Proceedings of 1992 International Conference
on Parallel and Distributed Systems
Dec. 16-18, 1992, Hsinchu, Taiwan, R.O.C.

The Stabilizing Computer

Extended Abstract

[1]Magdy S. Abadir and [2]Mohamed G. Gouda

[1]MCC
[2]Department of Computer Sciences
University of Texas at Austin

Abstract

A synchronous digital circuit is stabilizing iff starting from any un-intended state (for example, a don't care state, or an inconsistent combination of flipflop states), the circuit is guaranteed to converge to its intended states in short time. A stabilizing circuit can tolerate transient faults which may yield the circuit in an arbitrary, possibly un-intended, state. In this paper, we present a method for designing stabilizing digital circuits. The method is based on local detection and correction. The method guarantees that every designed circuit will converge to its intended states in linear time, linear in the number of machines in the circuit.

Keywords: design method, digital circuits, fault-tolerance, stabilization, synchronous circuits, transient-faults.

Acknowledgement: Gouda is supported in part by the grant 003658-250 from the Texas Advanced Technology Program, 1992-1994.

1 Introduction

The literature on designing fault-tolerant digital circuits is considerable, spanning several textbooks an extensive tutorials; see for example [1], [5], [15], and [16]. 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, roll-back-and-retry, and failsafe, are inadequate. First, some of these techniques, e.g. self-checking, are in fact mere extensions of techniques for dealing with permanent faults; hence, they are not firmly based on coherent theory that can deal with transient faults. Second, some of these techniques, e.g. self-checking and roll-back-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 stabilization, for dealing with transient faults in digital circuits. 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 un-intended states. e.g. don't care states, which can be reached only due to some transient fault. 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 within safe states, unless another transient fault occurs. A circuit so designed is called stabilizing.

This technique of stabilization does not suffer from the inadequacies of other techniques that deal with transient faults in digital circuits. First, stabilization is firmly based on a coherent theory that can deal with transient faults. Second, stabilization does not require extra circuits to be added to the original circuit; it merely requires redesigning the original circuit, to make it stabilizing, using a comparable amount of hardware. Finally, stabilization never puts the circuit out of

service waiting for the intervention of an external agent.

The idea of stabilization of computing systems is not new; see for instance [9], [10], and [14]. This idea has since been applied in a number of areas in fault-tolerant computing; see for example [2], [3], [4], [6], [7], [8], [12], and [13]. However, to the best of our knowledge, this is the first time that this idea has been proposed as a basis for dealing with transient faults in digital circuits.

The rest of this paper is organized as follows. In Section 2, we identify a class of machines that can be implemented by digital circuits. Then, in Section 3, we discuss stabilizing implementations of machines and present an algorithm that derives, from any given machine, a stabilizing digital circuit that implements the machine. In Section 4, we discuss how to connect a set of machines into a network. Then, in Section 5, we give a set of sufficient conditions for ensuring that a circuit implementation of a given network is stabilizing. One of those conditions is that the given network is stabilizing. Thus, in Section 6, we discuss a method for designing stabilizing networks. Concluding remarks are given in Section 7. Because of the space limitation, the proofs of our three theorems in the paper are omitted.

2 Machines and implementations

In this section, we present machines that can be implemented by digital circuits. We also discuss the concept of one machine being an implementation of another machine.

A machine is a triple

(S, V, T)

where

S is a finite set of states, one of which is called reference state,
V is a finite set of boolean variables called input variables, and
T is a finite set of steps, each of which is a triple (r, B, s), where
r and s are states in S, and
B is a logical proposition, not equal to false, over the variables in V.
Step (r, B, s) is called an outgoing step of state r, and an ingoing step of state s.

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

Because machines are to be implemented by digital circuits, each machine (S, V, T) satisfies the following two conditions.

Liveness:
Every state in S has at least one outgoing step in T. Moreover, if B.1, ... , B.k are the propositions of all outgoing steps of some state in S, then $B.1 \vee ... \vee B.k = true$.

Determinism:
If B.1 and B.2 are propositions of two distinct outgoing steps of some state in S, then $B.1 \wedge B.2 = false$.

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 step is represented by a labeled directed edge. In particular, a step (r, B, s) is represented by an edge from node r to node s, labeled with proposition B. Because of the liveness condition, each node in the graph has at least one outgoing edge.

Let (S, V, T) be a machine. A state s in S is called safe iff either s is the machine reference state, or there is a nonempty sequence of steps

(s.1, B.1, s.2), ... , (s.(k-1), B.(k-1), s.k)

in T such that s.1 is the machine reference state and s.k is state s. A state in S that is not safe is called unsafe.

It follows from this definition that if r is a safe state of some machine and (r, B, s) is a step of that machine, then s is a safe state of that machine.

An implementation of a machine (S, V, T) is a machine (S', V', T') such that the following three conditions are satisfied.

State Correspondence:
There is a one-to-one correspondence between the safe states in S and the safe states in S' such that the reference state in S corresponds to the reference state in S'.

Input Correspondence:
There is a one-to-one correspondence between the variables in V and the variables in V'.

Step Correspondence:
For every step (r, B, s) in T, where r and s are safe, there is a step (c.r, c.B, c.s) in T', where

c.r and c.s are the safe states in S' that correspond to states r and s in S, and
c.B is a proposition identical to B except that every variable in V is replaced by the corresponding variable in V'.

The first condition, state correspondence, implies that an implementation has the same number of safe states as the implemented machine. The second condition, input correspondence, implies that an implementation has the same number of input variables as the implemented machine. The third condition, step correspondence, implies that each safe state c.s in an implementation has at least as many outgoing steps as the corresponding state s in the implemented machine. Moreover, because the outgoing steps at s satisfy the two conditions of liveness and determinism, the corresponding outgoing steps at c.s satisfy the conditions of liveness and determinism. Thus, if c.s has more outgoing steps than s, then each of the additional step violates the liveness condition or the determinism condition. Therefore, the third condition implies that every safe state c.s in an implementation has exactly as many outgoing steps as the corresponding state s in the implemented machine.

It follows from this definition that every machine is an implementation of itself.

3 Stabilizing implementations of machines

Any execution of an implementation starting from a safe state closely resembles a corresponding execution of the implemented machine starting from the corresponding (safe) state. On the other hand, executions of an implementation starting from unsafe states are arbitrary. This arbitrariness can be constrained by requiring that every execution of an implementation starting from an unsafe state eventually reaches a safe state. This constraint can be formalized by the following definitions of stabilizing machines and stabilizing implementations.

A machine (S, V, T) is stabilizing iff for every directed cycle of steps
(s.1, B.1, s.2), ... , (s.(k-1), B.(k-1), s.k),
(s.k, B.k, s.1)
in T, at least one of the states s.1, s.2, ... , s.k is a safe state.

This definition of a stabilizing machine deserves a closer look. Let (S, V, T) be a stabilizing machine, and assume that T has the following directed cycle of steps:
(s.1, B.1, s.2), ... , (s.(k-1), B.(k-1), s.k),
(s.k, B.k, s.1).
According to the definition of stabilizing machine, at least one of the states s.1, ... , s.k is a safe state. Because there are no steps from safe states to unsafe states, it is straightforward to show that in fact each of the states s.1, ... , s.k is a safe state.

A machine (S', V', T') is a stabilizing implementation of a machine (S, V, T) iff (S', V', T') is an implementation of (S, V, T) and (S', V', T') is stabilizing.

Stabilizing implementations are useful for the following reason. If a stabilizing implementation is executed starting from any (possibly unsafe) state, the execution is guaranteed to reach a safe state after a finite number of steps. Once the execution reaches a safe state, it closely resembles a corresponding execution of the implemented machine within its safe states.

Next, we discuss an algorithm for designing a digital circuit that is a stabilizing implementation of any given machine. For simplicity, we assume that the states of the given machine are all safe states. (This is not a severe restriction: the unsafe states in any machine definition can be identified and removed from the machine definition.)

Algorithm 1

Given
A machine (S, V, T) whose states are all safe states.

Result
A digital circuit that is a stabilizing implementation of the given machine (S, V, T).

Steps
if the number of (safe) states in S equals 2^k, for some k

then use the standard procedure [11] to implement (S, V, T) as a digital circuit with k flipflops

else find an integer n such that $2^{n-1} < m < 2^n$, where m is the number of (safe) states in S, then implement (S, V, T) as a digital

circuit C with n flipflops according to the following five steps.

i. The states of C are represented by n bits (c.1, ... , c.n).

ii. Partition the 2^n states of C into m safe states and 2^n-m unsafe states such that every state of the form (0, c.2, ... ,c.n) is a safe state. (This is possible because the number m of safe states is larger than the number 2^n-m of unsafe states.)

iii. Establish a one-to-one correspondence between the safe states of C and the (safe) states in S.

iv. Use the standard procedure [11] to design the logic of bit c.1 such that the following two conditions hold. First, the next states after a safe state are the expected safe states as defined by the steps in T. Second, the next state after an unsafe state, which is of the form (1, c.2, ... , c.n), is some (safe) state of the form (0, c'.2, ... , c'.n).

v. Use the standard procedure [11] to design the logic for each of the other bits c.2, ... , c.n.

end

Theorem 1:
For any given machine, Algorithm 1 can be used to design a digital circuit that is a stabilizing implementation of the given machine.

Note that the circuit that results from Algorithm 1 is guaranteed to reach a safe state and start simulating the given machine in at most one step.

4 Networks and implementations

A network (of machines) is a pair

(M, F)

where M is a finite set of machines, and F is a finite set of boolean functions called binding functions.

Each binding function in F computes the current value, true or false, of one input variable in one machine in M from the the current state of another machine in M. In this case, the variable is said to be bound by the function. Each input variable in some machine in M is bound by at most one binding function in F. A variable that is bound by some function in F is called a bound variable of the network, and a variable that is not bound by any function in F is called an input variable of the network.

A state of a network (M, F) is defined by one state from each machine in M. The reference state of the network is defined by the reference state of each machine in M.

At a network state r, the value of each bound variable of the network is uniquely determined by the binding function of that variable. Thus, if each input variable of the network is assigned a value at state r, then exactly one step in every machine in the network can be executed starting from r. The execution of these steps in parallel starting from r yields the next network state s. In this case, the network state pair (r, s) is called a network step.

A computation of a network is an infinite sequence s.1, s.2, ... of network states such that each pair (s.i, s.(i+1)) is a network step.

Let (M, F) be a network. An implementation of (M, F) is a network (M', F') such that the following two conditions are satisfied.

Machine Correspondence:
There is a one-to-one correspondence between the machines in M and those in M' such that each machine in M' is an implementation of the corresponding machine in M.

Binding Correspondence:
There is a one-to-one correspondence between the binding functions in F and those in F' such that if a function f in F computes the current value of some input variable v in machine m from the current state of machine n, then the corresponding function f' in F' computes the current value of input variable v' in machine m' from the current state of machine n', where variable v corresponds to variable v, and machines m' and n' correspond to machines m and n, respectively. Moreover, the value of f for any state of n equals the value of f' for the corresponding state of n'.

The two conditions for a network (M', F') to be an implementation of a network (M, F) need

some explanation. Let m and n be two machines in M. Then, by the machine correspondence condition, M' has two machines m' and n' such that m' is an implementation of m and n' is an implementation of n. Because m' is an implementation of of m, there is a one-to-one correspondence between the input variables in m and those in m'. Let variable v in m corresponds to variable v' in m'. Now, let f be a binding function in F that computes the current value of v from the current state of machine n. By the binding correspondence condition, F' has a corresponding function f'. Function f' computes the current value of v' from the current state of machine n'.

It follows from this definition that every network is an implementation of itself.

5 Stabilizing implementations of networks

In this section, we define the concept of a stabilizing network, and discuss a set of sufficient conditions for a network implementation to be stabilizing.

A network state s is called safe iff there is a network computation s.1, s.2, ... , s.k, ... such that s.1 is the network reference state, and s.k is state s. A network state that is not a safe state is called unsafe.

It is straightforward to show that if r is a safe network state and (r, s) is a network step, then s is a safe state. It then follows that if a network computation has a safe state, then every state that follows the safe state in the computation is safe.

A network is stabilizing iff every network computation has a safe state.

We have defined the concept of stabilizing machines in Section 3 and the concept of stabilizing networks in the current section. For these two definitions to be consistent, one should expect that a network of one machine is stabilizing iff the machine is stabilizing. Consider a network that consists of one machine m and no binding functions. States of the network are simply the states of m, and steps of the network are the steps of m. Moreover, the network reference state is the reference state of m. The fact that machine m is stabilizing is equivalent to the fact that every state of m is reachable by a sequence of steps from the reference state of m. This latter fact is equivalent to the fact that every network state is reachable by some computation from the network reference state, which in turn is equivalent to the fact that the network is stabilizing. Thus, every network that consists of one machine is stabilizing iff its machine is stabilizing.

The following theorem states sufficient conditions for the stabilization of a network implementation.

Theorem 2:
If a network (M, F) is stabilizing, and if an implementation (M', F') of (M, F) is such that every machine in M' is a stabilizing implementation of the corresponding machine in M, then network (M', F') is stabilizing.

Theorem 2 states two sufficient conditions for a network implementation to be stabilizing. First, the original network should be stabilizing. Second, the implementation of each machine in the network should be stabilizing. In order to achieve the second condition, we have presented in Section 3 an algorithm for realizing stabilizing circuit implementations of machines. The remaining question now is how to design stabilizing networks, This question is partly answered in the next section.

6 A method for designing stabilizing networks

In this section, we discuss a method for designing stabilizing linear networks. We start by defining linear networks.

A network (M, F) is linear iff M = {m.1, ... , m.k} and for every binding function f in F, there is a machine m.i in M such that f computes the current value of an input variable in m.i either from the current state of m.(i-1) or from the current state of m.(i+1). Machine m.(i-1) is called the left neighbor of machine m.i, and machine m.(i+1) is called the right neighbor of m.i. Machine m.1 has no left neighbor, and so is called the left-most machine. Similarly, machine m.k has no right neighbor, and so is called the right-most machine.

Our method for designing stabilizing linear networks is based on two principles.

Local Detection in a Linear Network:
In each unsafe state of the network, at least two neighboring machines in the network have "inconsistent states".

Local Correction in a Linear Network:
Whenever two neighboring machines in the network have "inconsistent states", the state of the right machine is changed to become consistent with the state of the left machine.

Clearly, the local detection principle is not valid for every linear network. Thus, we need some sufficient conditions which if satisfied by a linear network, the local detection principle is valid for that network. A set of sufficient conditions is given in Theorem 3 below. But first, we need to define the concepts of left and right machine states, and the concept of left-right network states.

Let (M, F) be a linear network, and let m.i be a machine in M. A state s of m.i is a left state iff the left neighbor m.(i-1), if any, has a state r such that in every safe state of the network, if the state of m.i is s, then the state of m.(i-1) is r. State r is called the shadow of state s.

Similarly, we can define a right state of a machine in a linear network and its shadow state.

Note that the left-most machine in a linear network has no left neighbor; thus each of its states is a left state. Similarly, the right-most machine in a linear network has no right neighbor; thus each of its states is a right state.

It is possible that a machine state is both a left state and a right state. To avoid confusion, we henceforth regard such a state as a right state.

A state of a linear network is a left-right state iff in this state each machine in the network is either in a left state or in a right state, and if a machine is in a right state, then its right neighbor is in a right state.

The next theorem states a sufficient condition for the validity of the local detection principle for linear networks.

Theorem 3:
If every safe state of a linear network is a left-right state, then for each unsafe state s, there are at least two neighboring machines whose state pair in s does not occur in any safe state of the network.

The antecedent of this theorem is a condition on the safe states of a network, whereas the consequence is a condition on the unsafe states of the same network. This is nice for two reasons. First, the number of safe states in a network is usually much smaller than the number of unsafe states in the same network; thus by making the smaller set satisfy some condition, the much bigger set is guaranteed to satisfy a corresponding condition. Second, network designers usually design, and focus on, the safe states of their networks, this theorem allows them to continue this tradition.

7 Concluding remarks

The main result in this paper is Theorem 2. This theorem states two sufficient conditions for ensuring that a circuit implementation of a given network is stabilizing. First, the given network is stabilizing. Second, each machine in the circuit is implemented by a stabilizing circuit. In order to achieve these two conditions, we have developed Theorem 3 (to achieve the first condition) and Algorithm 1 (to achieve the second condition). Theorem 3 gives a sufficient condition to ensure that the local detection principle can be applied to a linear network and so the network can be made stabilizing by the local correction principle. Algorithm 1 can be used to develop stabilizing circuit implementation for any given machine.

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. Gouda, "Convergence of iteration systems", Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report Number STP-379-89, 1989; also Distributed Computing to appear 1992.

[3] A. Arora and M. G. Gouda, " Distributed reset", Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 472, Springer-Verlag, pp. 316-331, 1990.

[4] F. Bastani, I. Yen, and I. Chen, "Classes of inherently fault-tolerant distributed programs", IEEE Transactions on Software Engineering, Vo..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, 1989.

[9] E. W. Dijkstra, " Self-stabilizing systems in spite of distributed control", Communications of the ACM, Vol. 17, No. 11, 1974.

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

[11] D. D. Givone, "Introduction to Switching Circuit Theory", McGraw-Hill Computer Science Series, 1970.

[12] M. G. Gouda and N. Multari, "Stabilizing communication protocols", IEEE Transactions on Computers, Vol. 40, No. 4. pp. 448-458, 1991.

[13] S. Katz and K. Perry, "Self-stabilizing extensions for message passing systems", Proceedings of the ACM Symposium on Principles of Distributed Computing, pp. 91-101, 1990.

[14] L. Lamport, "Solved problems, unsolved problems, and non-problems in concurrency", Invited Talk, Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, pp. 1-11, 1984.

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

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