SELF-STABILIZING DIGITAL CIRCUITS

Magdy S. Abadir* and Mohamed G. Gouda

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

TR-90-10 April 1990

ABSTRACT

A digital circuit is self-stabilizing if whenever it is in some unsafe state, it converges of its own accord to a safe state in a short time. Such circuits promise an unprecedented degree of tolerance to transient faults. In this paper, we give necessary and sufficient conditions for a digital circuit to be self-stabilizing.

Keywords: digital circuits, transient faults, self-stabilization, design methodology.

---

* Microelectronics and Computer Technology Corporation, Austin, Texas.

1. Introduction

The literature on designing fault-tolerant digital circuits is considerable, spanning several textbooks and extensive tutorials [1, 5, 14, 15]. Most of this work, however, is dedicated to designing circuits that can tolerate permanent faults (usually stuck-at faults). Only a small amount of attention has so far been given to designing circuits that can tolerate transient faults.

The net result of this situation is that most current techniques for dealing with transient faults (e.g. self-checking, rollback-and-retry, and failsafe) are inadequate. First, some of these techniques (e.g. self-checking) are in fact an outgrowth of techniques for dealing with permanent faults; hence, they are not firmly based on a coherent theory that deals with transient faults. Second, some of these techniques (e.g. self-checking and rollback-and-retry) require extra circuits to be added to the original circuit in order to detect and possibly correct the effects of transient faults. Finally, some of these techniques (e.g. failsafe) require trapping the circuit, upon the detection of a fault, in a non-operating state until the fault is corrected by an external agent.

To remedy these inadequacies, we propose to investigate a new general technique, called *self-stabilization*, for dealing with transient faults. To explain this technique, note that each state of a circuit can be viewed as either safe or unsafe. The safe states are those that the circuit can reach during normal operation, while the unsafe states are those that can be reached due to some transient fault. Thus, to counter the effects of transient faults, a circuit should be designed such that whenever it is in some unsafe state, it converges of its own accord to a safe state in a short time. Once in a safe state, the circuit continues to be within safe states, unless another transient fault occurs. A circuit so designed is called self-stabilizing.

Self-stabilization does not suffer from the inadequacies of other techniques that deal with transient faults. First, self-stabilization is firmly based on a coherent theory that deals exclusively with transient faults. Second, self-stabilization does not require extra circuits to be added to the original circuit; it merely requires redesigning the original circuit, to make it self-stabilizing, using a comparable amount of hardware. (Sometimes the resulting self-stabilizing circuit requires less hardware than the original circuit; see for instance the example in Section 5.) Finally, self-stabilization never puts the circuit out of service waiting for the intervention of an external agent.

The rest of this paper is organized as follows. In Section 2, we develop a formal model, called networks, for representing digital circuits; then in Section 3, we discuss how to implement any given network by a digital circuit. In Section 4, we formally define the notion of self-stabilizing circuits, and state a central theorem that gives necessary and sufficient conditions for a digital circuit to be self-stabilizing. We utilize this theorem in Section 5 to design a self-stabilizing circuit that implements a traffic-light network. Concluding remarks are given in Section 6.

2. Networks of Machines

In this section, we present a formal model, networks of (Moore) machines, that can be used to represent digital circuits. In the next section, we discuss how to implement any given network by a digital circuit.

Machines

A *machine* is a triple

$$(Q, V, T)$$

where Q is a finite set of *states*, one of which is called the *reference state*,
V is a finite set of boolean variables called the *input variables*, and
T is a finite set of *transitions*: each transition is a triple (q, p, q') where q is a state in Q called the *current state* of the transition, q' is a state in Q called the *next state* of the transition, and p is a predicate over the variables in V.

For simplicity, we assume that the output of a machine at any instant is its current state at that instant.

Because machines are intended to model digital circuits, their transitions are required to satisfy the following two conditions:

i. *Determinism*: If p_1 and p_2 are the predicates of two machine transitions that have the same current state, then $p_1 \wedge p_2 = false$.

ii. *Non-blocking*: If $p_1, \ldots, p_n$ are the predicates of all machine transitions that have the same current state, then $p_1 \vee \ldots \vee p_n = true$.

Note that the second condition, non-blocking, implies that our machines are *fully specified.* This is a somewhat severe restriction (especially to hardware designers who are used to partially specified machines) that needs to be relaxed somewhat in the future.

It is sometimes convenient to represent each machine by a labeled directed graph as follows. Each machine state is represented by a node in the graph, and each machine transition is represented by a labeled directed edge: a transition (q, p, q') is represented by an edge from node q to node q', labeled with the predicate p. The node representing the reference state is distinguished by a small input arrow.

Example 1: Consider the traffic-light machine in Figure 1. For this machine,

the set of states	=	{red, green, yellow},
the set of input variables	=	{waiting, crossing}, and
the set of transitions	=	{(red, wait, red),
		(red, ¬wait, green),
		(green, crossing, green),
		(green, ¬crossing, yellow),
		(yellow, *true*, red)}

The reference state of this machine is "red".

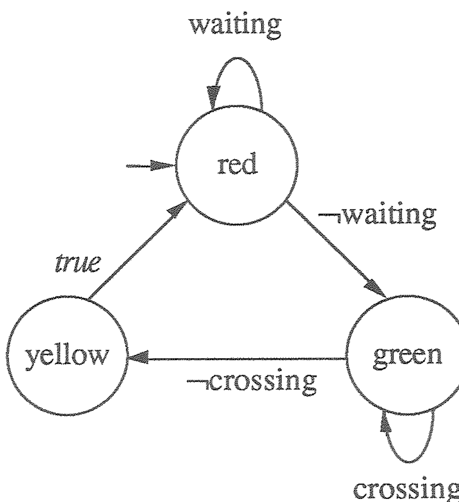


Figure 1. A traffic-light machine.

Informally, the machine remains in the "red" state until its input variable "waiting" becomes false, in which case its state becomes "green". The machine remains in the "green"

state until its input variable "crossing" becomes false, in which case its state becomes "yellow". From the "yellow" state, the machine proceeds in the next transition to the "red" state, and the cycle repeats. □

Networks

A network is a collection of machines and binding functions. Each *binding function* computes the current value of an input variable in one machine from the current state of another machine in the network; in this case the variable is said to be *bound by* the function.

Each input variable (in some machine in a network) is bound by *at most* one binding function (of the network). A variable that is bound by some function is called a *bounded variable* of the network, and a variable that is not bound by any function is called an *input variable* of the network.

Example 1 continues: As an example, consider a network that consists of two traffic-light machines M and M' that compete for right of way at an intersection. As shown in Figure 2, the two machines are similar to the one in Figure 1 with one exception: the reference state of M is "red" while the reference state of M' is "green".

The binding functions for the two variables "waiting" and "waiting' " are defined as follows:

waiting	=	false	if the current state of M' is yellow'
		true	otherwise
waiting'	=	false	if the current state of M is yellow
		true	otherwise

The two variables "crossing" and "crossing' " are not bound by any binding function; they are input variables of the network. □

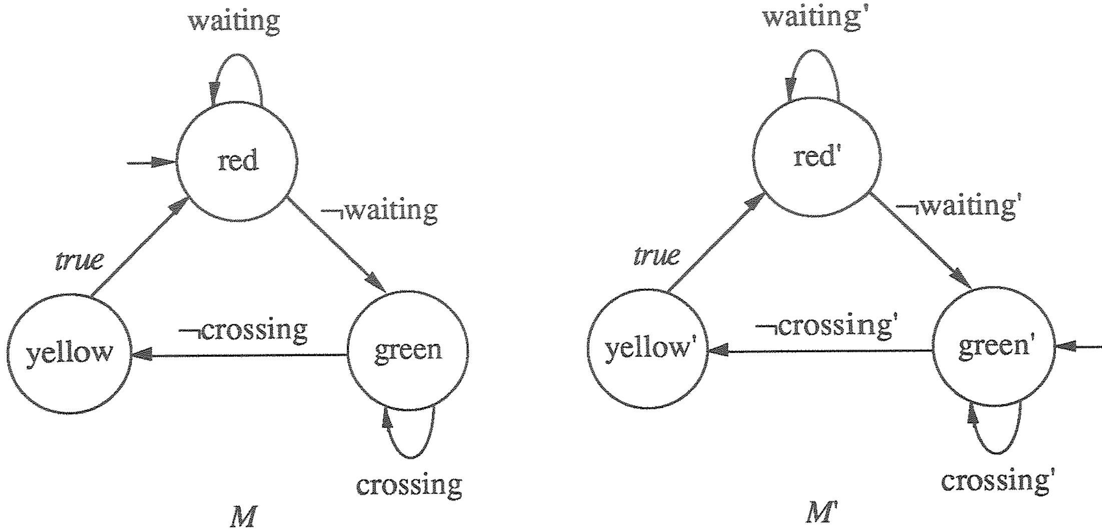


Figure 2. A network of two traffic-light machines.

A *global state* of a network is defined by one state from each machine in the network. The *reference state* of the network is defined by the reference state of each machine.

At a global state, the value of each bounded variable is uniquely determined by its binding function. Thus, if each input variable in the network is assigned a value at some global state s, then exactly one transition in every machine in the network can be executed starting from s. The execution of these transitions in parallel starting from s yields the next global state t; in this case, the pair (s, t) is called a network *step*.

A global state t is *reachable from* a global state s iff there is a finite sequence of global states $s_1,\ldots,s_n$ such that $s = s_1$, $t = s_n$, and each pair of consecutive states in the sequence is a step.

A global state is *safe* iff it is reachable from the reference state of the network.

Example 1 continues: The reference global state of our traffic-light network is (red, green'). The safe states of this network along with the network steps between them is shown in Figure 3. □

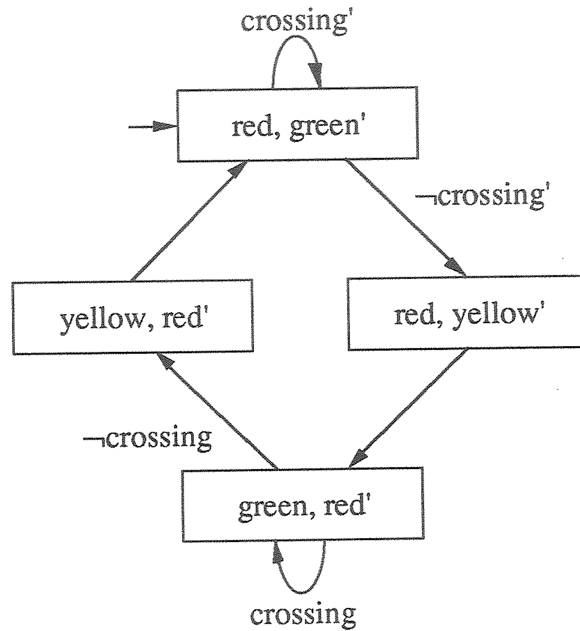


Figure 3. Safe state/step diagram of the traffic-light network.

3. Implementation of Networks by Digital Circuits

A network can be implemented by a digital circuit as follows: each machine in the network is implemented by a sequential circuit, and each binding function is implemented by a combinational circuit. These implementations can be developed according to the standard procedures for developing digital circuits.

Let C be a sequential circuit that implements some machine M. The number of states of C is a power of 2 and has to be greater than or equal to the number of states of M. Therefore, the set of states of C can be partitioned into "care" states and "don't care" states such that the following two conditions are satisfied:

i. *Conformance*: The care states of C correspond on a one-to-one basis to the states of M. Moreover, every transition between two care states in C is identical to a transition between the two corresponding states in M, and vice versa. (This condition implies that the behavior of C within its care states is isomorphic to the behavior of M.)

ii. *Avoidance*: There is no transition from a care state of C to a don't care state of C.

Define the *don't care graph* of C as follows. Each node in the graph represents a distinct don't care state in C, and each directed edge from a node q to a node q' in the graph represents a transition from the don't care state q to the don't care state q' in C.

Definition 1: C is called *convergent* iff the don't care graph of C is acyclic.

Note that the don't care graph of a circuit describes the circuit's behavior after it reaches a don't care state, due to some transient fault, and before it returns to a care state. This (abnormal) behavior is guaranteed to have a finite duration if the don't care graph is acyclic. Therefore, convergent circuits can counter the effects of transient faults better than divergent circuits.

Let D be a digital circuit that implements a given network N. Circuit D consists of some sequential circuits (each of which implements one machine in N) that are connected by some combinational circuits (each of which implements one binding function in N).

A *global state* of D is defined by one state from each sequential circuit in D. If all these states are care states, then the global state is a *care state* of D; otherwise it is a *don't care state* of D.

It is straightforward to show that the partitioning of the global states of D into care and don't care states satisfies the following two conditions.

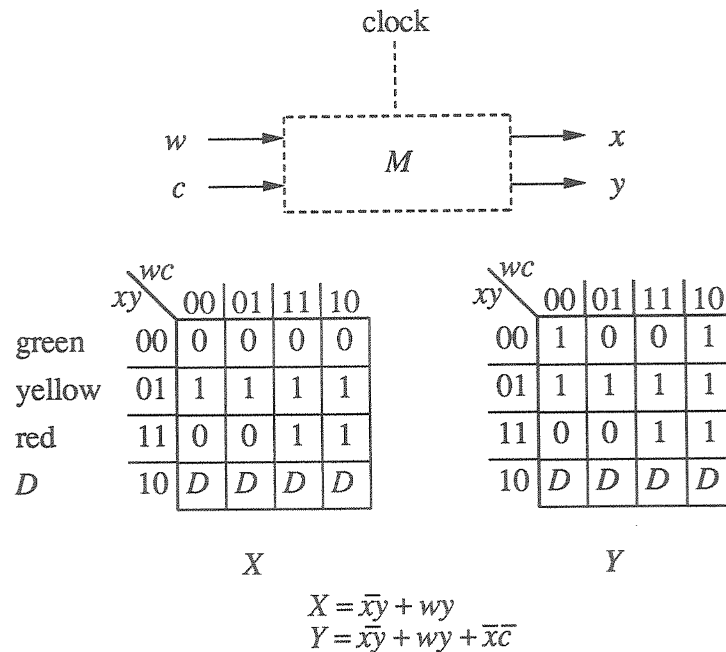
i. *Conformance*: The care states of D correspond on a one-to-one basis to the global states of N. Moreover, every step between two care states in D is identical to a step between the two corresponding global states in N, and vice versa.

ii. *Avoidance*: There is no step from a care state of D to a don't care state of D.

A care state of D is *safe* iff it corresponds to a safe state of N. A global state of D is *unsafe* iff it is either a don't care state, or a care state that corresponds to an unsafe state of N.

Example 1 continues: Let us consider how to implement our traffic-light network by a digital circuit. First, each of the two traffic-light machines is implemented by a sequential circuit. Second, each of the two binding functions is implemented by a combinational circuit. Finally, all four circuits are interconnected to form the required digital circuit.

An implementation of the traffic-light machine M is shown in Figure 4. The two inputs w and c of the circuit implement the two input variables "waiting" and "crossing" of M. The two outputs x and y of the circuit implement the states of M as follows. The care states of M namely green, yellow, and red are implemented by the xy-combinations 00, 01, and 11, respectively. The remaining combination 10 is a don't care state.



	xy \ wc	00	01	11	10
green	00	0	0	0	0
yellow	01	1	1	1	1
red	11	0	0	1	1
D	10	*D*	*D*	*D*	*D*

X

xy \ wc	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	0	0	1	1
10	*D*	*D*	*D*	*D*

Y

$$X = \bar{x}y + wy$$
$$Y = \bar{x}y + wy + \bar{x}\bar{c}$$

Figure 4. Implementation of the traffic-light machine M.

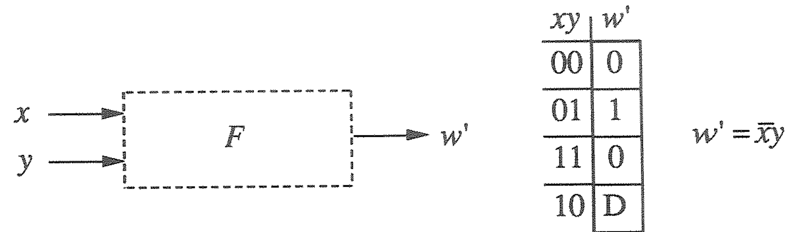
Referring to Figure 4, the "next state" XY of the circuit is determined from the "current state" xy of the circuit and the "current inputs" wc according to the following equations:

$$X = \bar{x}y + wy$$
$$Y = \bar{x}y + wy + \bar{x}\,\bar{c}$$

The other traffic-light machine M' can be implemented by a similar circuit. The inputs of the circuit are w' and c' which implement the two input variables "waiting' " and "crossing' ". The outputs of the circuit are x' and y' which implement the states of M'. The next state $X'Y'$ of the circuit is determined from its current state $x'y'$ and the current inputs $w'c'$ according to the following equations:

$$X' = \bar{x}'y' + w'y'$$
$$Y' = \bar{x}'y' + w'y' + \bar{x}'\,\bar{c}'$$

The binding function F that computes the current value of "w'" (which implements "waiting'" in M') in terms of the current state xy of M can be implemented by the combinational circuit $w' = \bar{x}y$ as shown in Figure 5.



xy	w'
00	0
01	1
11	0
10	D

Figure 5. Implementation of the binding function F that computes w' from xy.

Similarly, the other binding function F' can be implemented by the combinational circuit $w = \bar{x}'y$.

An outline for the circuit that implements the traffic-light network is shown in Figure 6.

□

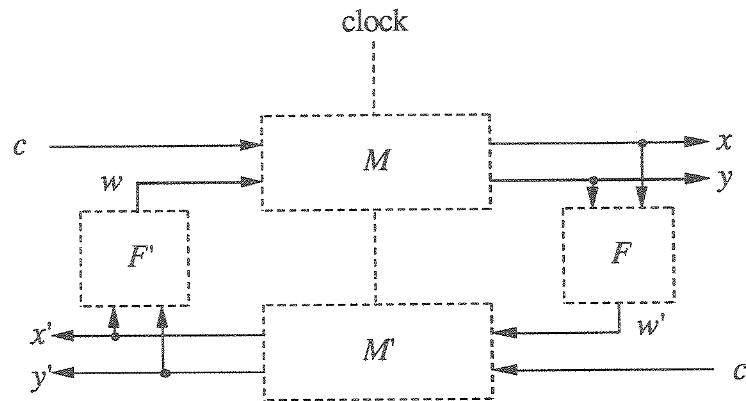


Figure 6. Implementation of the traffic-light network.

4. Self-Stabilization of Networks and Circuits

As mentioned in the previous section, a safe state of a network is a global state that is reachable from the reference state of the network. It follows that each safe state is followed by a safe state. In other words, once the network is in a safe state, it will continue to be in

safe states unless some transient fault leads the network to some unsafe state (i.e., one that is not reachable from the reference state of the network). In order for the network to be able to counter the effects of transient faults, it should be designed such that if it is ever in some unsafe state, then it is guaranteed to reach a safe state in a finite number of steps. This is precisely the definition of a self-stabilizing network.

Definition 2: A network is *self-stabilizing* iff starting from any unsafe state, the network is guaranteed to reach a safe state in a finite number of steps. []

A similar definition applies to digital circuits that implement given networks.

Definition 3: A digital circuit that implements a given network is *self-stabilizing* iff starting from any unsafe state, the circuit is guaranteed to reach a safe state in a finite number of steps. []

A central result of our theory is the next theorem which states necessary and sufficient conditions for a digital circuit to be self-stabilizing. (This theorem can be proved in a straightforward manner from the above definitions.)

Theorem 1: If a network is self-stabilizing and each of its machines is implemented by a convergent sequential circuit, then the resulting digital circuit that implements the network is self-stabilizing. Conversely, if a digital circuit that implements a given network is self-stabilizing, then the network is self-stabilizing and each of its machines is implemented by a convergent sequential circuit. []

Thus, in order to establish that a digital circuit D which implements a given network N is self-stabilizing, it is sufficient (and necessary) to verify the following two conditions:

i. *Stabilization*: N is self-stabilizing.

ii. *Convergence*: Each machine in N is implemented by a convergent sequential circuit in D.

The second condition, convergence, can be checked by a simple algorithm. Therefore, proving that a digital circuit is self-stabilizing amounts in principle to proving that its network is self-stabilizing.

Next we apply Theorem 1 to show that the digital circuit in Figures 4, 5, and 6 which implements our traffic-light network is not self-stabilizing.

Example 1 continues: To show that any digital circuit that implements the traffic-light network is not self-stabilizing, it is sufficient (according to Theorem 1) to show that the network itself is not self-stabilizing. Referring to Figure 3, the network has four safe states; they are:

(red, green'), (red, yellow'), (green, red'), (yellow, red')

It follows that the global state (red, red') is unsafe. If the network starts at this state where both "waiting" and "waiting' " are *true*, then the next state is also (red, red'), and the network is locked at this unsafe state indefinitely. This shows that the network is not self-stabilizing. Hence, each digital circuit that implements it, including the one in Figures 4, 5, and 6, is not self-stabilizing. □

5. A Self-Stabilizing Traffic-Light Circuit

Our traffic-light network is not self-stabilizing because it is *symmetric*: it has two identical machines and two identical binding functions. Thus, if the network ever falls into a symmetric (unsafe) state, e.g. (red, red'), it cannot find its way back to safe asymmetric states. Therefore, in order to make this network self-stabilizing, one needs to break the symmetry.

We choose to make the network asymmetric by changing one of its two machines, machine M, while keeping the other machine M' unchanged. We also change one of the two binding functions, F' (which computes "waiting" in M), while keeping the other function F unchanged.

The new machine M is shown in Figure 7, and the new binding function F' is defined as follows.

waiting	=	false	if the current state of M' is yellow' or red'
		true	otherwise

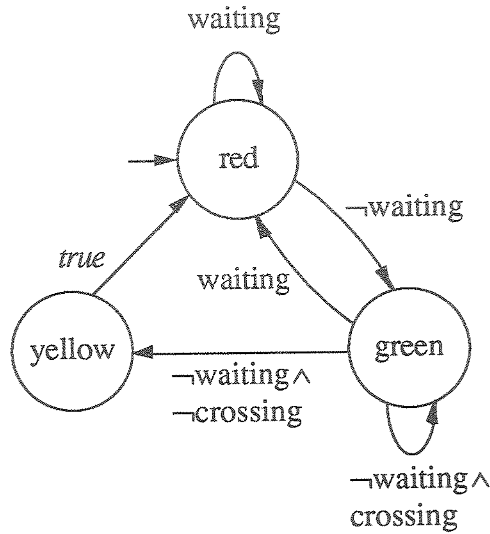


Figure 7. The redesigned *M*.

To show that the resulting network after these changes is self-stabilizing, we need to show that from any unsafe state the network will converge to a safe state in a finite number of steps. Figure 8 shows all the global states of the network, and the network steps between them. Clearly, if the network is ever in some unsafe state, it will reach one of its four safe states, namely (red, green'), (red, yellow'), (green, red), and (yellow, red), in at most two steps.

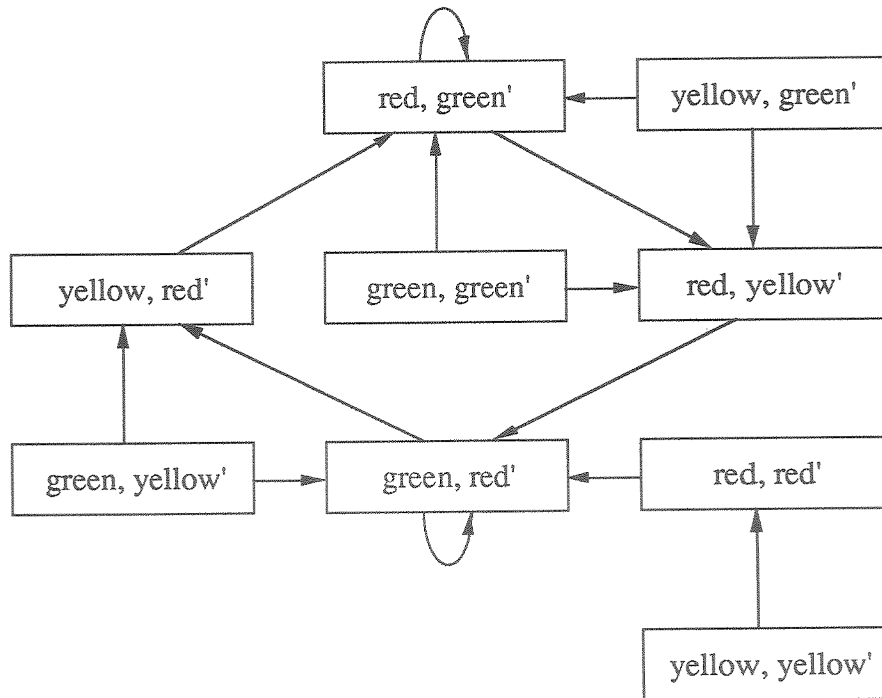


Figure 8. State/step diagram of the redesigned network.

An implementation of the new M is shown in Figure 9 and an implementation of the new F' is shown in Figure 10. Note that these implementations require *less hardware* than the implementations of the old M (Figure 4) and old F' (Figure 5). This shows that self-stabilization may contribute indirectly to network simplicity and hardware saving.

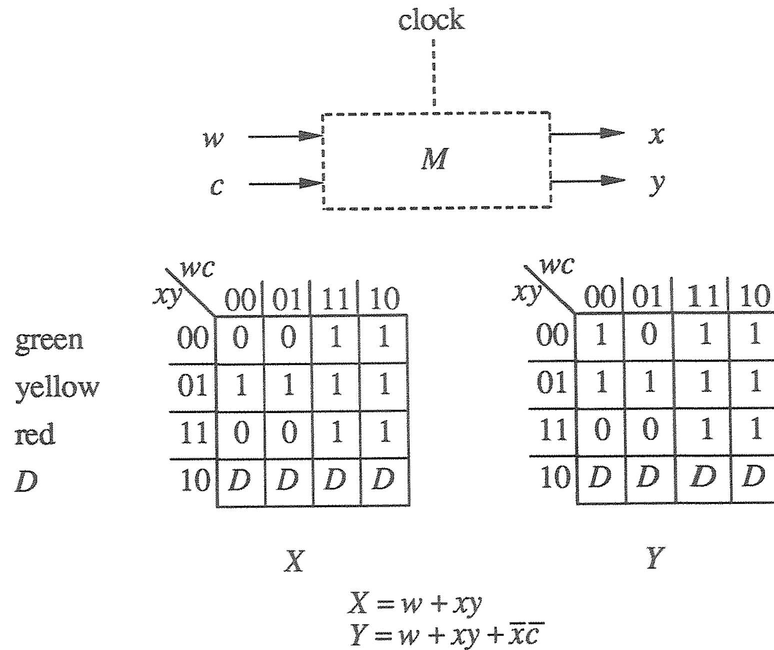


Figure 9. Implementation of the redesigned M.

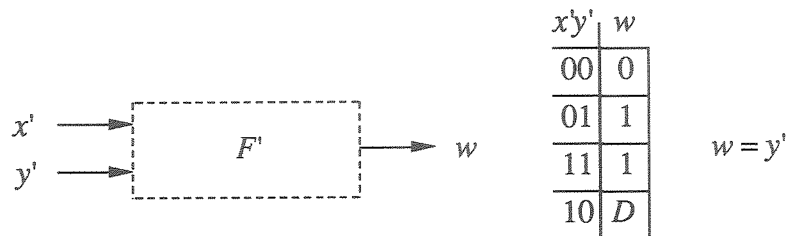


Figure 10. Implementation of the redesigned F'.

It is straightforward to check that the two sequential circuits in Figures 4 and 9 (which implement machines M' and M, respectively) are convergent. From the convergence of these circuits and from the fact that the network is self-stabilizing, we conclude that the resulting digital circuit that implements the network is self-stabilizing by Theorem 1.

6. Concluding Remarks

To the best of our knowledge, our proposal to study the design and applications of self-stabilizing digital circuits is the first of its kind. Nevertheless, there is a growing interest in self-stabilization and its applications to distributed systems. Among the more active researchers in this area are E. Dijkstra at the University of Texas at Austin [9], A. Arora and M. Evangelist at MCC [2, 3, 10], F. Bastani at the University of Houston [4], J. Burns at Georgia Tech and J. Pachl at IBM Zurich [7], G. Brown at Cornell University [6], A. Israeli and S. Moran at the Technion [8], S. Katz at the Technion and K. Perry at IBM Yorktown Heights [12], and N. Multari in the Air Force [11, 13].

Our discussion of self-stabilizing digital circuits suggests three important problems that merit further research.

First, Theorem 1 suggests that understanding how to achieve self-stabilization in networks is the key to achieving self-stabilizing digital circuits. Therefore, our first task is to explore different ideas, heuristics, and concepts that can be used in making a network self-stabilizing. One such concept is "asymmetry": recall that the traffic-light network is made self-stabilizing by breaking its symmetry. Other examples that we have considered confirm that asymmetry is a powerful concept for achieving self-stabilization. A second promising concept is "linear progression of convergence" from unsafe to safe states.

An important aspect of this task is to study different ways for achieving self-stabilization in networks of *partially specified* machines. (Recall that our current presentation is based on fully specified machines.)

A third problem that merits research is to identify classes of networks for which achieving self-stabilization is easy and inexpensive; for instance, networks that can be made self-stabilizing without changing the number of machines in the network or their interconnections. (Our traffic-light network is an example of such networks.)

References

[1] M. Abramovici, M. Breuer, and A. Friedman, "Digital systems testing and testable design," *Computer Science Press*, 1990.

[2] A. Arora, P. Attie, M. Evangelist, and M. G. Gouda, "Convergence of iteration systems," *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report Number STP-379-89; also submitted for journal publication.

[3] A. Arora and M. G. Gouda, "Distributed reset," submitted for journal publication.

[4] F. Bastani, I. Yen, and I. Chen, "Classes of inherently fault-tolerant distributed programs," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, pp. 1432–1442, 1988.

[5] M. Breuer and A. Friedman, "Diagnosis and reliable design of digital systems," *Computer Science Press*, 1976.

[6] G. M. Brown, M. G. Gouda, and C.-L. Wu, "Token systems that self-stabilize," *IEEE Transactions on Computers*, Vol. 38, No. 6, pp. 845–852, 1989.

[7] J. Burns and J. Pachl, "Uniform self-stabilizing rings," *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 2, pp. 330–344, 1989.

[8] S. Dolev, A. Israeli, and S. Moran, "Self-stabilization of dynamic systems," *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report Number STP-379-89.

[9] E. W. Dijkstra, "A belated proof of self-stabilization," *Distributed Computing*, Vol. 1, No. 1, pp. 1–2, 1986.

[10] M. G. Gouda and M. Evangelist, "Convergence response tradeoffs in concurrent systems," MCC Technical Report Number STP-124-89; also accepted for publication in *ACM Transactions on Programming Languages and Systems*, 1990.

[11] M. G. Gouda and N. Multari, "Self-stabilizing communication protocols," in preparation, 1990.

[12] S. Katz and K. Perry, "Self-stabilizing extensions for message-passing systems," *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report Number STP-379-89.

[13] N. Multari, "Towards a theory for self-stabilizing protocols," Ph.D. dissertation, Department of Computer Sciences, University of Texas at Austin, 1989.

[14] V. Nelson and B. Carroll (Eds.) "Tutorial: fault-tolerant computing," *The Computer Society of the IEEE*, 1987.

[15] D. K. Pradhan (Ed.), "Fault-tolerant computing, theory, and techniques," *Prentice Hall*, 1986.