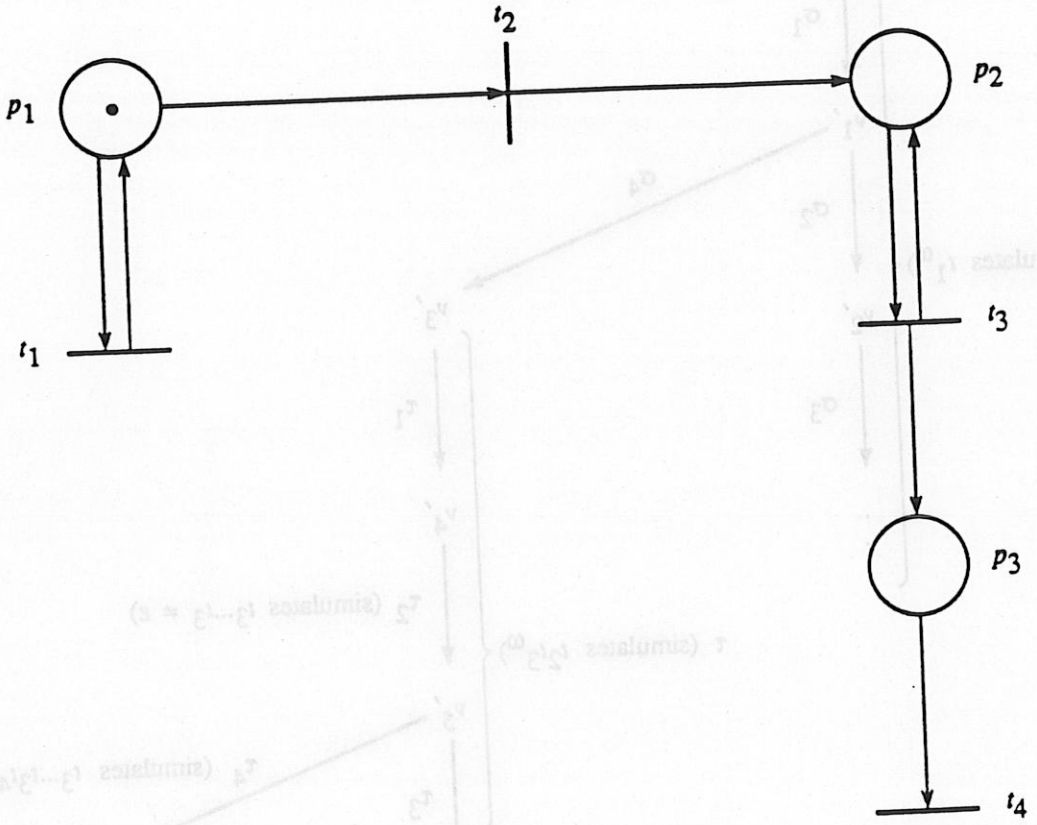


Figure 4

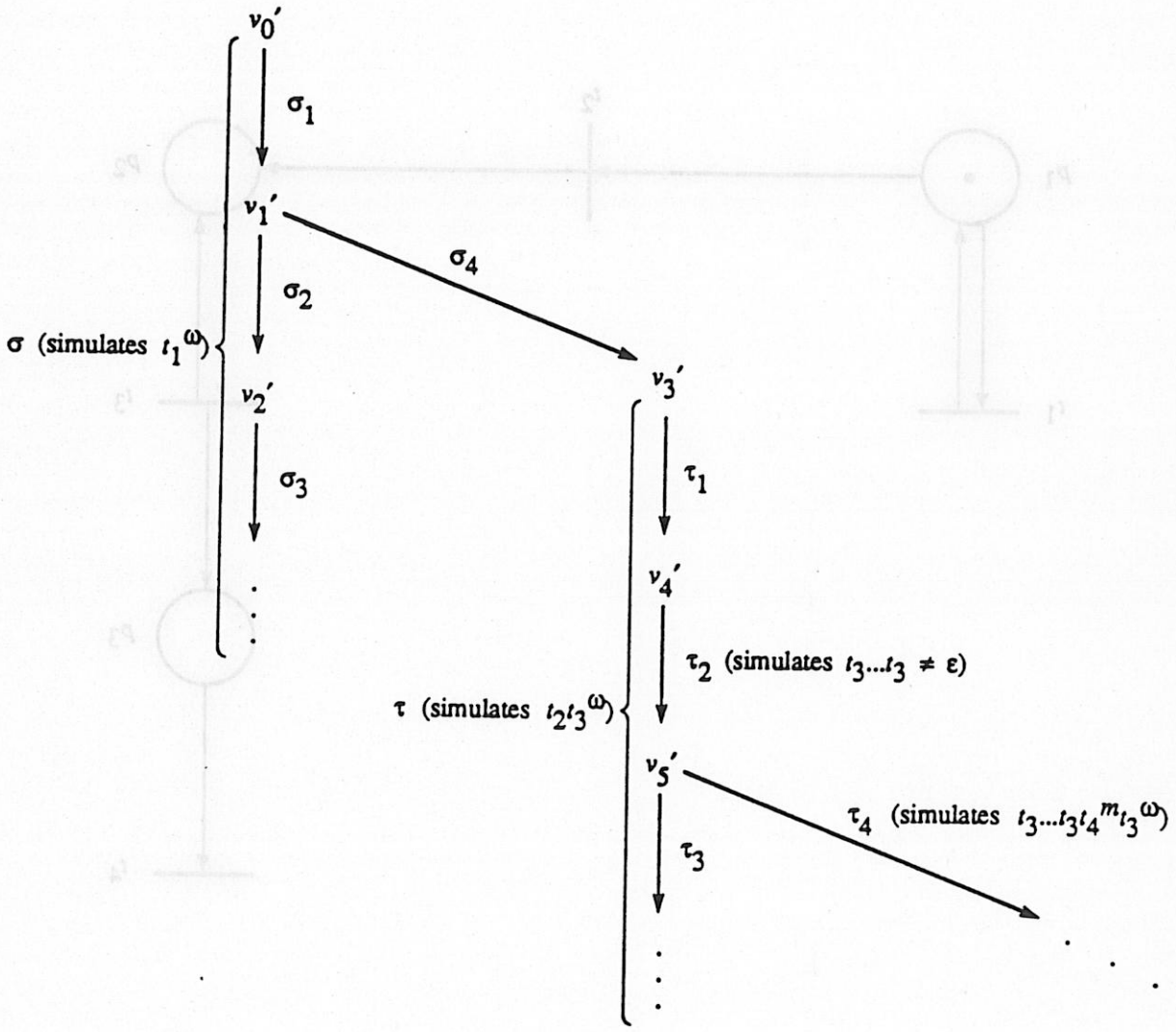

v_0'
σ_1
v_1'
σ_4
σ_2
σ (simulates t_1^{ω})
v_2'
v_3'
σ_3
τ_1
v_4'
τ_2 (simulates $t_3 \ldots t_3 \neq \varepsilon$)
τ (simulates $t_2 t_3^{\omega}$)
v_5'
τ_4 (simulates $t_3 \ldots t_3 t_4 {}^{m} t_3^{\omega}$)
τ_3

Figure 5

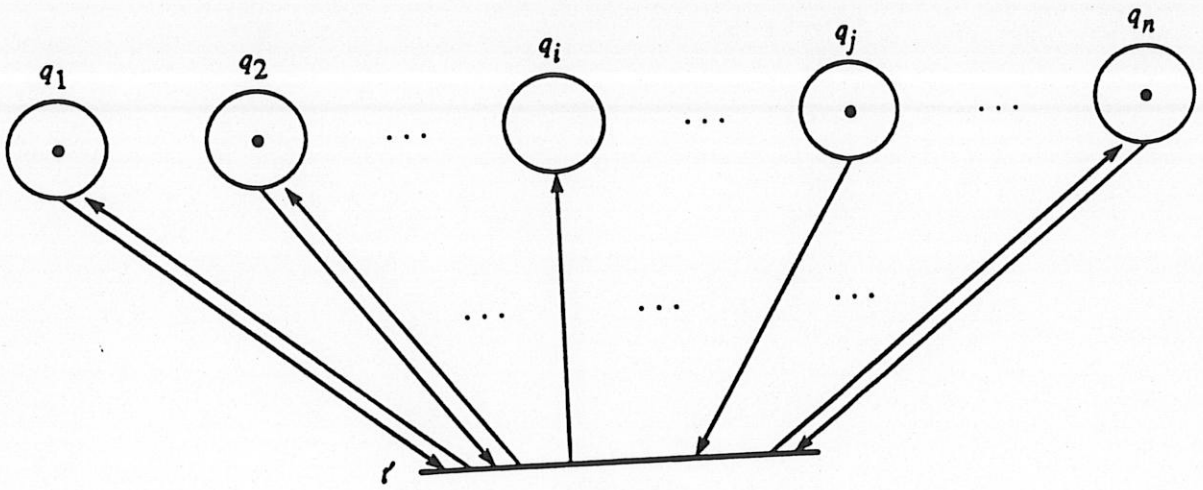


Figure 6

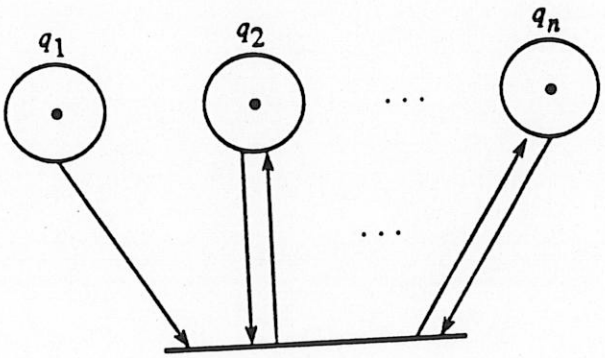


Figure 7

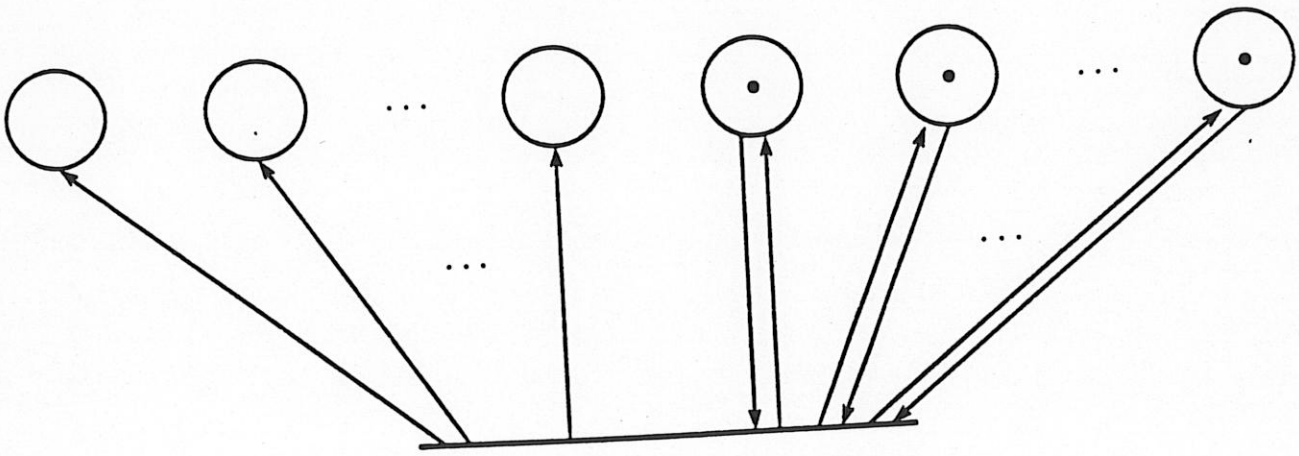
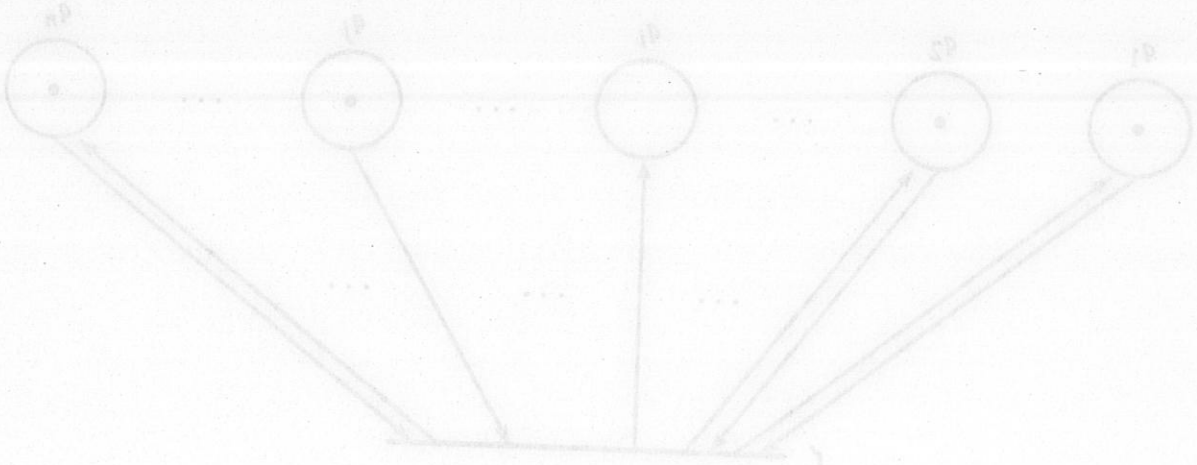


Figure 8



On Self-Stabilization, Nondeterminism, and Inherent Fault Tolerance

Farokh B. Bastani, I-Ling Yen, and Yi Zhao

Department of Computer Science
University of Houston
Houston, TX 77004

ABSTRACT

We first review self-stabilization and define it based on the required criteria and desired properties. An application of self-stabilization to detecting faulty processors in a Byzatine agreement setting is given. We then define inherent fault tolerance and illustrate it with the orignal inherently fault tolerant (IFT) example given by Dijkstra. The relationship between self-stabilization and inherent fault tolerance is that self-stabilizing programs retain stability in spite of undesireable state transitions while IFT programs possess stability in spite of the removal of some statements. We also review the role of nondeterminism in self-stabilization and inherent fault tolerance. Some theorectical aspects, namely, the correctness proof of self-stabilizing and IFT programs and the existence conditions for decentralized algorithms are presented. We conclude that a most desirable system is one that is self-stabilizing and inherently fault tolerant with decentralized control.

I. Introduction

In physics, an object is said to be in a stable state if the forces acting on it tend to restore it to its original state whenever it is perturbed from its position by a temporary external force. In the early 1970's, Dijkstra coined the term "self-stabilizing system" to refer to a distributed algorithm that could restore itself to a desired state from any arbitrary state within a finite number of steps. Dijskra illustrated this with a cyclic relaxation algorithm [Dij73] that allowed points on a unit circle to distribute themselves evenly along the circumference irrespective of the starting state. He also developed three self-stabilizing protocols for assuring mutual exclusion in a ring network where processes could only communicate with nearest neighbors [Dij74]. His algorithms required non-deterministic execution and special processes ("top" and "bottom") that behaved differently from the other processes. Kruiger extended self-stabilization to tree networks [Kru79] and LeLand applied it to develop a self-stabilizing token passing protocol [LeL79]. In his keynote address at the 3rd PPDC, Lamport bemoaned the fact that self-stabilization had not been pursued actively in the ensuing years [Lam83]. More recently, Brown, Gouda, and Wu have developed a self-stabilizing algorithm that does not require nondeterministic control and, hence, is viable for implementation using delay sensitive circuits [Bro89], and Burns and Pachl have developed an algorithm for a prime number of processes that dispenses with the need for special processes [Bur89a].

The major attraction of self-stabilization is the perceived elegance of eliminating clumsy mechanisms for detecting illegal states and initiating recovery actions. It also does not require system initialization, a task which is difficult to coordinate in distributed systems. The price paid is that the system may go through a vulnerable period when its behavior does not fully satisfy the requirements. However, for applications that require "good" behavior only on the average, e.g. load balancing and other optimization tasks, self-stabilization appears to be quite viable. It imposes very few overheads as compared with traditional all-or-nothing approaches.

Dijskra's cyclic relaxation algorithm [Dij73] has another interesting property, namely, it can tolerate the permanent failure of any single point. That is, if one of the points is immobilized, then the other points will adjust themselves properly without invoking any special actions, such as failure detection and recovery mechanisms. We observed this in 1984 and used the term "inherent fault tolerance" to distinguish it from conventional explicit fault tolerance methods such as checkpointing, rollback, and recovery. Subsequently, we developed the basic concepts of inherent fault tolerance along with several IFT programs for different applications, including a program for controlling a two dimensional robot [Bas85]. Independently, Hinton developed an IFT program for controlling a figure bending to pick up an object [Hin84] and Rudolph developed an (almost) IFT sorting algorithm using balanced networks [Rud85].

The rest of the paper is organized as follows. Section II defines some notations and the fixed point set of programs while section III defines self-stabilization and illustrates it with an example. Section IV defines inherently fault tolerant programs and section V discusses conditions under which nondeterministic process-control programs can be executed asynchronously and conditions for the existence of decentralized programs. Finally, section VI summarizes the paper and outlines some research directions.

II. System Model

In order to have a consistent definition for the topics we are discussing, we will give a system model in this section. This model will be general to all the definitions given later for self-stabilization and inherent fault tolerance.

A system can be expressed by three entities, (P, X, S), where P denotes a set of n_p statements, $P = \{p_i \mid 1 \le i \le n_p\}$; X denotes a set of external statements which can cause undesired state transition; and S denotes the system state space that is specified by certain system space constraints.

Let $P(s)$ denote the states obtained by applying P to state $s \in S$. The execution

sequence of P can be synchronous parallel, asynchronous parallel, or nondeterministic depending on the computation model. Note that the application of P to a state s in different execution models will be different. Under the synchronous parallel model, $P(s)$ is the state obtained by applying all p_i, $1 \le i \le n_p$ simultaneously till all statements complete their execution. Under the asynchronous parallel model, application of P will be over a unit time period where any p_i can start at an instance of the time period. Under the nondeterministic model, application of P refers to application of any one p_i in P. Also, we use $P^k(s)$ to denote the state in S reached by starting from s and repeatedly applying P k times. We can define the fixed point set of a program as follows.

Fixed point set. A set $FP(P)$, $FP(P) \subseteq S$, is a fixed point set of P iff $\forall x \in FP(P)::$ $(P(x) \in FP(P) \wedge \forall x' \in FP(P)::\ \exists k \ge 1, P^k(x') = x)$.

We will use $fp(P, s)$ to denote a fixed point set reached by starting from state s and applying P repeatedly. Note that, under the asynchronous parallel or nondeterministic execution model, the system may converge to different fixed point sets for a given starting state. However, without loss of generality, we will assume that only one fixed point set exists for any given starting state.

The goal of the system can be expressed by a predicate $G(\cdot)$, i.e., $G(s)$ is true iff s is a desired goal state. A goal state is also referred to as a legitimate state [Dij74]. The partial correctness of a program means that it should satisfy $\{s \mid G(s)\} \subseteq FP(P)$ (i.e., it should be in the correct state upon termination) The termination of the program requires the program to reach the fixed point set within a finite time.

III. Self-Stabilizing Systems

In this section we first define stability and characterize self-stabilization. Then we present an application of self-stabilization to the detection of faulty processors in a Byzantine agreement situation.

3.1. Stability of a System

Stability is an intuitively desirable attribute for any system. However, it must be built into the system since there do exist systems that are not stable. Here, we define several classes of systems with respect to their stability. We consider systems which are stable or multi-stable as well as those that are semi-stable, or unstable.

Stable system. A system (P, X, S) is a stable system iff

$\forall s \in S$, we have $fp(P, s) \neq \varnothing$ and $\forall x \in fp(P, s) :: G(x)$, i.e. system goal can be achieved irrespective of the current state.

Note that for a system to be stable requires not only that it should reach certain fixed point sets irrespective of the starting position, but also that the fixed point it reaches should be a legitimate state.

Multiple fixed point sets. A system has multiple fixed point sets iff

$\forall s \in S :: fp(P, s) \neq \varnothing \wedge (\exists s_1, s_2 \in S :: fp(P, s_1) \cap fp(P, s_2) = \varnothing)$.

Multi-stable system. A system is a multi-stable system iff it has multiple fixed point sets and it is a stable system.

The set of multi-stable systems is a subset of the set of stable systems, and it contains more than one disjoint fixed point sets.

Semi-stable system. A system is semi-stable iff it has multiple fixed point sets and $\exists s, s' \in S :: (\exists x \in fp(P, s), x' \in fp(P, s') :: \neg G(x) \wedge G(x'))$.

This indicates that the system has at least one *saddle point* which could make the system converge to an undesired fixed point and at least one desirable fixed point.

Generally, it is not interesting to have a system which satisfies the predicate $\forall s \in S :: (\forall x \in fp(P, s) :: \neg G(x))$, i.e. no fixed point sets of the system is the goal state.

Unstable system. $\exists s \in S :: fp(P, s) = \varnothing$. In this case, if the system starts from some state s, it will move infinitely far from any fixed point set. Clearly, only an infinite state machine can be unstable in this sense.

3.2. Self-Stabilization Model

Any formal definition of self-stabilization must fully capture the basic characteristics of self-stabilization. It is easier to give examples of systems that are obviously *not* self-stabilizing. For example, we certainly do not wish any statement p_i to have the following code.

$$\textbf{if } \neg G(s) \rightarrow s := s_0 \ [] \ G(s) \rightarrow \textbf{skip end if}$$

where s_0 is a state such that $G(s_0)$ is true.

Assume that state s can be expressed by a set of state variables. Then we can denote state s by $\{v_0, \cdots, v_m\}$ and let $R(p_i)$ denote the read-set of p_i (i.e., state variables that are referenced by p_i) and $W(p_i)$ denote the write-set of p_i (i.e., variables that are modified by p_i). A self-stabilizing system can be defined as a system which is a "stable system" and has the characteristics listed in the following.

1. *Local reference.* $\forall i, |R(p_i)| \ll m \wedge |\{j \mid R(p_i) \cap R(p_j) = \varnothing\}| \approx n_p$.
2. *Local effect.* $\forall i, |W(p_i)| \ll m \wedge |\{j \mid W(p_i) \cap W(p_j) = \varnothing\}| \approx n_p$.
3. *Local view.* $\forall i :: |\{s \mid \exists \, function \ f :: f(R(p_i)) \equiv G(s)\}| \ll$ number of possible states.
4. *Equivalent power.* The sets of transitions caused by different statements are similar.

These attributes collectively measure the degree of decentralization of a self-stabilizing system. (Attributes for distributed systems are presented by Jensen [Jen81].) Or, in another sense, they measure the degree to which a system is self-stabilizing. The ideal system would have (1) disjoint read-sets, (2) disjoint write-sets, (3) no state in which a p_i can infer whether G is true or not, and (4) identical statements. At the other extreme, we have a system where a statement has a global view and causes global changes in the state space.

There are several advantages in using self-stabilization. A self-stabilizing system is very efficient under normal operating situations. It can use datagram protocols and main

memory data structures since it can implicitly tolerate transient failures. Further, due to the local nature of each statement, it imposes less communication load on the system. Also, self-stabilizing programs are usually compact since there are no auxiliary mechanisms for failure detection and recovery.

On the other hand, it is harder to discover and prove self-stabilizing programs. Also, it has a "vulnerable" period during which it cannot meet the system requirements. Consequently, this approach is viable for applications that will not suffer extreme adverse consequences should the system fail for a finite time.

3.3. Applications of Self-Stabilization

It may not be possible to have self-stabilizing algorithms for all problems. However, there are a lot of problems for which there are elegant self-stabilizing solutions which result in simple and reasonably efficient code. We have developed self-stabilizing algorithms for several common problems, e.g. robot path planning [Bas85], two-variable inequality problem [Yen85], load balancing in a distributed systems [Kam87], and detection of faulty processors in a variation of the Byzantine agreement problem [Zha87].

Here, we will give a self-stabilizing "mostly Byzantine" agreement algorithm [Zha87] to illustrate the applicability of the self-stablization method. The Byzantine agreement problem addresses methods of reaching agreement in the presence of arbitrary processor failures in a distributed computing environment [Lam82] [Pea80]. Consider a set of processors $p_1, p_2, \ldots, p_n$, some of which may be faulty, that communicate by directly sending messages to one another. Each processor p_i has an initial value v_i. The goal is that, after a number of message exchanges, each processor p_i has to decide on a value d_i, such that the following conditions are satisfied:

(1) **Agreement:** Every reliable processor should decide on the same value, i.e., $d_i = d_j$ for all reliable processors p_i and p_j.

(2) **Validity:** If all the reliable processors have the same initial value v, then for every

reliable processor p_i, $d_i=v$.

Though it seems deceptively simple, this problem is intricately difficult if only oral communication (which is the primary message passing mechanism used by most Byzantine algorithms as well as by the "mostly Byzantine" algorithm in this section) is used. It is shown that in case of t faulty processors, at least $3t+1$ processors are needed [Lam82] [Pea80] to work for at least $t+1$ round of message exchanges to assure Byzantine agreement [Fis82] with no fewer than $\Omega(nt)$ messages [Dol85]. Algorithms that attain the lower bound of $t+1$ rounds use a number of message bits exponential in t [Dol82] [Lam82] [Pea80]. The best polynomial message algorithm so far is given by Srikanth and Toueg which uses $2t+1$ rounds and $O(nt^2\log n)$ message bits [Sri87]. Coan presented a canonical form of algorithm which approaches the lower bounds of rounds by a factor of $1+\varepsilon$, which can be arbitrarily close to 1, and uses $O(tn^{\lceil 2/\varepsilon \rceil +3})$ message bits [Coa86].

The mostly Byzantine agreement relaxes the requirement of reaching agreement once by trying to reach agreement many (potentially infinite) times while requiring agreement as an overall property [Zha87]. For example, if in the original Byzantine agreement problem, the Byzantine generals have to decide whether to attack or retreat in a battle, then in the mostly Byzantine agreement situation, the generals have to repeatly decide whether to attack or retreat in a long sequence of battles, as long as almost all the time the loyal generals make the same decisions. Such a kind of repeated attempts at reaching agreement may exist in some process control tasks. The requirement of the mostly Byzantine agreement can be stated as follows:

(1) **Validity:** If all the reliable processors have the same value at the start of an interval, then all the reliable processors should agree on that value at the end of that interval.

(2) **Finite Disagreement:** The number of times that reliable processors disagree is at most C, a constant depending only on the number of processors and the number of

faulty processors.

This requirement naturally suggests using a self-stablizing algorithm for this problem. The goal in this problem is to reach agreement in such a way that

$$\lim_{N\to\infty} \frac{\#\ disagreements\ in\ N\ tries}{N} = 0.$$

Self-stablization is achieved by starting with a fully connected network and each reliable processor doing a stabilizing step which consists of detecting faulty processors and disconnecting itself from those processors that it deems as being faulty. The algorithms given here assure that whenever the system is in an illegal state, i.e., whenever the reliable processors decide on different values, the stablizing step guarantees that in the next try the system will have a smaller probability of being in an illegal state. The system is self-stabilizing in the sense that the state of the system monotonically converges to the legitimate state, there is no decentralized control, and, although each processor can directly communicate with every other, the state information thus obtained may be corrupted due to the arbitrary behavior of faulty processors, so that it has only a local view of the system state. This approach shares some insight with pseudo-self-stabilization [Bur89b] in that the system is allowed to deviate from the legitimate state once it has entered the legitimate state from an illegal state as long as a stabilizing mechanism works to force the system back to its legitimate state. They differ in that in our approach the probability of deviating from legitimate state will become smaller after each deviation and eventually converge to zero, while pseudo-self-stabilization does not enforce this property.

processor p :

$traitor\_set_p := \emptyset$
repeat at time $T, 2T, \cdots$

round 1
$broadcast(v_p)$
$receive(v_*)$
if p cannot receive p_i's message $\rightarrow$

```
            traitor_set_p := traitor_set_p ∪ {i}
        end if
        V_p := (v_1, ··· ,v_n)

    round 2
        broadcast(V_p)
        receive(V_*)
        for i := 1 to n do
            d_i := select_majority {V_j(i) | 1≤j≤n ∧ j≠i}
            S_p(i) := {j | V_p(i)≠V_j(i)}
            if |S_p(i)|≥t+1 → traitor_set_p := traitor_set_p ∪ {i} end if
            decision_p := majority(d_1, ··· ,d_n)

forever

select_majority(v_1,v_2, ··· ,v_{n-1})≡
    for i := 1 to n-1 do
        if v_i="faulty" → v_i := DEFAULT end if
    if ∃v ::v ∈ {v_1, ··· ,v_{n-1}} ∧ count(v)≥⌈n/2⌉ → v []
        ∄v ::v ∈ {v_1, ··· ,v_{n-1}} ∧ count(v)≥⌈n/2⌉ → DEFAULT
    end if
```

Algorithm 1.

The algorithm given above works synchronously. At every time interval $T, 2T, \ldots$, the processors in the system try to reach agreement by using two rounds of message exchanges: (1) each processor sends out its own value, (2) each processor sends out all the values it receives during the first round. Based on the values it receives, each processor can determine whether a processor is faulty, in which case it disconnects itself from that processor and discards the value sent by that processor. A total of $O(n^3)$ message bits are exchanged in each attempt. The following theorem is proved in [Zha87].

Theorem. Algorithm 1 assures Mostly Byzantine agreement, if $n \geq 4t$.

This algorithm converges very fast in the sense that each faulty processor can only cause at most one disagreement, so that the system will be in the illegal state at most t times in the presence of t faulty processors. We can also get another algorithm with a smaller redundancy but with a slower convergence speed by replacing the *select_majority*

function with the one given below.

processor p :

select_majority $(v_1, v_2, \cdots, v_{n-1}) \equiv$
  $F := \{v_i \mid v_i = \text{“}faulty\text{”}\}$
  $V := \{v_1, \cdots, v_{n-1}\} - F$
  $f := |F|$
  if $f \geq 2t+1 \rightarrow DEFAULT$ []
    $f < 2t+1 \wedge \exists v :: v \in V \wedge count(v) \geq \lceil (n-f)/2 \rceil \rightarrow v$ []
    $f < 2t+1 \wedge \nexists v :: v \in V \wedge count(v) \geq \lceil (n-f)/2 \rceil \rightarrow DEFAULT$
  end if

Algorithm 2.

Theorem. Algorithm 2 assures the Mostly Byzantine agreement if $n \geq 3t+1$ [Zha87].

For $t > 8$, each processor can cause at most $t/2$ disagreements and when $n \geq 4t$, this algorithm has the same convergence speed as the first one.

IV. Inherent fault tolerance (IFT)

Self-stabilization has a major purpose of tolerating faults, i.e. any transient state failure can be inherently tolerated by the program. A notion related to this ability would be tolerance of permanent failure to execute some statements. As with the self-stabilization approach, we want the fault tolerance ability to be embedded in the program. This yields the notion of inherently fault tolerant programs. We will give a formal model of inherent fault tolerance in the following.

4.1. IFT Model

Here, we will use the notations introduced in the system model. Also, let $T(P, s)$ denote the time required to reach any state in $fp(P, s)$ by starting from a state s. The execution model of the program is that each statement p_i in program P is assigned to a processor and processors run in parallel either asynchronously or synchronously corresponding to the execution sequencing of P.

f -fault tolerance: A program $P=\{p_i \mid 1 \le i \le n_p\}$ is f -fault tolerant, where $0 \le f \le 1$, iff $\forall P_f, P_f \subseteq P$ and $|P_f| \ge (1-f) \times n_p$, we have $\forall s :: fp(P_f, s) \in FP(P)$.

Remark. Chandy and Misra have modeled Byzantine failures by removing *all* equations involving variables in the write-set of malicious processors from the **always** section of a Unity program [Cha88] [Gou89]. The removal of statements in the definition of f -fault tolerance corresponds to a *fail-stop* behavior. It is particularly interesting that Unity provides a common framework for modeling these different types of processor failures. □

IFT program. A program P is inherently fault tolerant if it satisfies the following:

(1) *Implicit fault tolerance:* $\exists p$, $p \in P$, such that p has fault detection and/or error recovery functionality;

(2) *Fault coverage:* P is f -fault tolerant, and $f > 0$;

(3) *Performance constraint:* $\forall P_1, P_2, P_1 \subset P_2 \subseteq P$, we have $\forall s, T(P_1, s) > T(P_2, s)$ if $T(P_2, s) < \infty$.

The definition implies certain desirable properties, such as the absence of explicit failure detection or recovery computations which, in turn, eliminates potential communication overhead; no performance penalty or excessively redundant hardware cost since the performance constraint requires that extra processing power should be used to increase the performance during fault free time. However, the definition still leaves open a wide range for the quality of an IFT program. For example, the robustness of an IFT program is determined mainly by its f value while its performance is affected by $T(P_1, s) / T(P_2, s)$ relative to $|P_1| / |P_2|$ for all $P_1 \subset P_2 \subseteq P$.

4.2. Some Examples of IFT Programs

Dijkstra's cyclic relaxation algorithm [Dij73] is an example of inherently fault tolerant programs. A simplified version of the problem is as follows. A set of points are scattered around the circumference of a circle of radius $1/(2\pi)$. Each point can independently make a sequence of changes in its position, at all times knowing only the distances

from its nearest clockwise neighbor and anticlockwise neighbor. What moves should each point make so that eventually all the points are equidistant from both their nearest neighbors?

Let the points be 0, 1, $\cdots$, $n-1$, so that point i is between points $i \ominus 1$ and $i \oplus 1$, where $\ominus$ and $\oplus$ are subtraction and addition operations modulo n, respectively. Let the clockwise distance along the circumference between neighboring points i and $i \oplus 1$ be x_i. Then, we have $\sum_{i=0}^{n-1} x_i = 1$. The algorithm proposed by Dijkstra is

for process p_i :

$$x_{i \ominus 1} := x_i := \frac{(x_{i \ominus 1} + x_i)}{2}$$

If any p_i is removed, the program will still be able to achieve the goal because there is a sufficient overlap in the responsibility of processors in achieving the system goal in a self-stabilizing manner.

Other examples of inherently fault tolerant programs are a two dimensional robot control program [Bas88] and a robust sorting network [Rud85].

V. Theoretical Aspects of Self-Stablization and IFT Programs

Two theoretical aspects which are rather important in any system model are correctness proof and existence properties. Program correctness proof includes termination proof and proof of partial correctness. The termination proof of an asynchronous program is especially difficult since statements can be executed at any time. We will discuss some techniques which help with this task in section 5.1. Also, whether a solution in the system model exists for each problem is a fairly fundamental issue. The requirement of decentralized flavor in self-stabilization and IFT control programs results in the possibility of nonexistence of solutions for some problems. We will address this existence issue in section 5.2.

5.1. Termination Proof of Asynchronous Programs

The major property of an asynchronous program is that the statements of the program can be executed at any time instance. This property increases the fault tolerance ability of the program since transient processor failures (i.e., failure to execute a statement for a finite duration) can be implicitly tolerated. However, the disadvantage of this property is the difficulty of proving the correctness of the program. The correctness proof of a program with nondeterministic or synchronous parallel execution can be done much more systematically due to execution constraints that allow it to be decomposed into fixed execution steps. Nondeterminism actually inherits the same fault tolerance advantages stated above, and it has some well-developed proof techniques [Man80] [Cha88]. Nevertheless, nondeterministic execution is not a desirable model for parallel programming environment due to synchronization overhead and one-at-a-time execution semantics. Moreover, the performance constraint of the IFT model requires performance improvement when extra processing units are provided. Thus, in order to take advantage of the simpler program proof in the nondeterministic model, we can use them only as a vehicle to design and prove algorithms while requiring the algorithm to be executed asynchronously and in parallel. To achieve this, we need to develop a model to identify and prove the correctness of rules for transforming nondeterministic computations to equivalent asynchronous parallel computations. A step in this direction is the following theorem which states the conditions required for the correctness of an IFT program to be preserved under such a transformation. Here, s denotes the current program state, $V_i(s)$ denotes the rate of change in the state due to the ith statement, and $V(s)$ denotes the resultant rate of change in the state.

Linearity condition: $V(s) = \sum_{i=1}^{n} \alpha_i(s) V_i(s)$, $\sum_{i=1}^{n} \alpha_i(s) > 0$.

Theorem. [Bas88] For a program satisfying the following conditions:

(1) the linearity condition

(2) $\forall i : 1 \le i \le n :: V_i$ is continuous, and

(3) $\forall i : 1 \leq i \leq n ::$ either

(a) $V_i(s) = \sum_{j \neq i} \beta_j(s) V_j(s)$, $\beta_j(s) \geq 0$, or

(b) $(\forall r :: V_i(r) = \sum_{j=1}^{n} \beta_j(r) V_j(s))$,

continuous nondeterministic termination $\Rightarrow$ continuous asynchronous parallel termination.

The above theorem is useful for proving the termination of asynchronous continuous parallel programs which is generally the characteristic of process-control systems, e.g. temperature, pressure, and rate of flow of reactant through a pipe, speed of robot, etc.

The continuous version of the cyclic relaxation algorithm introduced in section 4.2 converges even when the points move in parallel. To see this let us consider each of the conditions one by one. The state space is n-dimensional, with $x_0, \cdots, x_{n-1}$ being the n dimensions. Assume that the rate of change induced in the state by point p_i is V_i.

(1) The movement of each point has components along two dimensions and each dimension has components from only two points. Consider dimension x_i. Let $V_{i \ominus 1,i}$ and $V_{i,i}$ be the rates at which points $p_{i \ominus 1}$ and p_i affect x_i, with a positive value meaning that x_i is being increased and *vice versa*. Clearly, the resultant rate of change in x_i is $V_{i \ominus 1,i} + V_{i,i}$, so that the linearity condition is satisfied.

(2) Each V_i must be continuous in the state space. One possibility is to choose $V_{i,i}$ to be proportional to $(x_i - x_{i \ominus 1})$ and $V_{i,i \ominus 1}$ to be proportional to $(x_{i \ominus 1} - x_i)$.

(3) If V_i has magnitude 0, then it satisfies condition 3(a). If it has a nonzero magnitude, then it must have a nonzero magnitude along both $x_{i \ominus 1}$ and x_i. Since it can have components only along these two dimensions, therefore it satisfies condition 3(b).

Finally, it is possible to prove that the cyclic relaxation algorithm converges under continuous nondeterministic execution. (A proof of nondeterministic convergence is given in [Bas88].) Hence, from the above theorem we can conclude that the continuous version of the algorithm converges even under asynchronous parallel execution.

To see more clearly what the three conditions in the theorem mean, let us consider examples where these conditions are violated one at a time and where there is no parallel termination.

Linearity condition. Most physical systems satisfy this condition. However, consider the system shown in Figure 1. There are two iron bars whose rightmost and leftmost tips are labeled A and B, respectively. The goal of the system G is to make $x_A = x_B$, where x_A is the x-coordinate of point A and x_B that of point B. There are two actuators, namely, switches a and b, which can allow a current to pass through the coil around blocks A and B, respectively. If only one of the switches is turned on, then it induces a magnetic force which will attract the other bar. Hence, nondeterministic control can achieve the goal. But, if both the switches are turned on, then the bars have opposite polarities and hence repel each other, so that parallel actions fail to achieve the goal.

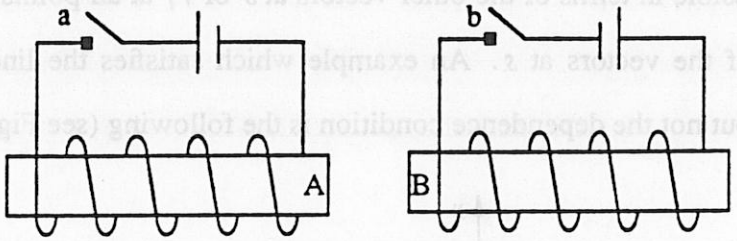


Figure 1. Violation of Linearity Constraint.

Continuity condition. Consider three points A, B, and C with an X on each of A and B (see Figure 2). It is required to reach a state in which there is an X on C and an X on either A or B. A nondeterministic program can be:

```
point A:
   if there is an X on B → move my X to C []
      there is no X on B → skip
   end if

point B:
   if there is an X on A → move my X to C []
      there is no X on A → skip
   end if
```

This has continuous nondeterministic termination since the moment either A or B moves, the other point will not move. However, the goal may not be achieved if the statements are executed in parallel since both the points may move toward C simultaneously.

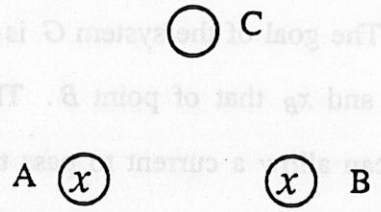


Figure 2. Violation of the Continuity Constraint.

Dependence condition. This condition means that the system will terminate starting from a state s provided that the velocity vectors are related. Specifically, each $V_i(s)$ should either be expressible in terms of the other vectors at s or V_i at all points should be expressible in terms of the vectors at s. An example which satisfies the linearity and continuity conditions but not the dependence condition is the following (see Figure 3).

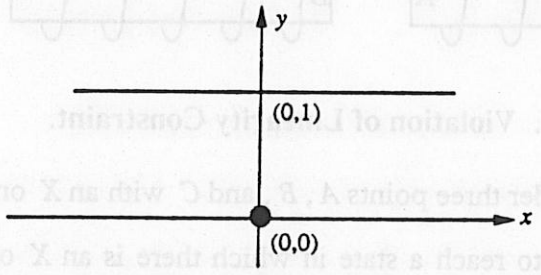


Figure 3. Violation of the Dependence Condition.

Two processes p_1 and p_2 control a point with coordinates (x,y). The goal of the program is to make $y \geq 1$. The velocity vectors are

$$V_1(x,y) = \mathbf{i} + (|x| + |y|)\mathbf{j}$$
$$V_2(x,y) = -\mathbf{i} + (|x| + |y|)\mathbf{j}$$

Suppose the point is located at (0,0). Then, continuous nondeterministic control will achieve the goal since any finite shift from the origin will cause an ever increasing component along the y-coordinate. However, the system does not terminate under parallel

control since $V_1(0,0)+V_2(0,0) = 0$. Condition 3(a) is not satisfied since $V_1(0,0) \neq \beta_2(0,0)V_2(0,0)$ for some $\beta_2(0,0) \geq 0$. Similarly, condition 3(b) is not satisfied since velocities at all other points have a nonzero y component, while the vectors at (0,0) have a zero y component.

If a problem cannot have a solution in the nondeterministic model which satisfies the properties required to guarantee the correctness under parallel execution, then we need to have other analysis to help prove the correctness of parallel execution. Possible problems in transforming from nondeterministic computation to parallel computation are oscillation, duplication, saddle points, etc. Some approaches such as randomized algorithms [Rab76] and simulated annealing [Lar87], can be used to resolve these problems.

5.2. Existence of Decentralized Solution

Both self-stabilization as well as inherent fault tolerance require a certain local decision making property for each statement. Consequently, a basic question for any given problem is whether there exists a decentralized solution for it. In this section we present one condition which guarantees the existence of a decentralized algorithm for process control systems having continuous state spaces. Let us give the problem model in the following.

As we defined in section 2, a system consists of three entities (P, X, S). Here, we will have some extra entities -- a set of actuators A and a set of sensors B, i.e. the system will be expressed by (P, X, S, A, B). The sensors provide the state information of the system, i.e., $\forall b \in B, b: S \rightarrow R^{|s|}$, and the actuators are devices which cause the actual state transitions, i.e. $\forall a \in A, a: S \times C_a \rightarrow S$, where C_a is the set of commands which can be issued to the actuator a. Each statement $p_i \in P$, where $p_i: S \rightarrow S$, will get the information about the state space S from the sensors and cause state transitions through the actuators. Let b_i denote the set of sensors which p_i uses to access information and a_i denote the set of actuators which p_i uses to cause state transitions. Here, we require

$a_i \cap a_j = \emptyset$ for $i \neq j$ in order to ensure that no conflicting commands can be issued to an actuator by different statements. Requiring a fixed set of actuators constrains the system decomposition and simplifies the definition and proof of existence of the decentralized solution. Simple but tedious generalization can be derived from the results obtained in the constrained system to the system which allows decomposition into arbitrary sets of actuators.

A centralized solution to a problem with a fixed set of actuators is to have only one statement which gets state information from all sensors and determines the actions of all actuators. On the other hand, a decentralized solution requires each p_i to determine independently the actions of the set of actuators a_i it controls. At least two statements should exist in the system.

In the following, we will define two properties for a system state space, namely, free space system and convex space system.

Reachable state space. A state space s_R is the reachable state space of system (P, X, S, A, B) starting from state s iff

(1) $s \in s_R$, and

(2) $\forall x \in s_R, a \in A, c \in C_a :: a(x,c) \in s_R$.

The reachable state space from state s are the states which are reachable by simply issuing any allowed commands to any actuator. Note that some states which do not satisfy the system state space constraints can still be in the reachable state space.

Free space. A system with state space S is a free space system iff $\forall s \in S$, we have $s_R = S$.

Traversable trajectory. A trajectory t consisting of a sequence of states $<s_0, s_1, \cdots, s_n>$, $s_i \in S$, is traversable iff

$\forall i : 0 \leq i < n :: (\exists (a_i, c_i) : a_i \in A \wedge c_i \in C_a :: a_i(s_i, c_i) = s_{i+1})$.

In other words, a traversable trajectory is any trajectory in the state space which can be traversed by the actuators.

Approaching trajectory. A trajectory t consisting of a sequence of states $<s_0, s_1, \cdots, s_n>, s_i \in S$, is an approaching trajectory from s_0 to s_n iff $\forall s_i, s_{i+1} \in t$, we have $D(s_i, t) > D(s_{i+1}, t)$, where $D(x, y)$ denotes the distance between states x and y.

Convex space. A system space S is convex iff $\forall s_i, s_j \in S$, the shortest trajectory between s_i and s_j is a traversable trajectory.

The following theorem about the existence of decentralized algorithm can be proven.

Theorem. A decentralized nondeterministic algorithm exists for a problem if it is a continuous system with a convex and free system space, and it satisfies the linearity condition.

The proof of this theorem is quite straightforward, and we will omit it here. Actually, the theorem will hold even if the convex space requirement is replaced by the following: for any two states in the state space, there exists an approaching trajectory which is traversable by the actuators. However, it is much easier to examine whether a problem satisfies the convex space requirement.

From the above theorem, we can conclude that the cyclic relaxation problem has a decentralized solution. To see this, consider the free space for a three-point problem (Figure 4). The free space consists of all and only points such that $x+y+z = 1$. Clearly, any path in this state space can be traversed by the three actuators. Hence, this problem has a decentralized nondeterministic solution. Another problem with a convex free space is the two dimensional robot with three degrees of freedom shown in Figure 5(a). A decentralized IFT control algorithm for this mechanism with the goal of positioning T on P appears in [Bas88]. The introduction of a single point obstacle O (Figure 5(b)) causes the free space to be non-convex. Whether a decentralized solution exists for this case is an open problem.

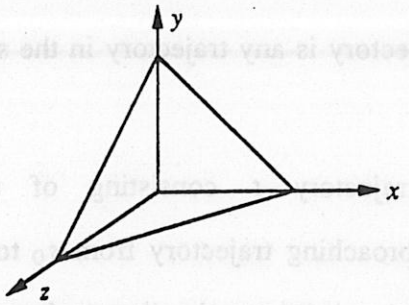


Figure 4. Free Space for Three-Point Problem.

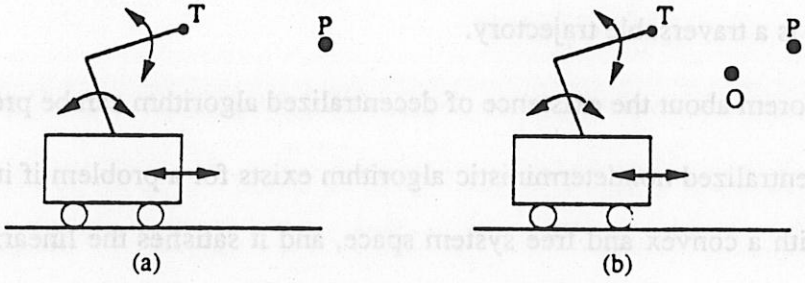


Figure 5. A Robot Control Problem.

5.3. Existence Issues with Conditional Subgoals

In order to achieve the system goal G, each p_i in P will act independently to reach its own local subgoal. These subgoals can be denoted by $g_1, g_2, \cdots, g_n$ where each g_i corresponds to p_i. As with the system goal, G, each g_i is a Boolean function of the state of the system, i.e., if s is the goal state of p_i, then $g_i(s)$ holds. These subgoals should satisfy $G = g_1 \wedge g_2 \wedge \cdots \wedge g_n$. Depending on the constraints on the predicate of the subgoals, these may be either conditional or unconditional subgoals. For a conditional subgoal, the processor tries to satisfy different predicates depending on the value of certain state variables,

g_i = **if** *condition*$\rightarrow g_{i1}$ [] $\neg$*condition*$\rightarrow g_{i2}$ **end if**,

where *condition*, g_{i1}, g_{i2} are atomic propositions (i.e., a Boolean function without any Boolean connectives). If we constrain the subgoal of each statement to be an atomic proposition, i.e., $\forall i$, g_i is an atomic proposition, then we will call the subgoal an unconditional subgoal.

The use of conditional subgoals, increases the set of problems amenable to decentralized decomposition. Thus, it is helpful to consider conditional subgoal decomposition when an unconditional subgoal decomposition is not possible. Here, we give an example which does not satisfy the convex free space constraint, but has a decentralized solution using conditional subgoals.

Consider the system shown in Figure 6. The goal is to turn the vehicle BA into pathway P_1P_2. The vehicle can move along line BA, rotate point B around point A, and rotate point A around point B. We assume that the state shown in Figure 5 is the last stage in a sequential decomposition for preparing to turn so that point A enters pathway P_1P_2 before point B. The turn can be made only if the length of the vehicle is less than $\sqrt{(x+d)^2+c^2(1+d/x)^2}$, where $x=(c^2d)^{1/3}$, c = length of P_1P_2, and d = length of P_2P_3.

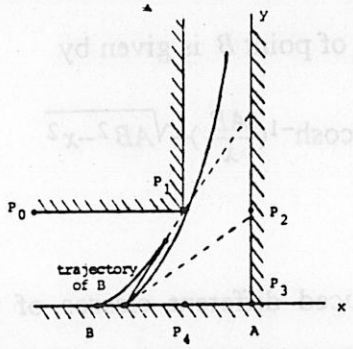


Figure 6. A Turning Problem.

The conditional subgoals for all controllers are given in the following.

L_{BA}:
  if $P_1 \notin AB \rightarrow X_A = X_{P_2}$ []
    $P_1 \in AB \rightarrow Y_B = Y_{P_3}$; **await**$(X_A = X_{P_2})$
  end if

R_B:
  if $P_1 \notin AB \wedge X_A = X_{P_2} \rightarrow Y_A \geq Y_{P_1}$ []
    $P_1 \in AB \wedge Y_A > Y_B \rightarrow$ **await**$(Y_B = Y_{P_3})$; $X_A = X_{P_2}$ []
    $P_1 \in AB \wedge Y_A = Y_B \rightarrow Y_A \geq Y_{P_1}$
  end if

R_A:
  if $A \in P_2P_3 \rightarrow Y_B = Y_{P_3}$ []
    $A \notin P_2P_3 \rightarrow Y_B = Y_A$

end if

This algorithm can tolerate the failure of either the linear motion or the rotational motion around point A. There are three cases:

(i) If all the three motions are available, then the trajectory of point A is along line P_2P_3 while that of point B is along line P_3P_4. This is a relatively fast way of making the turn.

(ii) If the linear motion is not available then the turn can be made only if length $AB <$ length P_1P_2. In this case, the controller for the rotational speed around point A first moves B till $Y_B=Y_A$. Then, the two rotational controllers move the vehicle sideways through pathway P_1P_2.

(iii) If the rotational motion around point A is not available then the trajectory of point A is along line P_2P_3 while that of point B is given by

$$y = AB\cosh^{-1}(\frac{AB}{x})-\sqrt{AB^2-x^2}$$

VI. Conclusion

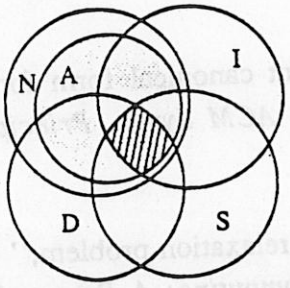
In this paper, we have introduced different classes of programs. Based on the required properties of the program fixed point relative to the program statements, we have introduced self-stabilizing programs and inherently fault tolerant programs. Self-stabilizing programs have the property of reaching the goal state irrespective of the current state, which implies tolerance of transient failures. In the Unity model [Cha88], this corresponds to a program without an **initially** section. Inherently fault tolerant programs have the characteristic of reaching the goal state in spite of the removal of certain fraction of the statements. This property implies tolerance of fail-stop permanent failures.

Both self-stabilizing and inherently fault tolerant programs can be established under nondeterministic, asynchronous parallel, or synchronized parallel models. We have also introduced the pros and cons of programs in these models. Asynchronous parallel and

nondeterministic models have the property that the statements of the program have no precedence relations, which implies several desired properties such as automatic tolerance of transient failures. On the other hand, any parallel model has the advantage of enhanced performance under a parallel environment. Thus, asynchronous parallel is the most desirable programming model.

As we discussed in section 4, a property that both the self-stabilizing and inherently fault tolerant programs should have is decentralization. Limited state references and independent decision making eliminates the communication overhead which is important in parallel systems, and generally, the property also implies a simpler software which facilitates software reliability. We have discussed the existence conditions for decentralized self-stabilization and IFT solutions in the paper.

From the discussion given in this paper about these different properties of programs, we can use a diagram to display the relationship between them. The diagram in Figure 7 shows different sets of programs with different properties.



A = the set of programs which can reach the goal by having their statements executed asynchronously
N = the set of programs which can reach the goal by having their statements executed nondeterministically
S = self-stabilizing programs
I = inherently fault tolerant programs
D = decentralized programs

Figure 7. Classes of Programs

As we have discussed, all the properties displayed by the classes of programs given above are desirable. Thus, an ideal program in a parallel environment would be a program that satisfies the notion of self-stabilization and inherent fault tolerance and its

statements can be executed asynchronously and it is decentralized.

VII. References

[Bas85] F.B. Bastani and I.L. Yen, "Analysis of an inherently fault tolerant program," *Proc. COMPSAC '85*, Chicago, Oct. 1985.

[Bas88] F.B. Bastani, I.-L. Yen, and I.-R. Chen, "A class of inherently fault tolerant distributed programs," *IEEE Trans. Softw. Eng.*,, Vol. SE-14, No. 10, Oct. 1988.

[Bro89] G.M. Brown, M.G. Gouda, and C.-L. Wu, "Token systems that self-stabilize," *IEEE Trans. Comp.*, Vol. 38, No. 6, June 1989, pp. 845-852.

[Bur89a] J.E. Burns and J. Pachl, "Uniform self-stabilizing ring," *ACM TOPLAS*, Vol. 11, No. 2, April 1989, pp. 330-344.

[Bur89b] J.E. Burns, Presentation at *MCC Workshop on Self-Stabilization*, Aug. 1989.

[Cha88] K.M. Chandy, J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[Coa86] B. A. Coan, "A communication-efficient canonical form for fault-tolerant distributed protocols," *Proc. 5th Annu. ACM Symp. Principles of Distributed Comput.*, Aug., 1986, pp. 63-71.

[Dij73] E.W. Dijkstra, "The solution to a cyclic relaxation problem," EWD 386, in E.W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, 1982.

[Dij74] E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Comm. ACM*, Vol. 17, No. 11, Nov. 1974, pp. 643-644.

[Dol82] D. Dolev, "The Byzantine Generals strike again," *Journal of Algorithms*, Vol. 3, No. 1 (Jan., 1982), pp.257-274.

[Dol85] D. Dolev and H. R. Strong, "Bounds on information exchanged for Byzantine agreement," *Journal of ACM*, Vol. 32, No. 1 (Jan., 1985), pp. 191-204.

[Fis82] M.J. Fischer and N.A. Lynch, "A lower bound for the time to assure

interactive consistency," *Info. Proc. Letts.*, Vol. 14, No. 4, June 1982, pp. 183-186.

[Gou89] M.G. Gouda, Private communication, Aug. 1989.

[Hin84] G.E. Hinton, "Parallel computations for controlling an arm," *J. of Motor Behavior*, Vol. 16, No. 2, 1984, pp. 171-194.

[Jen81] E.Jensen, "Distributed control," in B.W. Lampson, M. Paul, and H.J. Siegert, (Eds.), *Distributed Systems - Architecture and Implementation*, Springer-Verlag, New York, 1981, pp. 175-190.

[Kam87] M.K. Kam and F.B. Bastani, "A self-stabilizing ring protocol for load balancing in distributed real-time process control systems," *Tech. Rep. #UH-CS-87-8*, Dep. Comp. Sc., Univ. Houston, Houston, Texas, Nov. 1987.

[Kru79] H.S.M. Kruijer, "Self-stabilization (in spite of distributed control) in tree-structured systems," *Info. Proc. Lets.*, Vol. 8, No. 2, Feb. 1979, pp. 91-95.

[Lam82] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals problem," *ACM Trans. on Programming Language and Systems*, Vol. 4, No. 3 (July, 1982), pp. 382-401.

[Lam83] L. Lamport, "Solved problems, unsolved problems and non-problems in concurrency," *Operating Systems Review*, Vol. 19, No. 4, Oct. 1985, pp. 34-44.

[Laa87] P.J.M. van Laarhoven, E.H.L. Aarts, *Simulated Annealing: Theory and Applications*, D. Reidel, Holland, 1987.

[LeL79] G. LeLann, "An analysis of different approaches to distributed computing," *Proc.* 1st Int. Conf. Dist. Proc. Sys., Huntsville, AL, Oct. 1979, pp. 222-232.

[Man80] Z. Manna, *Lectures on the Logic of Computer Programming*, SIAM, Philadelphia, 1980.

[Pea80] M. Pease, R. Shostak and L. Lamport, "Reaching agreement in the presence of faults," *Journal of ACM*, Vol. 27, No. 2 (Apr., 1980), pp. 228-234.

[Rab76] M. Rabin, "Probabilistic algorithms," In *Algorithms and Complexity: New Directions and Recent Results*, (J. Traub, Ed.), Academic Press, New York, 1976, pp. 21-39.

[Rud85] L. Rudolph, "A robust sorting network," *IEEE Trans. Comp.*, Vol. C-34, No. 4, April 1985, pp. 326-335.

[Sri87] T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Distributed Computing*, Vol. 2, No. 3 (1987), pp. 80-94.

[Sta82] J.A. Stankovic, N. Chowdhury, R. Mirchandaney, I. Sidhu, "An evaluation of the applicability of different mathematical approaches to the analysis of decentralized control algorithms," *Proc. COMPSAC '82*, Chicago, Ill., Nov. 1982, pp. 62-69.

[Ure87] A. Uresin, M. Dubois, "Asynchronous relaxation of nonnumerical data," *Proc. Intl. Conf. Parallel Proc.*, Aug. 1987.

[Yen85] I.-L. Yen, *The Role of Parallel Processing in Application Programs*, M.S. thesis, Dep. Comp. Sc., Univ. Houston, Aug. 1985.

[Zha87] Y. Zhao and F. B. Bastani, "A self-stabilizing algorithm for Byzantine agreement," *Tech. Rep. #UH-CS-87-6*, Dep. Comp. Sc., Univ. Houston, Oct. 1987.