Token Systems that Self-Stabilize

GEOFFREY M. BROWN, MEMBER, IEEE, MOHAMED G. GOUDA, AND CHUAN-LIN WU, SENIOR MEMBER, IEEE

Abstract—**We present a new class of mutual exclusion systems, in which processes circulate one token, and each process enters its critical section when it receives the token. Each system in the class is *self-stabilizing*; i.e. if it starts at any state, possibly one where many tokens exist in the system, it is guaranteed to converge to a good state where exactly one token exists in the system. The systems are better than previous systems in that their state transitions are *noninterfering*; i.e., if any state transition is enabled at any instant, then it will continue to be enabled until it is executed; this makes the systems easier to implement as delay-insensitive circuits.**

Index Terms: **Delay-insensitive circuits, noninterference, self-stabilization, token systems.**

I. Introduction

In 1973, Dijkstra coined the term "self-stabilizing" to distinguish any system that has the following property: if the system starts at any, possibly illegitimate state, it is guaranteed to converge to a legitimate state in a finite time [2]. Self-stabilizing systems are attractive for two reasons. First, they are highly robust in the face of transient errors. Second and most important, the existence of self-stabilizing systems demonstrates that the usual reliance on the initial state of a system to establish correctness is in principle unnecessary.

In this paper, we present a class of self-stabilizing mutual exclusion systems in which processes are expected to circulate exactly one token. Each system is self-stabilizing in the sense that, regardless of the number of tokens that exist initially, the system is guaranteed to reach a state in which only one token exists. Our interest in token systems that are self-stabilizing is twofold. First, token systems have many applications in distributed systems, and they are widely used in the protocols of local area networks. Thus, it is always desirable to make them as fault-tolerant as possible. Second, the traditional method of ensuring fault tolerance of a system, by adding fault detection and recovery procedures to it, seems to result in a complicated system. Designing such a systems with the view that all its states are possible, but not necessarily desirable, initial states can lead to a system that is inherently simple, yet fault tolerant.

The problem of constructing self-stabilizing token systems has been considered by Dijkstra in his original papers on self-stabilization, and he discussed the construction of three such systems. However, in establishing the correctness of each system, Dijkstra assumed the existence of a central demon that repeatedly selects one process at random, then allows it to execute one state transition while freezing the other processes [3], [4]. The need for this central demon results from the fact that the state transitions of neighboring processes are interfering, i.e., an enabled transition in one process can be disabled by the execution of an enabled transition in a neighboring process. The central demon, then, ensures that enabled interfering transitions are executed in sequence and not in parallel. Problems arise if one tries to implement Dijkstra's systems in asynchronous hardware. First, implementing a central demon as a separate sequential machine violates the philosophy of system distribution. Second, although the central demon can be implemented by introducing local mutual exclusion mechanisms that permit a process to execute only when its neighbor processes are disabled from execution, this implementation would require the use of arbiters, which are better avoided. Thus, it is desirable to design self-stabilizing token systems whose correctness is not based upon central demons, i.e., systems whose state transitions are noninterfering in the following sense: if a state transition is enabled, it continues to be enabled regardless of the execution of other transitions, until it is executed.

We present, in this paper, a sequence of three self-stabilizing token systems whose state transitions are noninterfering. The first system is the most succinct. Each subsequent system is a refinement of its predecessor. The last system in the sequence is a self-stabilizing, delay-insensitive circuit [11]. Two other interesting variations of the original system are discussed at the end.

In a related work, Lamport championed the cause of self-stabilization [7], and later presented a class of self-stabilizing mutual exclusion systems [8]. In his systems, each process can communicate directly with every other process; thus, the system topology is a fully-connected graph rather than a simple cycle or a linear chain as in the case of our token system. this makes his problem easier than ours. Kruijer has presented a self-stabilizing token system in which the system topology is a tree [6]; the state transitions of his system are interfering.

The paper is organized as follows. In Sections II and III, we present our first system and its correctness proof, respectively. Then, the second system and its relationship with the first

Manuscript received November 11, 1986; revised July 24, 1987. This work was supported in part by the MCD fund of the University of Texas at Austin, in part by IBM, in part by Dupont, and in part by MCC.

G. M. Brown was with the Department of Electrical and Computer Engineering, the University of Texas at Austin, Austin, TX 78712. He is now with the School of Electrical Engineering, Cornell University, Ithaca, NY 14853.

M. G. Gouda is with the Department of Computer Sciences, the University of Texas at Austin, Austin, TX 78712.

C.-L. Wu is with the Department of Electrical and Computer Engineering, the University of Texas at Austin, Austin, TX 78712.

IEEE Log Number 8927559.

system are discussed in Sections IV and V, respectively. In Section VI, we present a delay-insensitive asynchronous circuit that implements the second system. Finally, in Section VII, we discuss our work and present two interesting variations of the first system.

II. First System

We consider an array of n, $n \geq 2$, communicating processes: the leftmost process is called *bottom*, the rightmost process is called *top*, and the processes in between are identical and each of them is called *middle*. Bottom communicates with its right neighbor, top communicates with its left neighbor, and each middle communicates with both its left and right neighbors. The processes communicate using synchronous, CSP-like primitives [5], i.e., for the communication to take place between two adjacent processes, each process should be ready to communicate with the other.

The processes execute the following programs.

```
bottom  ::  *[   C; R; W]
middle  ::  *[L; C; R; W]
top     ::  *[L; C      ].
```

The operators and commands in these programs can be defined informally as follows:

- $*$ signifies indefinite looping.
- ; signifies sequential composition.
- C is the command to execute the process's critical section. (Executing C is equivalent to having possession of a token.)
- R is a command to communicate with the right neighbor; it causes the executing process to wait until its right neighbor reaches L; then both the executing process and the right neighbor move simultaneously to their next commands.
- W is a waiting command; it causes the executing process to wait until its right neighbor reaches L; then the executing process moves to the next command leaving the right neighbor at L. (Martin [9] was the first to notice the importance of such waiting commands in the CSP framework; he calls them *probes*.)
- L is a command to communicate with the left neighbor; it is similar to R except that it is concerned with the left neighbor instead of the right.

As we show later, this system has a number of interesting properties, including self-stabilization, but before we list and verify these properties, we need to define the system in more formal terms.

The system is defined formally by a finite number of system states and state transitions. A *system state* is any string of symbols that satisfies the regular expression

$$(C_b + R_b + W_b) \cdot (L_m + C_m + R_m + W_m)^{n-2} \cdot (L_t + C_t)$$

where n is an integer greater than 1 (recall that n signifies the number of processes in the system), and the subscripts b, m, and t indicate bottom, middle, and top, respectively. Since each system state is a string of n symbols taken from a finite domain, and since n is fixed, the number of distinct states is finite. The special state $C(L)^{n-1}$ is called the *home state* of the system. (We adopt the convention of omitting symbol subscripts whenever they can be deduced from the context; thus, we write the home state $C(L)^{n-1}$, instead of the more specific $C_b(L_m)^{n-2}L_t$.)

The *state transitions* of the system are defined as follows. (Let $i \in \{b, m\}$ and $j \in \{m, t\}$.)

t_0:	C_i	$\rightarrow$	R_i	(Bottom or middle executes critical section)
t_1:	C_t	$\rightarrow$	L_t	(Top executes critical section)
t_2:	R_iL_j	$\rightarrow$	W_iC_j	(Two adjacent processes communicate)
t_3:	W_bL_j	$\rightarrow$	C_bL_j	(Bottom stops waiting on right neighbor)
t_4:	W_mL_j	$\rightarrow$	L_mL_j	(Middle stops waiting on right neighbor).

A state transition $x \rightarrow y$ is *enabled* at the kth symbol, $1 \leq k \leq n$, of a system state s if and only if there exist substrings u and v such that u has $(k - 1)$ symbols and $s = u \cdot x \cdot v$. Notice that if $u \cdot x \cdot v$ is a system state and $x \rightarrow y$ is a state transition, then $u \cdot y \cdot v$ is a system state; in this case, the state $u \cdot y \cdot v$ is said to *follow* the state $u \cdot x \cdot v$ over the transition $x \rightarrow y$. (At this point, the reader is encouraged to check that these five state transitions faithfully represent the three programs: bottom, middle, and top.)

A system state s' is *reachable from* a system state s if and only if there is a sequence of system states $s_0, s_1, \cdots, s_{r-1}$, such that $s_0 = s$, $s_{r-1} = s'$, and s_{k+1} follows s_k for each k. *Starting from* a state s, *the system will reach* a state s' *within* k, $k \geq 0$, transitions if and only if $s' = s$, or $k \geq 1$ and starting from each state that follows s, the system will reach s' within $(k - 1)$ transitions. *Starting from s, the system will reach* s' if and only if there exists a nonnegative k such that starting from s, the system will reach s' within k transitions.

III. Correctness of First System

In this section, we prove that our system satisfies the properties of: noninterference, liveness, mutual exclusion, progress, and self-stabilization.

Theorem 1 (Noninterference): For every transition t, if t is enabled at the kth symbol of a system state s, and if a system state s' follows s over a transition t' enabled at the k'th symbol of s where $t \neq t'$ or $k \neq k'$, then t is enabled at the kth symbol of s'.

Proof: The proof is considering the five cases: $t = t_0$, $t = t_1$, $t = t_2$, $t = t_3$, and $t = t_4$. If t_0 or t_1 is enabled at the kth symbol of s, then this symbol is C and no other transition can modify it. If t_2 is enabled at the kth symbol of s, then the kth and $(k + 1)$th symbols are R and L, respectively; thus, both these symbols can be modified by t_2 only. If t_3 or t_4 is enabled at the kth symbol of s, then the kth and $(k + 1)$th symbols are W and L, respectively, and no other transition can modify these symbols. □

Theorem 2 (Liveness): At every system state, at least one state transition is enabled.

Proof: Let s be any system state. If s has one or more C symbols, then t_0 or t_1 is enabled at s. If s has no C symbols, then its leftmost symbol is either R or W and its rightmost symbol is L. Thus, s must have two adjacent symbols of the form RL in which case t_2 is enabled at s, or of the form WL in which case either t_3 or t_4 is enabled at s. □

Theorem 3 (Mutual Exclusion): Every system state that is reachable from the home state has at most one C symbol.

Proof: The regular expression $(W)^k \cdot (L + C + R + W) \cdot (L)^{n-k-1}$, for some k, $0 \leq k \leq n - 1$, is an *invariant* that is satisfied by all states that are reachable from the home state. This is because

i) the home state $C(L)^{n-1}$ satisfies this regular expression when $k = 0$, and

ii) for each system state s that satisfies this regular expression and each system transition t, if system state s' follows s over t, then s' satisfies this regular expression.

Since every state that satisfies this regular expression has at most one C symbol, Theorem 3 follows. □

Theorem 4 (Progress): For each k, $1 \leq k \leq n$, starting from the home state the system will reach a state in which the kth symbol is C.

Proof: The theorem follows from the next assertion which can be proved by inspection. For each k, $1 \leq k \leq n - 1$, starting from the state $(W)^{k-1}C(L)^{n-k}$, the system will reach state $(W)^kC(L)^{n-k-1}$ over two transitions, t_0 followed by t_2. □

Theorem 5 (Self-Stabilization): Starting from any state, the system will reach the home state.

Proof: We carry out the proof in three steps.

i) First, we show that starting from any state, the system will reach a state of the form $C \cdot x$, where x is any substring.

ii) Then, we show that starting from any state of the form $C \cdot x$, the system will reach the state $(W)^{n-1}C$.

iii) Finally, we show that starting from the state $(W)^{n-1}C$, the system will reach the home state.

Proof of i: We define a nonnegative rank on the system state s as follows:

rank of s = the integer resulting from replacing the symbols of s by integer digits:

each C is replaced by 3,
each R is replaced by 2,
each W is replaced by 1, and
each L is replaced by 0.

Consider how the transitions t_0, t_1, t_2, and t_4 modify the value of the rank. t_0 replaces one 3 by one 2, and t_1 replaces one 3 by one 0 in the rank; t_2 replaces 20 by 13, and t_4 replaces 10 by 00 in the rank. Because all transitions, except t_3, reduce the value of the rank, and the value of the rank can never become negative, and some transition is always enabled (from Theorem 2), then transition t_3 must eventually be executed. The system state that follows the execution of t_3 must be of the form $C \cdot x$, where x is a substring.

Proof of ii: A state of the form $C \cdot x$ satisfies the regular expression $(W)^j \cdot C \cdot y$, $0 \leq j \leq n - 2$; thus, it is sufficient to show that starting from any state s that satisfies $(W)^j \cdot C \cdot y$, $0 \leq j \leq n - 2$, the system will reach a state that satisfies $(W)^{j+1} \cdot C \cdot z$, $0 \leq j \leq n - 2$. First, transition t_3 cannot be executed at s. If any of the transitions t_1, t_2, and t_4 are executed at s, then the resulting state also satisfies the same regular expression but has a smaller rank value than that of s. Therefore, eventually t_0 is executed leading the system to a state that satisfies the regular expression $(W)^j \cdot R \cdot y$, $0 \leq j \leq n - 2$. Now consider a state s' that satisfies this last regular expression. Transition t_3 cannot be executed at s'. If any of the transitions t_0, t_1, and t_4 are executed at s', then the resulting state also satisfies this same regular expression but has a smaller rank value than that of s'. Therefore, eventually t_2 is executed leading the system to a state that satisfies the regular expression $(W)^{j+1} \cdot C \cdot z$, $0 \leq j \leq n - 2$. □

Proof iii: Starting from the state $(W)^{n-1}C$, the system will reach the home state $C(L)^{n-1}$, over the sequence of transitions t_1, followed by $(n - 2)$ of t_4, followed by t_3. □

IV. Second System

As mentioned earlier, the advantage of the preceding system over Dijkstra's systems [3] lies in the fact that its transitions are noninterfering, which makes it easier to implement in hardware. But, despite its freedom from interference, this system still cannot be easily implemented in hardware. This is because one of its state transitions, namely t_2, requires that two adjacent processes change their local states simultaneously, a requirement that cannot be accomplished using delay-insensitive circuits. Thus, our next step is to modify the system such that each of the transitions of the modified system can only change the local state of one process.

The modification consists of replacing the CSP communication commands, which are responsible for the simultaneous activities of adjacent processes, by appropriate waiting commands in the process programs. The processes of the modified system, henceforth called the Second System, execute the following programs.

```
bottom  ::  *[        D; S; T; X]
middle  ::  *[M; N; D; S; T; X]
top     ::  *[M; N; D           ].
```

The commands in these programs can be defined informally as follows:

D	is the command to execute the process's critical section.
S (or T, or X)	is a waiting command that causes the executing process to wait until its right neighbor reaches the command N (or any command other than N, or the command N, respectively).
M (or N)	is a waiting command that causes the executing process to wait until its left neighbor reaches any command other than T (or the command T, respectively).

Formally, a state of the Second System is a string of symbols that satisfies the regular expression

$$(D_b + S_b + T_b + X_b) \cdot (M_m + N_m + D_m + S_m + T_m + X_m)^{n-2} \cdot (M_t + N_t + D_t).$$

The state transitions of the system are defined as follows. (Let $i \in \{b, m\}$ and $j \in \{m, t\}$.)

$u0$:	D_i	$\rightarrow$	S_i	(Bottom or middle executes critical section)
$u1$:	D_t	$\rightarrow$	M_t	(Top executes critical section)
$u2$:	S_iN_j	$\rightarrow$	T_iN_j	(Bottom or middle at S stops waiting when right neighbor is N)

$u3$:	$T_i(\neg N)$	$\rightarrow$	X_i	(Bottom or middle at T stops waiting when right neighbor is not N)
$u4$:	X_bN_j	$\rightarrow$	D_bN_j	(Bottom at X stops waiting when right neighbor is N)
$u5$:	X_mN_j	$\rightarrow$	M_mN_j	(Middle at X stops waiting when right neighbor is N)
$u6$:	$(\neg T)M_j$	$\rightarrow$	N_j	(Middle or top at M stops waiting when left neighbor is not T)
$u7$:	T_iN_j	$\rightarrow$	T_iD_j	(Middle or top at N stops waiting when left neighbor is T).

We now define the concept of a transition being enabled at the kth symbol of a state s. Since transitions u_0, u_1, u_2, u_4, u_5, and u_7 are similar to the transitions of the First System, our previous definition still applies to them. Transition u_3, $T_i(\neg N) \rightarrow X_i$, is said to be *enabled at* the kth symbol of a state s if the kth symbol of s is T and the $(k + 1)$th symbol of s is any symbol other than N. Similarly, transition u_6, $(\neg T)M_j \rightarrow N_j$, is said to be *enabled at* the kth symbol of a state s if the kth symbol of s is M and the $(k - 1)$th symbol of s is any symbol other than T.

If a transition u of the form $x \rightarrow y$, $x(\cdots) \rightarrow y$, or $(\cdots)x \rightarrow y$ is enabled at a state $a\cdot x\cdot b$, then the state $a\cdot y\cdot b$ is said to *follow* the state $a\cdot x\cdot b$ *over* transition u. This concept of "a system state following another" can be used to define the concepts of "a system state being reachable from another" and "the system reaching a state starting from another" as we did for the First System.

V. Correctness of Second System

In this section, we prove that the Second System satisfies the same five properties that we proved for the First System. However, rather than prove these properties from scratch as we did for the First System (henceforth denoted FS), we prove that the Second System (henceforth denoted SS) is, in some sense, equivalent to FS, and therefore it satisfies the required properties since FS satisfies them. In this way, the relationship between the two systems is made explicit.

First we define a mapping f from the system states of SS to the system states of FS. For each state s of SS, the corresponding state $f(s)$ of FS is constructed from s as follows:

Each D is replaced by C.
Each S is replaced by R.
Each T whose right neighbor is not an N is replaced by W.
Each T whose right neighbor is an N is replaced by R.
Each X is replaced by W.
Each M is replaced by L.
Each N is replaced by L.

We now show that the mapping f satisfies some useful properties.

Lemma 1: The kth symbol of a state s of SS is D if and only if the kth symbol of state $f(s)$ of FS is C.

Proof: The lemma is a direct consequence of mapping f. [ ]

Lemma 2: If s and s' are system states of SS such that s' follows s, then either $f(s) = f(s')$, or $f(s')$ follows $f(s)$ over some state transition of FS.

Proof: The proof is by considering the eight cases: s' follows s over u_k, for $k = 0, \cdots, 7$.

For $k = 0$, $f(s')$ follows $f(s)$ over t_0.
For $k = 1$, $f(s')$ follows $f(s)$ over t_1.
For $k = 2$, $f(s') = f(s)$.
For $k = 3$, $f(s') = f(s)$.
For $k = 4$, $f(s')$ follows $f(s)$ over t_3.
For $k = 5$, $f(s')$ follows $f(s)$ over t_4.
For $k = 6$, $f(s') = f(s)$.
For $k = 7$, $f(s')$ follows $f(s)$ over t_2. □

Lemma 3: Every sequence $s_0, s_1, \cdots$ of the states of SS, where for each k, s_{k+1} follows s_k and $f(s_k) = f(s_{k+1})$, is finite.

Proof: Let $s_0, s_1, \cdots$ be a sequence of the states of SS such that for each k, s_{k+1} follows s_k and $f(s_{k+1}) = f(s_k)$. From the proof of Lemma 2, if s follows over u_0, u_1, u_4, u_5, or u_7, then $f(s_k) \neq f(s_{k+1})$. For every k in the sequence, s_{k+1} follows s_k over u_2, u_3, or u_6. Define the following nonnegative rank for each state s_k in the sequence

$$\text{rank of } s_k = 2*(\#S) + \#T + \#M$$

where $\#S$ (or $\#T$ or $\#M$) is the number of S (or T or M) symbols in s_k. Transition u_2 replaces one S symbol by one T symbol, u_3 replaces one T symbol by one X symbol, and u_6 replaces one M symbol by one N symbol. Hence, each of these three transitions reduces the rank, and the sequence is finite. □

Each state of SS that is mapped by f to the home state of FS satisfies the regular expression: $D \cdot (M + N)^{n-1}$; we call each such state *a home state* of SS. Since SS has more than one home state, the definitions of: mutual exclusion, progress, and self-stabilization for FS need to be modified before they can be used for SS; in particular, each mention of "the" home state should be replaced by "a" home state. The definitions of the two other properties, namely noninterference and liveness, remain unchanged. We are now ready to establish the correctness of SS.

Theorem 6 (Noninterference of SS): For every transition u, if u is enabled at the kth symbol of a system state s of SS, and if a system state s' follows s over a transition u' enabled at the k'th symbol of s where $u \neq u'$ or $k \neq k'$, then u is enabled at the kth symbol of s'.

Proof: The proof is similar to that of Theorem 1. □

Theorem 7 (Liveness of SS): At every system state of SS, at least one state transition is enabled.

Proof: Let s be any system state of SS. If s has at least one D symbol, then u_0 or u_1 is enabled at s. If s has no D symbols, then its leftmost symbol is S, T, or X, and its rightmost symbol is either M or N. Therefore, s has two adjacent symbols of the form

SM (in which case u_6 is enabled),
SN (in which case u_2 is enabled),
TM (in which case u_3 is enabled),
TN (in which case u_7 is enabled),
XM (in which case u_6 is enabled),
XN (in which case u_4 or u_5 is enabled). □

Theorem 8 (Mutual Exclusion of SS): Every system state

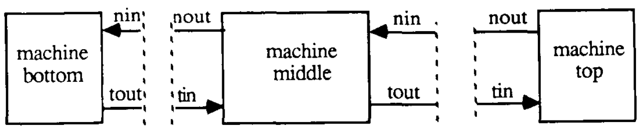

Fig. 1. Implementation.

that is reachable from a home state of SS has at most one D symbol.

Proof: The theorem follows from Theorem 3 and Lemmas 1 and 2. □

Theorem 9 (Progress of SS): Starting from a home state of SS, the system will reach a state in which the kth symbol is D, for each k, $1 \leq k \leq n$.

Proof: The theorem follows from Theorem 4 and Lemmas 1, 2, and 3. □

Theorem 10 (Self-Stabilization of SS): Starting from any state of SS, the system will reach a home state.

Proof: The theorem follows from Theorem 5 and Lemmas 2 and 3. □

VI. Implementation

The Second System can be implemented as a delay-insensitive circuit. In particular, each process can be implemented as an asynchronous sequential machine that has one input line from, and one output line to, each neighboring machine. For convenience, we call the machine that implements process bottom (middle, or top) machine bottom (machine middle, or machine top, respectively). The system configuration is shown in Fig. 1. Notice that each connecting wire has two names, one name is used in discussing the machine from which the wire is output, and the other is used in discussing the machine to which it is input.

We start by defining machine middle; later, we show that machines bottom and top can be constructed by slight modifications to machine middle. The state of process middle has six distinct values: M, N, $\cdots$, or X. Thus, machine middle must have at least three flip-flops to store the current value of its state. We design machine middle to have three flip-flops, named "a", "b," and "c;" thus, its state has eight distinct values. To ensure self-stabilization, each of these values should correspond to a value of the state of process middle. (If there is a value of the state of machine middle that does not correspond to a value of the state of process middle, then starting the machine at this value does not necessarily guarantee that the system will converge to a good state, violating the requirement of self-stabilization.) We choose the mapping from the states of matching middle to those of process middle shown in Table I.

This mapping satisfies the property that any legal change in the value of the state of process middle, i.e., M to N, N to D, $\cdots$, or X to M, can be realized by changing the value of only one flip-flop in machine middle. (This is not the only mapping that satisfies this property, and we could equally well have used any other mapping that satisfies this property.)

Machine middle has two outputs named "nout" and "tout;" their values depend upon the machine's state as shown in Table II.

TABLE I
Correspondence Between the States of Machine Middle and Those of Process Middle

State of Machine Middle (abc)	State of Process Middle
000	M
001	N
011	D
010	D
110	D
111	S
101	T
100	X

TABLE II
Output of Machine Middle

nout = 1	if current state of middle = 001 (which corresponds to state N)
= 0	otherwise
tout = 1	if current state of middle = 101 (which corresponds to state T)
= 0	otherwise

TABLE III
Transitions of Machine Middle

Current State (abc)	Current Input (nin tin)	Next State (abc)
000	x 0	001
001	x 1	011
011	x x	010
010	x x	110
110	x x	111
111	1 x	101
101	0 x	100
100	1 x	000

From the state transitions of the Second System, one can factor out the state transitions of process middle. These can then be translated, taking into account the values of the two inputs nin and tin into state transitions of machine middle. This translation is straightforward with one exception, the transition $D \rightarrow S$ of process middle is translated into three transitions of machine middle:

$$011 \rightarrow 010$$

$$010 \rightarrow 110$$

$$110 \rightarrow 111$$

The resulting state transitions of machine middle are shown in Table III. ("x" denotes a don't care value.)

So far, we have only encoded process middle in a binary form; it remains now to show how to implement this binary form as an asynchronous machine. First, from Tables I and II, we get

$$\text{nout} = a' \cdot b' \cdot c, \text{ and}$$

$$\text{tout} = a \cdot b \cdot c,$$

where "a'" denotes the Boolean negation of "a" and "$\cdot$" denotes the Boolean "AND" operator. Second, by choosing flip-flops "a," "b," and "c," to be SR flip-flops, Table III yields the following equations for the two inputs S_i and $R_i (i = a, b, c)$ of each flip-flop ("+" is the Boolean "OR" operator.)

$$S_a = a' \cdot b \cdot c'$$

$$R_a = a \cdot b' \cdot c' \cdot \text{nin}$$

$$S_b = a' \cdot b' \cdot c \cdot \text{tin}$$

$$R_b = a \cdot b \cdot c \cdot \text{nin}$$

$$S_c = (a' \cdot b' \cdot c' \cdot \text{tin}') + (a \cdot b \cdot c')$$

$$R_c = (a \cdot b' \cdot c \cdot \text{nin}') + (a' \cdot b \cdot c).$$

A schematic for machine middle is shown in Fig. 2; it has the three flip-flops a, b, and c, two combinational circuits y and z, the nine internal wires: k, S_k, and R_k, where $k = a, b$, or c.

We design machines bottom and top to resemble machine middle; in particular, each is assigned three flip-flops, also named "a," "b," and "c" for convenience, to store the current state. Tables IV and V define the correspondence between the states of machines bottom and top and those of processes bottom and top, respectively.

The output and transition tables for bottom and top are identical to Tables II and III, respectively.

From the transition table of machine bottom, the only way bottom can make a transition from state 000 is when tin = 0, and the only way it can make a transition from state 001 is when tin = 1. (Otherwise, the state transitions of bottom are independent of tin.) But in the first case nout = 0 and in the second nout = 1. Therefore, we choose

$$\text{tin} = \text{nout}$$

for machine bottom. Similarly from the transition table of machine top, we choose

$$\text{tin} = \text{tout}'$$

for top. The resulting schematics for machines bottom and top are shown in Fig. 3. This completes our implementation of the self-stabilizing token system as a delay-insensitive circuit.

The above circuit is not self-stabilizing in the absolute sense since it is not guaranteed to converge to a legitimate state starting from any state. For example, if the circuit starts at a state in which the two inputs of a flip-flop are both "1," then the circuit's behavior is indeterminate. That our circuit is not self-stabilizing in the absolute sense should not be surprising; in fact, by extending the above argument, it can be shown that no circuit, synchronous or asynchronous, is self-stabilizing in the absolute sense. Therefore, we have to make some assumption concerning the initial state of our circuit. Moreover, to ensure that at no instant during the circuit's execution the two inputs of a flip-flop are both "1," we need to make an

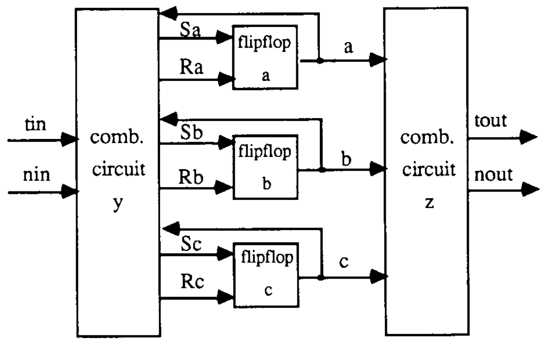

Fig. 2. Schematic of machine middle.

TABLE IV
CORRESPONDENCE BETWEEN THE STATES OF MACHINE BOTTOM AND THOSE OF PROCESS BOTTOM

State of Machine Bottom (abc)	State of Process Bottom
000	D
001	D
011	D
010	D
110	D
111	S
101	T
100	X

TABLE V
CORRESPONDENCE BETWEEN THE STATES OF MACHINE TOP AND THOSE OF PROCESS TOP

State of Machine Top (abc)	State of Process Top
000	M
001	N
011	D
010	D
110	D
111	D
101	D
100	D

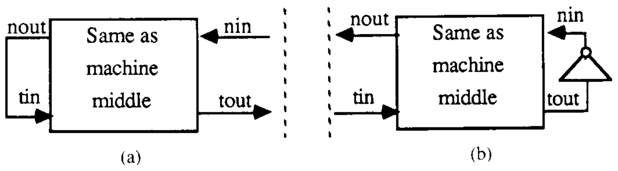

Fig. 3. Schematics of machines (a) bottom and (b) top.

assumption that relates the "delays" of the combinational circuits and flip-flops in each sequential machine in the circuit. (Such an assumption does not violate the delay insensitivity of the circuit provided it does not relate the delays of combinational circuits or flip-flops in different machines [11]). In order to state these two assumptions, we need the following definitions.

We assume that at each instant each wire has a single *value*, either "0" or "1." A wire that is an output of a combinational circuit or a flip-flop C is said to be in *equilibrium* at some instant if and only if its value is consistent with the function of C and the values of the input wires of C at that instant. We assume that if a wire is in nonequilibrium with value v at some instant t, then either it is in nonequilibrium with value v at all preceding instants, or there is an instant t' preceding t such that the wire is in equilibrium with value v at t', and it is in nonequilibrium with value v at every instant between t' and t.

Any wire that is an output of a combinational circuit C and is in nonequilibrium at some instant will return to equilibrium within some maximum time period called the *delay* of C. Any wire that is the output of a flip-flop F and is in nonequilibrium at some instant will return to equilibrium only after it has been in nonequilibrium for some minimum time period called the *delay* of F provided that the inputs of F have fixed values during the period in which the wire is in nonequilibrium. We are now ready to state the two assumptions upon which the correctness of our circuit is based:

0. Initially all wires that are outputs of combinational circuits are in equilibrium.
1. The delay of combinational circuit y in a machine is less that $\text{minimum}_{k=a,b,c}$ (delay of flip-flop k) in the same machine.

(Notice that the delay of combinational circuit z, which connects each machine with its neighboring machine remains unconstrained.)

With these assumptions, it can be shown that the two inputs of any flip-flop in the circuit cannot be "1" at any instant during the circuit's execution. Also, it can be shown that the inputs of each flip-flop cannot change so long as its output is in nonequilibrium. In other words, the circuit behaves in accordance with the three transition Tables III, IV, and V.

VII. Discussion

We have developed a self-stabilizing delay-insensitive circuit that implements a token system. We believe that the task of developing this circuit was greatly simplified by starting with systems whose state transitions are noninterfering.

The token systems that we presented are simple. Each process in the First System has at most three noncritical section states; this is the smallest number of states known for any such system. In this regard, our First System matches Dijkstra's best system [3]. Each process in the Second System has at most five noncritical section states. Finally, our hardware implementation requires only one type of machine (recall that machines bottom and top are created by making external modifications to machine middle), and only two wires between each pair of neighboring machines. The simplicity and robustness of our systems should demonstrate the elegance of self-stabilization in achieving fault tolerance.

The First System invites a number of interesting variations; we consider here two of them. The first variation reduces the potential number of state transitions that occur between one process relinquishing the token, i.e. leaving state C, and another process receiving the token, i.e., entering state C. The second variation demonstrates that the waiting commands used in our system are not necessary, and can be replaced by using CSP primitives.

In our system, the token propagates from left to right; then, once top relinquishes the token, $(n - 1)$ state transitions must occur before the next process, bottom, receives the token and enters state C. By modifying the code of process middle to become

```
middle  ::  $*[L;\ C;\ R;\ W;\ C]$
```

only one state transition occurs between any process relinquishing the token and another process receiving it. In this system, the token travels in a ping-pong fashion, propagating from left to right then from right to left, reflecting off bottom and top. An apparent problem with this variation is that it is not strictly fair; each middle process will receive the token twice as often as bottom or top.

The second variation, due to I. Forman at MCC, implements our system without using waiting commands. The processes execute the following programs:

```
bottom  ::  $*[R_0 \rightarrow$           $C\ [\!]\ R_1 \rightarrow skip]$
middle  ::  $*[R_0 \rightarrow L_0;\ L_1;\ C\ [\!]\ R_1 \rightarrow skip]$
top     ::  $*[$      $L_0;\ L_1;\ C$
```

The new operators can be defined informally as follows.

$L_i (i = 0, 1)$	is a command to communicate with the left neighbor. L_i causes the executing process to wait until its left neighbor reaches R_i, then both the executing process and the left neighbor move to their next commands.
$R_0 \rightarrow S_0\ [\!]\ R_1 \rightarrow S_1$	indicates waiting for the right neighbor to reach either L_0, in which case the executing machine executes R_0 then S_0, or L_1, in which case the executing machine executes R_1 then S_1.
skip	is the command to do nothing.

In the original system, commands R and W both test the state of the right neighbor of the executing process; the difference between the two commands is that W leaves the neighbor at command L, while R allows the neighbor to progress to command C. In this variation, a process that communicates with its right neighbor is able to determine which of the two L commands its right neighbor executed; one of the L commands, L_0, leaves the neighbor executing an L command, namely L_1, while the other L command, L_1, allows the neighbor to progress to command C. In effect, a single bit is transferred from right to left when two neighbor processes communicate.

The correctness of these two variations can be established by showing that each is, in some sense, equivalent to the First System as we did for the Second System.

For other examples of self-stabilizing systems and protocols, we refer the reader to [1] and [10].

Acknowledgment

We are indebted to E. W. Dijkstra for without his guidance this paper would not have been possible. We are thankful to the members of the Austin Tuesday Afternoon Club for suggesting many improvements in our presentation, in particular we would like to thank K. M. Chandy, A. Emerson, W. Hesselink, C. A. R. Hoare, W. Hunt, S. Lam, C. Lengauer, J. Misra, and A. Pneuli. We would also like to thank P. Attie, M. Evangelist, I. Forman, S. Katz, and J. Peterson of MCC for many helpful comments. The comments of the anonymous referees are also appreciated.

References

[1] G. M. Brown, "Self-stabilizing distributed resource allocation," Ph.D. dissertation, Dep. Elec. Comput. Eng., Univ. of Texas, Austin, 1987.
[2] E. W. Dijkstra, "EWD391 self-stabilization in spite of distributed control," reprinted in *Selected Writings on Computing: A Personal Perspective*. Berlin, Germany: Springer-Verlag, 1982, pp. 41-46.
[3] ——, "Self stabilizing systems in spite of distributed control," *Commun., ACM*, vol. 17, pp. 643-644, Nov. 1974.
[4] ——, "A belated proof to self-stabilization," *Distribut. Comput.*, vol. 1, pp. 5-6, Jan. 1986.
[5] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666-677, Aug. 1978.
[6] H. S. M. Kruijer, "Self-stabilization (in spite of distributed control) in tree-structured systems," *Inform. Processing Lett.*, vol. 8, pp. 91-95, Feb. 1979.
[7] L. Lamport, "Solved problems, unsolved problems, and non-problems in concurrency," Invited Address, in *Proc. Third ACM Symp. Principles Distribut. Comput.*, Vancouver, B.C., Canada, Aug. 1984, pp. 1-11.
[8] ——, "The mutual exclusion problem: Part II—Statement and solutions," *J. ACM*, vol. 33, pp. 327-348, Apr. 1986.
[9] A. J. Martin, "The probe: An addition to communication primitives," *Inform. Processing Lett.*, vol. 20, pp. 125-130, Apr. 1985.
[10] N. Multari, "Self-stabilizing protocols," Ph.D. dissertation, Dep. Comput. Sci., Univ. of Texas at Austin, 1987.
[11] C. Seitz, "System timing," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Reading, MA: Addison-Wesley, 1980.

Geoffrey M. Brown (S'82-M'87) received the B.S. degree in engineering from Swarthmore College, Swarthmore, PA, in 1982, the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 1983, and the Ph.D. degree in electrical engineering from the University of Texas at Austin in 1987.

From 1983 to 1984 he worked for Motorola, Austin, TX. Currently, he is an Assistant Professor in the School of Electrical Engineering, Cornell University, Ithaca, NY.

Mohamed G. Gouda has received five degrees in engineering (B.Sc. Cairo University 1968), mathematics (B.Sc. Cairo University 1971, M.A. York University 1972), computing science (M.Sc. and Ph.D. University of Waterloo 1973 and 1977).

He has worked for Honeywell Corporation (1977-1980), MCC (Summer 1986), and Bell Labs (Summer 1987). Since 1980, he has been with the Department of Computer Sciences, University of Texas at Austin, where he is currently an Associate Professor.

Dr. Gouda is the Editor-in Chief of the journal *Distributed Computing*, a fact that describes his area of research. He is also a member of the Austin Tuesday Afternoon Club.

Chuan-Lin Wu (S'74-M'78-SM'85) received the B.S.E.E. and M.S. degrees from National Chiao Tung University, Taiwan, in 1970 and 1973, respectively, and the Ph.D. degree in electrical and computer engineering from Wayne State University, Detroit, MI, in 1979.

He became an Assistant Professor in the same department where he earned his Ph.D. at Wayne State University in 1979. Next he joined the computer science faculty of Wright State University, Dayton, OH, and Ohio State University, Columbus, as an Assistant Professor in 1980 and 1981, respectively. In September 1981, he joined the faculty of The University of Texas at Austin as an Assistant Professor of Electrical and Computer Engineering, and has been an Associate Professor since September 1985. His research interests include multiprocessing, computer architecture, local area networks, expert systems, computer vision, and computer-aided VLSI design. He has extensive publications on these and related areas. As a frequently invited speaker for short courses and seminars, he has served as a consultant.

Dr. Wu has also served as an editor of IEEE TRANSACTIONS ON COMPUTERS for the period 1982-1986 and as an editor of Computer Society Press for the period of 1981-1985. He was the editor of the two special issues on multiprocessing technology (June 1985) and interconnection networks (December 1981) of *Computer*. He co-edited *Interconnection Networks for Parallel/Distributed Processing*, a tutorial book (Rockville, MD: Computer Society Press). He was also elected distinguished visitor of the Computer Society for the period 1983-1985. He is a member of the Association for Computing Machinery.