Multiphase Stabilization

Mohamed G. Gouda

Abstract—We generalize the concept of stabilization of computing systems. According to this generalization, the actions of a system S are partitioned into n partitions, called phase 1 through phase n. In this case, system S is said to be n-stabilizing to a state predicate Q iff S has state predicates P.0, ..., P.n such that P.0 = true, P.n = Q, and the following two conditions hold for every j, $1 \leq j \leq n$. First, if S starts at a state satisfying P.(j-1) and if the only actions of S that are allowed to be executed are those of phase j or less, then S will reach a state satisfying P.j. Second, the set of states satisfying P.j is closed under any execution of the actions of phase j or less. By choosing n = 1, this generalization degenerates to the traditional definition of stabilization. We discuss three advantages of this generalization over the traditional definition. First, this generalization captures many stabilization properties of systems that are traditionally considered nonstabilizing. Second, verifying stabilization when $n > 1$ is usually easier than when n = 1. Third, this generalization suggests a new method of fault recovery, called multiphase recovery.

Index Terms—Computing system, convergence, multiphase recovery, periodic reset, self-stabilization, spanning tree construction.

1 Introduction

Stabilization is an important property of computing systems. This property can be used in explaining fault tolerance (in particular, fault occurrence [1], fault recovery [1], [4], and fault containment [7]) of these systems. It can also be used in explaining the adaptivity of these systems to their environments [9]. Moreover, the strive to achieve stabilization has led to many interesting systems [3], [13].

Unfortunately, the growing research concerning system stabilization is marred by two problems:

1. The definition of a stabilizing system is strict. Thus, many computing systems that one feels should be considered stabilizing are in fact nonstabilizing according to this definition.
2. The strictness of this definition makes it hard to design stabilizing systems or verify their correctness.

In this paper, we address these two problems by introducing a generalization for system stabilization, called multiphase stabilization. Systems that are stabilizing according to the traditional definition are still stabilizing according to the new generalization. However, some systems that are traditionally considered nonstabilizing become stabilizing according to this generalization. We also argue that the new generalization makes proving system stabilization easier and suggests a new method of fault recovery.

The rest of this paper is organized as follows: In Section 2, we define multiphase stabilizing systems and propose a new fault recovery method for these systems in Section 3. In Section 4, we discuss three examples of multiphase stabilizing systems and, in Section 5, we present several theorems that can be used in verifying the stabilization properties of multiphase systems. In Section 6, we discuss how to combine two (or more) multiphase stabilizing systems into a single system. Then, in Section 7, we outline a transformation for transforming any multiphase stabilizing system into a single-phase stabilizing system. Concluding remarks are in Section 8.

• *The author is with the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188.*
E-mail: gouda@cs.utexas.edu.

Manuscript received 5 Apr. 1999; accepted 22 June 2001.
Recommended for acceptance by W. Sanders.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 109528.

2 Definition of Multiphase Stabilization

A (computing) system is a nonempty set of variables whose values are from predefined domains and a nonempty set of actions that can be executed to update the values of the variables. Each action is of the form:

$$< \text{guard} > \ -> \ < \text{statement} >$$

where $< \text{guard} >$ is a Boolean expression over the system variables and $< \text{statement} >$ is a sequence of assignment and conditional statements over the system variables.

A state of a system S is an assignment of a value to each variable of S. The value assigned to each variable is from the domain of that variable.

A state p of a system S is called a fixed point iff the guard of each action of S is false at state p.

A transition of a system S is a triple (p, c, q), where p and q are states of S, c is an action of S, the guard of action c is true at state p, and executing the statement of action c when system S is in state p yields S in state q. For any transition (p, c, q), p is called the tail state of the transition, q is called the head state of the transition, and action c is said to be enabled at state p.

A computation of a system S is a nonempty sequence of transitions of S such that the following three conditions hold:

1. **Order**: In the sequence, the head state of each transition is the same as the tail state of the next transition, if any.
2. **Maximality**: If the sequence is finite, then the head state of the last transition in the sequence is a fixed point.

3. **Fairness**: If the sequence contains a contiguous subsequence of the form (p.0, c.0, p.1) , (p.1, c.1, p.2) , ... , and if an action c is enabled at each of the states p.0, p.1, ... in the subsequence, then the subsequence is finite or action c is one of actions c.0, c.1, ... in the subsequence.

The tail state of the first transition in a computation is called the initial state of the computation. If a computation is finite, then the head state of the last transition in the computation is called the final state of the computation. If a transition in a computation has a state p (as the tail or head state of that transition), then the computation is said to reach state p.

A state predicate of a system S is a function that has a Boolean value, true or false, at each state of S. Let true denote the state predicate whose value is true at each state of system S.

Let P be a state predicate of a system S. A state of S is called a P-state iff the value of P is true at that state.

Let P and Q be state predicates of a system S. Predicate P equals predicate Q, denoted P = Q, in system S iff P and Q have equal values at every state of S. Predicate P implies predicate Q, denoted P => Q, in system S iff, for every state p of system S, if P is true at p, then Q is true at p. Note that P = Q in S iff P => Q and Q => P in S.

Let n be an integer whose value is at least 1. An n-phase system consists of a system S and a partitioning of the actions of S into n nonempty partitions, named phase 1 through phase n. Note that any system S can be made 1-phase by grouping all the actions of S into one phase.

Let S be an n-phase system and let P be a state predicate of S. Predicate P is called j-closed in S, where $1 \leq j \leq n$, iff, for each transition (p, c, q) of system S, if p is a P-state and c is an action of phase j or less, then q is a P-state.

Lemma 1. *For any n-phase system S,*

1. *the state predicate true is n-closed in S and*
2. *any k-closed predicate in S is also j-closed, where* $1 \leq j \leq k \leq n$.

Let S be an n-phase system and let P and Q be two state predicates of S. System S is j-convergent from P to Q, where $1 \leq j \leq n$, iff the following two conditions hold:

1. **Reaching from P to Q**: For each P-state p, p is a Q-state or system S has a computation whose initial state is p and whose actions are of phase j or less and each such computation of S reaches a Q-state.
2. **Closure of Q**: Predicate Q is j-closed in S.

An n-phase system S is n-stabilizing to a state predicate Q iff system S has n+1 state predicates P.0, P.1, ..., P.n such that the following two conditions hold:

1. **Boundary**: P.0 = true and P.n = Q.
2. **Convergence**: For every j, $1 \leq j \leq n$, S is j-convergent from P.(j-1) to P.j.

In the special case where n = 1, the definitions of 1-closure, 1-convergence, and 1-stabilization degenerate to the traditional definitions of closure, convergence, and stabilization, as given, for example, in [8].

Lemma 2. *For any 1-phase system S, S is 1-convergent from true to Q iff S is 1-stabilizing to Q.*

3 Multiphase Recovery

In this section, we describe a fault recovery method for multiphase stabilizing systems. This method is called multiphase recovery.

Let S be an arbitrary n-phase system. In general, the computations of S can be partitioned into fault computations and fault-free computations. Let Q be an n-closed state predicate of system S and assume that Q defines all states that are reachable in every fault-free computation of S. In other words, each fault-free state of S is a Q-state and vice versa. The occurrence of one or more errors can lead system S to a fault state (that is, not a Q-state). If system S is n-stabilizing to Q, then a recovery procedure can be invoked periodically to ensure that if S ever reaches a fault state, then, eventually, S will return to a fault-free state. This recovery procedure is described next.

Because system S is n-stabilizing to Q, S has n + 1 state predicates P.0, P.1, ... , P.n that satisfy the two conditions of boundary and convergence discussed in Section 2. The periodic recovery procedure consists of n consecutive stages. In the jth stage, where $1 \leq j \leq n$, only those S actions of phase j or less are allowed to be executed. Execution of these actions proceeds until system S reaches a (P.j)-state, then the jth stage terminates and the (j + 1)th stage starts. During the nth stage, system S reaches a Q-state and the recovery is complete.

Note that this recovery procedure is invoked periodically, regardless of whether system S is in a fault state. Fortunately, only actions of system S are executed during the recovery procedure. Therefore, if the recovery procedure is invoked when system S is already in a Q-state and if no fault occurs during the recovery, then the ongoing computation of system S continues to be a fault-free computation of the system. Thus, the effect of the recovery procedure remains mostly transparent to any outside observer of the system. The only observable effect of the recovery procedure is a possible increase in the system delay to produce a response because some system actions are prevented from being executed during the recovery.

It is instructive to compare this method of multiphase recovery with the recovery method that uses periodic reset. (Note that we are interested here in resets that are invoked every so many hours rather than resets that are invoked in response to some detected errors, as discussed in [12].) These two methods are invoked periodically to ensure that if a system ever reaches a fault state, then, eventually, that system will return to a fault-free state. However, whereas multiphase recovery is transparent when no fault occurs, a periodic reset is usually intrusive, forcing the system to abandon its current state (even if that state is fault-free) and return to its initial state. The intrusive nature of periodic reset suggests that the time period between two successive resets should be relatively large. On the other hand, the transparent nature of multiphase recovery suggests that the time period between two successive invocations of the recovery procedure can be relatively short. Therefore, multiphase recovery can erase the effects of a fault from a system faster than recovery using periodic resets.

4 Examples of Multiphase Stabilization

In this section, we discuss three examples of multiphase stabilizing systems. These examples illustrate some advantages of the new definition of stabilization over the traditional one.

Example 1. In this example, we discuss a system S that is stabilizing according to the new definition of stabilization, but not according to the traditional definition. Let S be a 2-phase system that consists of r processes s[i : 0..r-1]. Each process s[i] has one Boolean variable st[i] and one action. The action of process s[i] reads variables st[i] and st[i-1 **mod** r], then updates st[i]. The action of process s[0] belongs to phase 2 and the actions of all other processes belong to phase 1. The processes of system S are defined as follows:

```
process s[0]
var     st[0] :        boolean
begin
        st[0] = st[r-1]  -> st[0] := not st[0]       {phase 2}
end

process s[i : 1..r-1]
var     st[i] :            boolean
begin
        st[i] ≠ st[i-1]  -> st[i] := not st[i]       {phase 1}
end
```

Let P and Q be the following state predicates of system S.

$$P = (\forall i, 0 \le i < r, st[i] = st[0]).$$
$$Q = (\exists i, 0 \le i < r, (\forall j, 0 \le j \le i, st[j] = st[0]) \wedge (\forall j, i < j < r, st[j] \ne st[0])).$$

It is straightforward to show that P is 1-closed, Q is 2-closed, S is 1-convergent from true to P, and S is 2-convergent from P to Q. Therefore, system S is 2-stabilizing to Q and multiphase recovery can be used to make S recover from any faults as follows: Periodically, the action of process s[0] is prevented from execution long enough until system S reaches a P-state. Then, when all actions of system S are allowed to be executed, S reaches a Q-state.

Note that if system S was considered 1-phase (rather than 2-phase), then S would not be 1-stabilizing to Q. This shows that S is not stabilizing to Q according to the traditional definition of stabilization, but it is stabilizing to Q according to the new definition. Thus, the new definition of stabilization seems to illuminate some stabilization properties that cannot be captured by the traditional narrower definition.

Example 2. In this example, we show that the well-known alternating-bit protocol is stabilizing according to the new definition of stabilization, but not according to the traditional definition.

Consider an alternating-bit protocol S that consists of two processes s and r. Process s sends data messages to process r, which replies by sending back an ack message for every data message it receives. Each data message sent by s is stored in a channel from s to r until the message is finally received by r or lost. Similarly, each ack message sent by r is stored in a channel from r to s until the message is finally received by s or lost.

Each of the two channels stores the sequence of messages that are sent into the channel but not yet received or lost from the channel. Sending a message into a channel consists of adding the message at the tail of the message sequence in the channel. Receiving a message from a channel consists of removing the head message from the message sequence in the channel. Losing a message from a channel consists of removing any message in the message sequence in the channel. For example, sending a message m into a channel whose message sequence is "m0; m1; m2" makes the sequence "m0; m1; m2; m." Also, receiving a message from a channel whose message sequence is "m0; m1; m2" makes the sequence "m1; m2." Finally, losing message m1 from a channel whose message sequence is "m0; m1; m2" makes the sequence "m0; m2."

In protocol S, process s has a Boolean variable bs and process r has a Boolean variable br. Each data message sent by s is of the form data(b), where b is the value of variable bs when the message is sent.

When r receives a data(b) message, r compares the current value of variable br with b. If br = b, then r recognizes that the received data(b) message is the expected one. In this case, r stores the message, changes the value of br, and sends back an ack(b) message to s. If $br \neq b$, then r recognizes that the received data(b) message is another copy of the last received message. In this case, r discards the message, keeps the value of br unchanged, but still sends back an ack(b) message to s (because the last ack(b) message may have been lost during transmission from r to s).

When s receives an ack(b) message, s compares the current value of variable bs with b. If bs = b, then s recognizes that the received ack(b) message is the expected acknowledgment for the last data(b) message sent by s. In this case, s changes the value of bs and starts to send the next data(bs) message to r. If $bs \neq b$, then s recognizes that the received ack(b) message is not the expected one. In this case, s keeps the value of bs unchanged and continues to resend the last data(b) message (because earlier copies of that message may have been lost during transmission from s to r).

Processes s and r in the alternating-bit protocol S are as follows:

```
process s
var  bs, b :        boolean
begin
        true              -> send data(bs) to r {phase 2}
[]      rcv ack(b) from r -> if  bs ≠ b  -> skip
                             []  bs = b  -> bs := not bs
                             fi                  {phase 1}
end
```

```
process r
var   br, b :        boolean
begin
      rcv data(b) from r   -> if br ≠ b  ->
                                 {discard data(b)} skip
                             [] br = b  ->
                                 {store data(b)}
                                 bs := not bs
                             fi; send ack(b) to s
                                              {phase 1}
end
```

Note that the first action of process s belongs to phase 2 and the second action of process s and the action of process r belong to phase 1.

Let ch.s.r denote the sequence of data messages in the channel from process s to process r and let ch.r.s denote the sequence of ack messages in the channel from process r to process s. Then, ch.s.r and ch.r.s can be used in defining the state predicates P and Q of protocol S as follows:

$$P = (\text{ch.s.r = empty sequence}) \wedge (\text{ch.r.s = empty sequence}).$$

$$Q = (\text{bs} = \text{br} \wedge \text{ch.s.r} = (\text{data}(\textbf{not}\ \text{bs}))^*; (\text{data(bs)})^* \wedge \text{ch.r.s} = (\text{ack}(\textbf{not}\text{bs}))^*) \wedge (\text{bs} \neq \text{br} \wedge \text{ch.s.r} = (\text{data(bs)})^* \wedge \text{ch.r.s} = (\text{ack}(\textbf{not}\ \text{bs}))^*; (\text{ack(bs)})^*).$$

It is straightforward to show that P is 1-closed, Q is 2-closed, S is 1-convergent from true to P, and S is 2-convergent from P to Q. Therefore, protocol S is 2-stabilizing to Q and multiphase recovery can be used to make S recover from any faults as follows: Periodically, the first action of process s is prevented from execution long enough to ensure that protocol S reaches a P-state. Then, all actions of protocol S are allowed to be executed and S reaches a Q-state.

Note that if protocol S was considered 1-phase, then S would not be stabilizing to Q. This shows that S is not stabilizing to Q according to the traditional definition of stabilization. It is shown in [10] that any network protocol whose variables have bounded domains cannot be stabilizing according to the traditional definition of stabilization. However, as illustrated by Example 2, this negative result does not extend to multiphase stabilization.

Example 3. In this example, we discuss a system S that is stabilizing according to both the new definition of stabilization and the traditional definition. However, we argue that proving stabilization of S according to the new definition is easier than proving it according to the traditional definition.

Let S be a 3-phase system that consists of r processes s[i : 0..r-1]. There is a one-to-one correspondence between the processes of system S and the vertices of a connected and undirected graph G. For every i, where $0 \leq i < n$, let v[i] denote the vertex in graph G that corresponds to process s[i] in system S. Two processes s[i] and s[j] are neighbors in S iff there is an edge between their corresponding vertices v[i] and v[j] in G. The actions of a process s[i] can read the variables of another process s[j], only if s[i] and s[j] are neighbors.

The processes of system S maintain a rooted spanning tree whose root is process s[0]. In particular, each process s[i] has two variables ds[i] and pr[i]. Variable ds[i] stores the number of edges for reaching s[0] from s[i] over the spanning tree and variable pr[i] stores the index of the parent of s[i] in the tree. Note that the parent of a process s[i] is one of the neighbors of s[i]. For process s[0], the two variables ds[0] and pr[0] have fixed values, namely, 0 and 0. Thus, process s[0] has no actions (for updating the variables ds[0] and pr[0]). Each other process s[i] has three actions. In each action, s[i] reads its own two variables and the two variables of a neighbor, then updates its own variables. The processes of system S are defined as follows:

```
process s[0]
var   ds[0] :   0..0,
      pr[0] :   0..0
begin {there are no actions in s[0]}
end

process s[i : 1..r-1]
var   ds[i] :   0..n,
      pr[i] :   index of the parent of s[i] in the
                spanning tree
par   g     :   index of an arbitrary neighbor of s[i]
begin
      ds[i] < n ∨ ds[pr[i]] < n ∨ ds[i] ≠ ds[pr[i]]+1
                         -> ds[i] := ds[pr[i]]+1 {phase 1}
[]    ds[i] < n ∨ ds[pr[i]] = n  ->  ds[i] := n  {phase 2}
[]    ds[i] = n ∨ ds[g] < n-1    ->  ds[i] := ds[g]+1;
                                     pr[i] := g  {phase 3}
end
```

Note that, in the third action, process s[i] detects that ds[i] = n and there is a neighbor s[g] where ds[g] < n-1. In this case, s[i] makes s[g] its parent in the tree. Note also that, in each process, the first action belongs to phase 1, the second action belongs to phase 2, and the third action belongs to phase 3.

Let P, P′, and Q be the following state predicates of system S.

$$P = (\forall i, 1 \leq i < r,\ \text{ds}[i] = n \vee \text{ds}[\text{pr}[i]] = n \vee \text{ds}[i] = \text{ds}[\text{pr}[i]] + 1).$$
$$P' = (\forall i, 1 \leq i < r,\ \text{ds}[i] = n \vee \text{ds}[i] = \text{ds}[\text{pr}[i]] + 1).$$
$$Q = (\forall i, 1 \leq i < r,\ \text{ds}[i] = \text{ds}[\text{pr}[i]] + 1).$$

It is straightforward to show that P is 1-closed, P′ is 2-closed, and Q is 3-closed. It is also straightforward to show that S is 1-convergent from true to P, S is 2-convergent from P to P′, and S is 3-convergent from P′ to Q. Thus, system S is 3-stabilizing to Q and multiphase recovery can be used to make S recover from errors as follows: Periodically, the second and third actions in every process in S are prevented from execution long enough until S reaches a P-state. Then, the second actions in all processes in S are allowed to be executed, but the third actions are still prevented from execution until S reaches a P′-state. Finally, all actions of system S are allowed to be executed, causing S to reach a Q-state.

Note that if system S is considered 1-phase, then S is 1-stabilizing to Q [5]. However, proving stabilization when S is 3-phase is easier than when S is 1-phase for the following reason: If S is 3-phase, then proving 3-phase stabilization of S consists of proving the following three assertions:

- S is 1-convergent from true to P,
- S is 2-convergent from P to P′, and
- S is 3-convergent from P′ to Q.

On the other hand, if S is 1-phase, then proving 1-phase stabilization of S consists of proving only one assertion:

- S is convergent from true to Q.

Proving this one assertion is harder than proving all three former assertions. Thus, choosing S to be 3-phase, rather 1-phase, has the effect of partitioning one complex proof obligation into three simple ones.

5 Theorems of Multiphase Stabilization

In this section, we present some theorems that follow from the definitions of closure, convergence, and stabilization in Section 2. As discussed in [8], similar theorems also follow from the traditional definitions of closure, convergence, and stabilization. This shows that the new definitions inherit many of the nice features of the traditional definitions.

In the following theorems, let S be any n-phase system, and let P, P′, Q, and Q′ be state predicates of S. Also, let j and k be any two integers, where $1 \le j \le k \le n$.

Base Theorem. *The state predicate true is j-closed in S, system S is j-convergent from true to true, and system S is n-stabilizing to true.*

Junctivity of Closure Theorem. *If P is j-closed in S and Q is k-closed in S, then $P \wedge Q$ and $P \wedge Q$ are j-closed in S.*

Junctivity of Convergence Theorem. *If S is j-convergent from P to Q and S is j-convergent from P′ to Q′, then S is j-convergent from $P \vee P'$ to $Q \vee Q'$ and S is j-convergent from $P \wedge P'$ to $Q \wedge Q'$ (provided $Q \wedge Q'$ is not false).*

Junctivity of Stabilization Theorem. *If S is n-stabilizing to P and S is n-stabilizing to Q, then S is n-stabilizing to $P \vee Q$ and S is n-stabilizing to $P \wedge Q$. (provided $P \wedge Q$ is not false).*

From Closure to Convergence Theorem. *If Q is j-closed in S and P => Q in S, then S is j-convergent from P to Q.*

From Convergence to Convergence Theorem: (Transitivity of Convergence). *If S is j-convergent from P to P′ and S is j-convergent from P′ to Q, then S is j-convergent from P to Q.*

From Convergence to Stabilization Theorem. *If S is n-convergent from P to Q and S is n-stabilizing to P, then S is n-stabilizing to Q.*

Weakening of Convergence Theorem. *If S is j-convergent from P to Q, P′ => P in S, Q => Q′ in S, and Q′ is j-closed in S, then S is j-convergent from P′ to Q′*

Weakening of Stabilization Theorem. *If S is n-stabilizing to Q, Q => Q′ in S, and Q′ is n-closed in S, then S is n-stabilizing to Q′.*

6 Hierarchies of Multiphase Stabilizing Systems

In this section, we show that the new definition of stabilization, like the traditional one, supports system compositions that preserve stabilization. In particular, we show that it is possible to combine a system S that is m-stabilizing to Q with a system T that is n-stabilizing to R such that the resulting system is (m + n)-stabilizing to $Q \wedge R$.

Consider the 3-phase system S in Example 3 above. In system S, the processes s[i : 0..r-1] maintain a rooted spanning tree whose root is process s[0]. The spanning tree is maintained as each process s[i] stores the index of its parent in the tree in a variable pr[i]. Recall that system S is 3-stabilizing to the state following predicate Q.

$$Q \;=\; (\forall i, 1 \le i < r,\;\; ds[i] = ds[pr[i]] + 1).$$

Now, consider a 2-phase system T that consists of the processes t[i : 0..r-1]. Like system S, there is a one-to-one correspondence between the processes of system T and the vertices of a connected and undirected graph G. For every i, where $0 \le i < r$, let v[i] denote the vertex in graph G that corresponds to process t[i] in system T. Two processes t[i] and t[j] are neighbors in T iff there is an edge between their corresponding vertices v[i] and v[j] in G. The actions of a process t[i] can read the variables of another process t[j] only if t[i] and t[j] are neighbors.

The processes of system T are arranged in a rooted spanning tree whose root is process t[0]. Each process t[i] has a variable pr[i] that stores the index of the parent of t[i] in the tree. The processes of system T use this spanning tree to propagate consecutive sequence numbers from process t[0] to every other process in T. The current sequence number of each process t[i] is stored in an integer variable sq[i]. When t[0] detects that its current sequence number in sq[0] equals that of each of its children in the tree, t[0] increments the value of sq[0] by one. When any other t[i] detects that its current sequence number in sq[i] equals that of each of its children in the tree, t[i] assigns to sq[i] the current sequence number of its parent t[pr[i]]. The processes of system T are defined as follows:

```
  process t[0]
  var pr[0] :  0..0,
      sq[0] :  integer
  begin
        (∀t[j] neighbor of t[0],  pr[j] ≠ 0  ∨  sq[j] = sq[0])
                            −>  sq[0] := sq[0] + 1  {phase 2}
end

process t[i : 1..r-1]
var pr[i] :   index of the parent of t[i] in the spanning tree,
    sq[i] :   integer
```

```
begin
        (∀t[j] neighbor of t[i],  pr[j] ≠ i ∨ sq[j] = sq[i]) ∧
        (sq[i] ≠ sq[pr[i]])  -> sq[i] := sq[pr[i]]   {phase 1}
end
```

Note that the action of process t[0] belongs to phase 2 and the actions of all other processes belong to phase 1. Let Q′ and R be the following state predicates of system T.

$$Q' = (\forall i, 1 \le i < r,\ sq[i] = sq[pr[i]]).$$
$$R = (\forall i, 1 \le i < r,\ sq[i] = sq[pr[i]] \lor sq[i] = sq[pr[i]] + 1).$$

It is straightforward to show that Q′ is 1-closed in T, R is 2-closed in T, T is 1-convergent from true to Q′, and T is 2-convergent from Q′ to R. Therefore, system T is 2-stabilizing to R.

The 3-phase system S in Example 3 can be combined with the above 2-phase system T to construct a 5-phase system named ST. Specifically, each process s[i] in system S is combined with the corresponding process t[i] in system T to construct a process st[i] in system ST as follows: First, the variables in process s[i] and the variables in process t[i] are added as variables in process st[i]. Second, each action in s[i] and each action in t[i] is added as an action in st[i]. Third, for each action in s[i], the phase of this action in st[i] equals its phase in s[i] and, for each action in t[i], the phase of this action in st[i] equals its phase in t[i] plus three, which is the number of phases in system S. The processes of system ST are defined as follows:

```
process st[0]
var     ds[0] :  0..0,
        pr[0] :  0..0,
        sq[0] :  integer
begin
        (∀st[j] neighbor of st[0],  pr[j] ≠ 0 ∨ sq[j] = sq[0])
                        -> sq[0] := sq[0] + 1  {phase 5}
end

process st[i : 1..r-1]
var     ds[i] :  0..n-1,
        pr[i] :  index of the parent of st[i] in the
                 spanning tree,
        sq[i] :  integer
par     g     :  index of an arbitrary neighbor of st[i]
begin
        ds[i] < n ∧ ds[pr[i]] < n ∧ ds[i] ≠ ds[pr[i]]+1
                        -> ds[i] := ds[pr[i]] + 1   {phase 1}
[]      ds[i] < n ∧ ds[pr[i]] = n  ->  ds[i] := n {phase 2}
[]      ds[i] = n ∧ ds[g] < n - 1  ->  ds[i] := ds[g] + 1;
                                       pr[i] := g {phase 3}
[]      (∀st[j] neighbor of st[i],  pr[j] ≠ i ∨ sq[j] = sq[i]) ∧
        (sq[i] ≠ sq[pr[i]])  -> sq[i] := sq[pr[i]] {phase 4}
end
```

Because system S is 3-stabilizing to the state predicate Q and system T is 2-stabilizing to the state predicate R, it is straightforward to show that the constructed system ST is 5-stabilizing to the state predicate Q∧R.

$$Q \land R = (\forall i, 1 \le i < r,\ ds[i] = ds[pr[i]] + 1) \land$$
$$(\forall i, 1 \le i < r, sq[i] = sq[pr[i]] \lor sq[i] = sq[pr[i]] + 1).$$

7 Transforming Multiphase Stabilization to Single-Phase Stabilization

In this section, we discuss how to transform any system that is n-stabilizing, where $n \ge 2$, to one that is 1-stabilizing. The transformation consists of four steps. In each step, we add variables and actions to the system processes to accomplish a certain task. The tasks that are accomplished in these four steps are as follows:

1. **First Step**. Make the processes maintain a rooted spanning tree.
2. **Second Step**. Make the processes propagate sequence numbers in the range 0..3 along the spanning tree in a cyclic fashion. In each cycle, the sequence numbers are propagated as follows: First, sequence number 0 is propagated from the root to the leaves. Second, sequence number 1 is propagated from the leaves to the root. Third, sequence number 2 is propagated from the root to the leaves. Finally, sequence number 3 is propagated from the leaves to the root.
3. **Third Step**. Make the root of the spanning tree use the propagated sequence numbers to detect whether execution of the original actions in the system has terminated.
4. **Fourth Step**. Make the processes execute the original actions according to their phases and make the root of the spanning tree detect termination of the current phase and initiate execution of the next phase.

To describe these four steps in more detail, consider the following n-stabilizing system, where $n \ge 2$.

```
process f[i : 0..r-1]
var   u[i] :          <type of u[i]>,
      ...
      v[i] :          <type of v[i]>
begin
      G.i.1   ->      S.i.1              {phase 1}
[]    G.i.2   ->      S.i.2              {phase 2}
...
[]    G.i.n   ->      S.i.n              {phase n}
end
```

For simplicity, we assume that each process in this system has n actions: one action in each phase. We also assume that each system computation whose actions are of phase n - 1 or less is finite. Next, we describe how to apply the above four steps to transform this n-stabilizing system to a 1-stabilizing system.

First Step. Referring to Example 3 in Section 4, the processes s[i : 0..r-1] maintain a rooted spanning tree, whose root is s[0]. To maintain this tree, each process s[i] has two variables ds[i] and pr[i] and three actions. (The only exception is that s[0] has no actions.) Thus, to make the processes f[i : 0..r-1] maintain a spanning tree whose root is f[0], add to each process f[i] two variables ds[i] and pr[i] and the same actions as those in the corresponding process s[i]. However, unlike the actions in s[i], which were assumed to be partitioned into three phases, the added actions to f[i] are assumed to belong to only one phase, namely phase 1.

The added variables and actions make the processes f[i : 0..r-1] maintain a rooted spanning tree whose root is f[0]. Moreover, the tree maintenance system is 1-stabilizing [5].

Second Step. In order to make the processes in f[i : 0..r-1] propagate sequence numbers over the maintained spanning tree, add to each process f[i] the following variable

```
var   sq[i] :   0..3
```

and the following four actions:

```
[] sq[i] =  0 ∧ (∀f[j] neighbor of f[i],  pr[j] ≠ i  ∨  sq[j] = 1)
             ->    sq[i] := 1; X
[] sq[i] =  1 ∧ (i = 0 ∨  sq[pr[i]] = 2)    ->    sq[i] := 2
[] sq[i] =  2 ∧ (∀f[j] neighbor of f[i],  pr[j] ≠ i  ∨  sq[j] = 3)
              ->    sq[i] := 3; Y
[] sq[i] =  3 ∧ (i = 0 ∨  sq[pr[i]] = 0)    ->    sq[i] := 0; Z
```

All added actions are assumed to belong to phase 1.

Note that the added actions to process f[i] refer to the variables pr[i] in f[i] and pr[j] in every neighboring f[j]. These variables were added to f[i : 0..r-1] in the first step of the transformation. Note also that the first added action has a statement X, the third action has a statement Y, and the fourth action has a statement Z. For now, we assume that each of these is a **skip** statement and, so, has no effect on action executions. (These statements are redefined below.)

It is shown in [11] that any execution of the added actions leads to a state satisfying the predicate (∀i, i = 0..r-1, sq[i] = 3). Once this state is reached, sequence number 0 is propagated (by the fourth action) from the root to the leaves, then sequence number 1 is propagated (by the first action) from the leaves to the root, then sequence number 2 is propagated (by the second action) from the root to the leaves, then sequence number 3 is propagated (by the third action) from the leaves to the root, and the cycle repeats.

Third Step. In order to make the root f[0] detect whether execution of the original actions in f[i : 0..r-1] has terminated, modify each process f[i] as follows: First, add to process f[i] the following variable:

```
var   tr[i] :   boolean  {termination detected}.
```

Second, modify the statement "S.i.k" in each original action of process f[i] to become "S.i.k; tr[i] := **false**," where k = 1..n. Third, redefine the two statements X and Y (that were introduced in the second step) in f[i] to become as follows:

```
X:  tr[i]  :=    true
Y:  tr[i]  :=    (tr[i]  ∧
                 (not G.i.1 ∧ not G.i.2 ∧ ... ∧ not G.i.n)  ∧
                 (∀f[j] neighbor of f[i],  pr[j] ≠  i  ∨  tr[j])
                 ).
```

Recall that G.i.1, G.i.2, ... , G.i.n are the guards of the original actions of process f[i].

Because of the second modification, the added variable tr[i] is assigned the value false whenever process f[i] executes one of its original actions. Because of the third modification, variable tr[i] is assigned the value true whenever process f[i] propagates the sequence number 1. Also, variable tr[i] is assigned a termination indication value whenever f[i] propagates the sequence number 3. The assigned termination indication value is true iff the following three conditions hold.

1. The current value of tr[i] is already true. This indicates that process f[i] has not executed any original action during the time period after f[i] propagated the sequence number 1 and until it propagated the sequence number 3.
2. No original action of process f[i] is currently enabled.
3. For every child f[j] of process f[i] in the spanning tree, the current value of variable tr[j] is true.

Therefore, the termination indication value assigned to variable tr[0] is true iff no original action in the f[i : 0..r-1] system is enabled.

Fourth Step. In order to make the root f[0] detect termination of the current phase and initiate execution of the next phase, modify each process f[i] as follows: First, add to process f[i] the following variable:

```
var   ph[i] :     1..n    {current phase of f[i]}.
```

Also add to process f[0] the following variable:

```
var    t :     0..tmax  {time spent in phase n}.
```

Second, modify the guard "G.i.k" in each original action of process f[i] to become "G.i.k ∨ ph[i] ≥ k," where k = 1..n. Third, redefine the two statements Y and Z (that were introduced in the second step) in f[i] to become as follows:

```
Y:  tr[i] :=    (tr[i]  ∧
                (not (G.i.1 ph[i] ≥ 1) ∧ ... ∧ not (G.i.(n - 1)
                ∧ ph[i] ≥ n - 1)) ∧
                (∀f[j] neighbor of f[i],  pr[j] ≠  i  ∨  tr[j])
                )


Z:  if    i = 0 ∧ (ph[i] < n ∧ not tr[i])   ->    skip
    []    i = 0 ∧ (ph[i] < n ∧       tr[i])   ->
                                           ph[i] := ph[i]+ 1; t := 0
    []    i = 0 ∧ (ph[i] = n t < tmax)     ->    t := t + 1
    []    i = 0 ∧ (ph[i] = n t = tmax)     ->    ph[i] := 1
    []    i ≠  0                            ->    ph[i] := ph[pr[i]]
    fi
```

Because of the second modification, each process f[i] cannot execute its kth original action until the value of its variable ph[i] is at least k. Because of the third modification, whenever process f[i] propagates the sequence number 0, it executes statement Z whose effect depends on the value of i. Executing statement Z in process f[0] has the following effect: Process f[0] checks whether the current phase is less than n and whether it has terminated. If so, f[0] assigns variable ph[0] to its next value to initiate the next phase. If the current phase is n, then f[0] keeps this phase for tmax cycles before changing the current phase to 1. Executing statement Z in every other process f[i] has the following effect: Process f[i] assigns its variables ph[i] the value of ph[j], where f[j] is the parent of f[i] in the spanning tree.

The cost of applying this four-step transformation is not overwhelming. This transformation adds five variables (namely, ds[i], pr[i], sq[i], tr[i], and ph[i]) to each process f[i] in the system and adds a sixth variable (namely, t) to process f[0]. It also adds seven actions to each process f[i : 1..r-1] and adds four actions to process f[0].

8 Concluding Remarks

In this paper, we presented a new definition of system stabilization, namely multiphase stabilization, and showed that this definition is a proper generalization of the traditional definition of single-phase stabilization. We argued that it is easier to verify the stabilization properties of n-phase systems when n is large than when n is small. We also showed that each n-stabilizing system can be transformed into a 1-stabilizing system. The proposed transformation is reasonable, requiring that each process in the system be enhanced by no more than six variables and seven actions.

It is instructive to compare multiphase stabilization with the convergence stairs presented in [10]. In both multiphase stabilization and a convergence stair, the convergence of a system (from any state to a specified state) occurs at stages. However, in the case of multiphase stabilization, different phases of actions are allowed to execute at each stage, whereas, in the case of a convergence stair, all actions are allowed to execute at each stage and, so, convergence is harder to achieve (and verify) in this case.

It is argued in [2] that the actions of many (1-) stabilizing systems can be partitioned into closure actions and convergence actions. Clearly, each such system can be viewed as a 2-stabilizing system where the convergence actions belong to phase 1 and the closure actions belong to phase 2.

Two issues that we have not addresses in this paper concerning n-stabilizing systems are how to design such systems, especially when n is relatively large, and what is the effect of n on the convergence time of such systems. We believe that these issues are important and merit investigation.

Acknowledgments

This work is supported in part by the DARPA contract F33615-01-C-1901 from the Defense Advanced Research Projects Agency. It is also supported in part by two IBM Faculty Partnership Awards for the academic years of 2000-2001 and 2001-2002, and by a grant from the Texas Advanced Research Program provided by the Texas Higher Education Coordinating Board.

References

[1] A. Arora and M.G. Gouda, "Closure and Convergence: A Foundation for Fault-Tolerant Computing," *IEEE Trans. Software Eng.*, vol. 19, no. 3, pp. 1015-1027, Mar. 1993.
[2] A. Arora, M.G. Gouda, and G. Varghese, "Constraint Satisfaction as a Basis for Designing Nonmasking Fault-Tolerance," *J. High-Speed Networks*, vol. 5, no. 3, pp. 293-306, 1996.
[3] Y. Afek and A. Bremler, "Self-Stabilizing Unidirectional Network Algorithms by Power Supply," *Chicago J. Theoretical Computer Science*, vol. 4, no. 3, pp. 1-48, 1998.
[4] A. Bui, A.K. Datta, F. Petit, and V. Villain, "State-Optimal Snap-Stabilizing PIF in Tree Networks," *Proc. Third Workshop Self-Stabilizing Systems*, pp. 78-85, 1999.
[5] N.S. Chen, F.P. Yu, and S.T. Huang, "A Self-Stabilizing Algorithm for Constructing Spanning Trees," *Information Processing Letters*, vol. 39, pp. 147-151, 1991.
[6] S. Dolev and T. Herman, "Super Stabilizing Protocols for Dynamic Distributed Systems," *Proc. Second Workshop Self-Stabilizing Systems*, published as a technical report, Dept. of Computer Science, Univ. of Nevada, Las Vegas, pp. 3.1-3.15, 1995.
[7] S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju, "Fault-Containing Self-Stabilizing Algorithms," *Proc. ACM Symp. Principles of Distributed Computing*, pp. 45-54, 1996.
[8] M.G. Gouda, "The Triumph and Tribulation of System Stabilization," *Invited Paper, Proc. Int'l Workshop Distributed Algorithms*, J.M. Helary and M. Raynal eds., pp. 1-18, 1995.
[9] M.G. Gouda and T. Herman, "Adaptive Programming," *IEEE Trans. Software Eng.*, vol. 17, no. 9, pp. 911-921, Sept. 1991.
[10] M.G. Gouda and N. Multari, "Stabilizing Communication Protocols," *IEEE Trans. Computers*, vol. 40, no. 4, pp. 448-458, Apr. 1991.
[11] F.F. Haddix, "Alternating Parallelism and the Stabilization of Cellular Systems," PhD dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin, 1999.
[12] S. Katz and K.J. Perry, "Self-Stabilizing Extensions for Message Passing Systems," *Distributed Computing*, vol. 7, pp. 17-26, 1993.
[13] S. Kutten and B. Patt-Shamir, "Stabilizing Time-Adaptive Protocols," *Theoretical Computer Science*, vol. 220, pp. 93-111, Year?

Mohamed G. Gouda received BSc degrees in both engineering and mathematics, both from Cairo University, Egypt. Later, he obtained the MA degree in mathematics from York University, Toronto, Canada, and Masters and PhD degrees in computer science from the University of Waterloo, Ontario, Canada. He worked for the Honeywell Corporate Technology Center in Minneapolis from 1977-1980. In 1980, he joined the University of Texas at Austin, where he currently holds the Mike A. Myers Centennial Professorship in Computer Sciences. He spent one summer at Bell Labs in Murray Hill, New Jersey, one summer at the Microelectronics and Computer Technology Corporation in Austin, and one winter at the Eindhoven Technical University in the Netherlands. His research areas are distributed and concurrent computing and network protocols. In these areas, he has been working on abstraction, formality, correctness, nondeterminism, atomicity, reliability, security, convergence, and stabilization. He has published more than 50 journal papers and more than 170 conference papers. He supervised more than 17 PhD dissertations. Professor Gouda was the founding editor-in-chief of the Springer-Verlag journal *Distributed Computing* from 1985-1989. He served on the editorial board of *Information Sciences* from 1996-1999 and he is currently on the editorial boards of *Distributed Computing* and the *Journal of High Speed Networks*. He was the program committee chairman of ACM SIGCOMM Symposium in 1989. He was the first program committee chairman of the IEEE International Conference on Network Protocols in 1993. He was the first program committee chairman of the IEEE Symposium on Advances in Computers and Communications, which was held in Egypt in 1995. He was the program committee chairman of the IEEE International Conference on Distributed Computing Systems in 1999. He is on the steering committee of the IEEE International Conference on Network Protocols and is an original member of the Austin Tuesday Afternoon Club. Professor Gouda is the author of the textbook *Elements of Network Protocol Design*, published by John Wiley & Sons in 1998. This is the first ever textbook where network protocols are presented in abstract and formal setting. Currently, he is writing the textbook *Elements of Secure Network Protocols*. Professor Gouda is the 1993 winner of the Kuwait Award in Basic Sciences. He was the recipient of an IBM Faculty Partnership Award for the academic year 2000-2001 and again for the academic year 2001-2002. He won the 2001 IEEE Communication Society William R. Bennet Best Paper Award for his paper "Secure Group Communications Using Key Graphs," coauthored with C.K. Wong and S.S. Lam and published in the February 2000 issue of the *IEEE/ACM Transactions on Networking* (vol. 8, no. 1, pp. 16-30).

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.