Distrib. Comput. (2007) 20:21–28
DOI 10.1007/s00446-007-0033-1

The alternator

Mohamed G. Gouda · F. Furman Haddix

Received: 28 August 1999 / Accepted: 5 July 2000 / Published online: 12 June 2007

Abstract An alternator is an array of interacting processes that satisfy three conditions. First, if a process has an enabled action at some state, then no neighbor of that process has an enabled action at the same state. Second, along any concurrent execution, each action is executed infinitely often. Third, along any maximally concurrent execution, the alternator is stabilizing to states where the number of enabled actions is maximal. In this paper, we specify an alternator with arbitrary topology and verify its correctness. We also show that this alternator can be used in transforming any system that is stabilizing assuming serial execution, to one that is stabilizing assuming concurrent execution.

Keywords Self-stabilization · Distributed systems · System transformation · Serial execution · Concurrency

1 Introduction

In proving that a system is stabilizing, one needs to choose between the following two assumptions concerning the execution of enabled actions.

(i) Serial execution: Enabled actions are executed one at a time.
(ii) Concurrent execution: Any nonempty subset of enabled actions is executed at a time.

M. G. Gouda (✉)
The University of Texas at Austin, Austin, TX, USA
e-mail: gouda@cs.utexas.edu

F. F. Haddix
Texas State University–San Marcos, San Marcos, TX, USA
e-mail: fh10@txstate.edu

Because it is easier to prove stabilization under the assumption of serial execution, most stabilizing systems are proved correct under that assumption. Unfortunately, many practical implementations of stabilizing systems can support concurrent execution of actions but not serial execution. This situation raises an important question. Given that a system is stabilizing under the assumption of serial execution, is the system stabilizing under the assumption of concurrent execution?

The answer to this question is "no" in general. For example, consider a system that consists of two boolean variables b and c and two actions defined as follows.

$b \neq c \rightarrow b := c$
$b \neq c \rightarrow c := b$

Assuming a serial execution of the actions, this system is stabilizing to a state where b = c. However, if the two actions are always executed simultaneously, then this system will stay in states where $b \neq c$ indefinitely.

To deal with this negative result, two approaches have emerged. The first approach is to identify special classes of systems that satisfy the following property: If a system in any of these classes is stabilizing under the assumption of serial execution, then the same system is also stabilizing under the assumption of concurrent execution. Examples of this approach are reported in [1] and [2].

The second approach to deal with this negative result is to identify system transformations that satisfy the following property. If a system is stabilizing under the assumption of serial execution, then the identified transformation can be used to transform this system to one that is stabilizing under the assumption of concurrent execution. For example, consider the above system that is stabilizing to states where b = c under serial execution, but not under concurrent execution.

This system can be transformed by adding a third boolean variable x and modifying the two actions of the system to become as follows.

$b \neq c \wedge x \rightarrow b := c$

$b \neq c \wedge \textbf{not}\ x \rightarrow c := b$

The transformed system is stabilizing to states where b = c under concurrent execution.

Examples of such system transformations are reported in [4,6], and [7] and in the current paper. The transformation in [6] ensures that each concurrent execution of actions is serializable, as defined in the literature of database systems, and so each concurrent execution is equivalent to a serial execution. The transformation in [4] ensures that during any concurrent execution, conflicting actions (those that read and write common variables) are not executed simultaneously. Unfortunately, this transformation is applicable only to systems with linear topologies. Two generalizations of this transformation (that are applicable to systems with arbitrary topologies) are described in [7] and the current paper. The generalization in [7] uses unbounded variables, but achieves the nice property of silent stabilization, as characterized in [3]. By contrast, the generalization in the current paper uses bounded variables but its stabilization property is not silent.

The transformation in the current paper is based on a class of systems called alternators. We specify the alternator in Sect. 2, and discuss its properties in Sect. 3. Then in Sect. 4, we show how to use the alternator to transform any system (with arbitrary topology) that is stabilizing under the assumption of serial execution to one that is stabilizing under concurrent execution. In Sect. 5, we give an example of a system that is stabilizing assuming serial execution and not stabilizing assuming concurrent execution. We show how to use the alternator to transform this system to one that is stabilizing assuming concurrent execution. In Sect. 6, we present a low-atomicity version of the alternator and discuss its properties. Concluding remarks are presented in Sect. 7.

2 Specification of the alternator

An alternator is an array of processes $p[i : 0 \ldots n-1]$ that are defined below. The topology of an alternator is a connected undirected graph where each node represents one process in the alternator. Two processes in an alternator are called neighbors iff there is an (undirected) edge between the two nodes representing the processes in the topology of the alternator.

The cyclic distance d of an alternator is defined as follows. If the topology of the alternator has no cycles, then d = 2. Otherwise, d is the number of edges in the longest simple cycle in the topology of the alternator. Note that if the topology of an alternator has cycles (of length 3 or more), then $d \geq 3$.

Each process p[i] in an alternator has one variable v[i] and one action. The value of variable v[i] is in the range $0 \ldots 2d-1$, where d is the cyclic distance of the alternator. The action of process p[i] is of the form

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

where $<$ guard $>$ is a boolean expression over variable v[i] of p[i] and those of the neighbors of p[i], and $<$ statement $>$ is an assignment statement of the form v[i] := v[i] + 1 **mod** 2d. A process p[i] in an alternator is defined as follows.

```
process p[i : 0..n–1]                    {comment: n≥ 2}
const   d     :     integer              {cyclic distance of the alternator}
var     v[i]  :     0..2d–1
begin
      (For every j, where p[j] is a neighbor of p[i],
            (v[i] ≠ v[j]    ∨    i < j)   ∧
            (v[i] ≠ v[j] – 1 mod 2d)
      )
            →     v[i] := v[i] + 1 mod 2d
end
```

A state of an alternator is an assignment of a value, from the range $0 \ldots 2d-1$, to every variable v[i] in the alternator, where d is the cyclic distance of the alternator.

An action of a process in the alternator is enabled at a state s iff the guard of the action is true at state s.

Theorem 1 (Deadlock Freedom) *For each state s of an alternator, the action of at least one process in the alternator is enabled at state s.*

Proof Each state s of the alternator can be represented by a graph G.s as follows.

(i) Each node i in G.s represents process p[i] in the alternator.
(ii) There is a directed edge from node i to node j in G.s iff processes p[i] and p[j] are neighbors in the alternator and the values of their variables v[i] and v[j] satisfy the following condition at state s:

$(v[i] = v[j] \wedge i > j) \vee$
$(v[i] = v[j] - 1 \ \textbf{mod}\ 2d)$

Clearly, the action of a process p[i] in the alternator is enabled at a state s iff node i has no outgoing directed edges in G.s.

Next, we prove that for any state s, G.s has no directed cycles. The proof is by contradiction. Assume that G.s has a directed cycle. Then, there are two cases to consider concerning the values of the v[i] variables around this cycle (and we prove that each of these two cases leads to a contradiction).

Case 1 (Values of the v[i] variables around the cycle are all equal)
In this case, for every two successive nodes i and j in the directed cycle, i > j (because v[i] = v[j] and there is a directed edge from node i to node j). However, this contradicts the fact that the node identifiers cannot always decrease as we traverse the directed cycle.

Case 2 (Values of the v[i] variables around the cycle are unequal)
In this case, there are two successive nodes x and y in the directed cycle such that v[x] $\neq$ v[y]. Moreover, because for any two successive nodes i and j in the directed cycle, v[i] = v[j] or v[i] = v[j] $-$ 1 **mod** 2d, we conclude that v[x] = v[y] $-$ 1 **mod** 2d. However, because the directed cycle has at most d nodes, it is impossible to start at node y, traverse the directed cycle encountering at most d nodes, and return to node x such that for each pair of successively encountered nodes i and j, v[i] = v[j] or v[i] = v[j] $-$ 1 **mod** 2d. Contradiction.

It follows from this discussion that for any state s of the alternator, the graph G.s has no directed cycles. Thus, G.s has at least one node i that does not have outgoing directed edges. Therefore, the action of process p[i] is enabled for execution at state s. □

A concurrent transition (or maximally concurrent transition, respectively) of an alternator is a pair (s, s′) of states such that starting the alternator at state s then executing the statements of one or more actions (or all actions, respectively) that are enabled at s yields the alternator in state s′.

A concurrent computation (or maximally concurrent computation, respectively) of an alternator is an infinite sequence s.0, s.1, ... of states such that each pair (s.i, s.(i + 1)) of consecutive states in the sequence is a concurrent transition (or maximally concurrent transition, respectively).

(Note that Theorem 1 guarantees that each concurrent computation and each maximally concurrent computation is infinite.)

A state s of an alternator is called loose iff for every pair of neighboring processes p[i] and p[j] in the alternator, the values of variables v[i] and v[j] at state s satisfy the following condition.

$$(v[i] \neq v[j]) \wedge (v[i] \neq v[j] - 1 \textbf{ mod } 2d) \wedge (v[i] - 1 \textbf{ mod } 2d \neq v[j])$$

Observe that if an alternator is at a loose state s, then the action of every process in the alternator is enabled at s. Observe also that a maximally concurrent transition (where all enabled actions are executed simultaneously) starting from a loose state yields the alternator in another loose state. From these observations, all states in a maximally concurrent computation that starts at a loose state are loose.

3 Properties of the alternator

In this section, we show that an alternator satisfies the following three interesting properties.

(i) Safety: At each state s of the alternator, if the actions of two neighboring processes p[i] and p[j] are enabled at s, then v[i] $\neq$ 2d $-$ 1 or v[j] $\neq$ 2d $-$ 1 at s.
(ii) Progress: Along each concurrent computation of the alternator, the action of each process is executed infinitely often.
(iii) Stabilization: Each maximally concurrent computation of the alternator has an infinite suffix where each state is loose.

The property of safety needs some explanation. As discussed in Sect. 4, the action of each process p[i] performs effective work only when its execution starts while v[i] = 2d $-$ 1. Thus, the property of safety simply states that the actions of neighboring processes cannot perform effective work simultaneously. In other words, the effective work performed by the action of one process cannot adversely interfere with the effective work performed by the action of a neighboring process.

Theorem 2 (Safety) *At each state s of the alternator, if the actions of two neighboring processes p[i] and p[j] are enabled at s, then v[i] $\neq$ 2d $-$ 1 or v[j] $\neq$ 2d $-$ 1 at s.*

Proof Let s be a state of the alternator, and let p[i] and p[j] be any two neighboring processes where the values of variables v[i] and v[j] satisfy the following condition at state s: v[i] = 2d $-$ 1 and v[j] = 2d $-$ 1. Assume that the action of process p[i] is enabled for execution at state s. Then, i < j. Therefore, the action of process p[j] is not enabled for execution at state s. □

Theorem 3 (Progress) *Along each concurrent computation of the alternator, the action of each process is executed infinitely often.*

Proof The proof is by contradiction. Assume that along some concurrent computation, the action of some process is not executed infinitely often. By Theorem 1, the action of another process is executed infinitely often along the same computation. Because the topology of the alternator is connected, there are two neighboring processes p[x] and p[y] such that the action of p[x] is not executed infinitely often along the computation, and the action of p[y] is executed infinitely often along the computation. Thus, this computation has an infinite suffix where the action of process p[x] is never executed and the action of process p[y] is executed infinitely often. In other words, along this infinite suffix, the value of v[x] remains fixed, and the value of v[y] is incremented by

one modulo 2d infinitely often. Therefore, at some state s in his infinite suffix, v[y] = v[x] − 1 **mod** 2d. But then, the action of process p[y] cannot be executed after state s in this infinite suffix. Contradiction. □

Theorem 4 (Stabilization) *Each maximally concurrent computation of the alternator has an infinite suffix where each state is loose.*

Proof The proof consists of two steps that are based on the concept of uneven states. A state s of the alternator is called uneven iff for every pair of neighboring processes p[i] and p[j], v[i] ≠ v[j] at state s. In the first step of the proof, we show that every maximally concurrent computation of the alternator has an infinite suffix where each state is uneven. Then in the second step, we show that each maximally concurrent computation of the alternator, where every state is uneven, has an infinite suffix where each state is loose.

First step:
Consider a maximally concurrent computation of the alternator. By Theorem 3, the action of every process is executed infinitely often along this computation. From the program of the alternator, executing the action of a process p[i] ensures that the value of variable v[i] is different from the value of every variable v[j], where p[j] is a neighbor of p[i]. Therefore, every state that occurs in the maximally concurrent computation, after each action is executed at least once, is uneven.

Second step:
Consider a maximally concurrent computation of the alternator where each state is uneven. Each (uneven) state s in this computation can be represented by a graph G.s (similar to that in the proof of Theorem 1 as follows.

(i) Each node i in G.s represents process p[i] in the alternator.
(ii) There is a directed edge from node i to node j in G.s iff processes p[i] and p[j] are neighbors in the alternator and the values of their variables v[i] and v[j] satisfy the following condition at state s:

$$(v[i] = v[j] - 1 \ \mathbf{mod}\ 2d)$$

As shown in the proof of Theorem 1, G.s has no directed cycles. Thus the directed edges in G.s form a set, of possibly overlapping, routes. Each route is a simple directed path in G.s where the tail node has no incoming directed edges and the head node has no outgoing directed edges. Thus, each route consists of a tail node, followed by a sequence of zero or more middle nodes, followed by a head node.

Consider any route R in G.s, and let us examine what happens to route R during a maximally concurrent transition (s, s′). The tail node and each middle node of R are not enabled at state s, and cannot be executed in the transition (s, s′), and so they remain in R. The head node of R is enabled at state s, and its execution in the transition (s, s′) causes its incoming directed edges at state s to become undirected at state s′. In other words, the head node of R at state s is no longer part of R at state s′. However, it is possible that a new node is added to R as tail at state s′. This node, if it exists, is enabled at state s, and its execution in the transition (s, s′) establishes a directed edge from this node to the old tail of R. In summary, route R loses its head node in each maximally concurrent transition, but may gain a new tail node in the same transition.

Next, we show that route R cannot gain a new tail node in every maximally concurrent transition. The proof is by contradiction. Assume that route R will gain a new tail node in every maximally concurrent transition. Because every G.s has the same finite number of nodes, R will gain at least one node i, as its tail node, more than once. In order for route R to gain node I twice, the tail of R needs first to reach node i traverse a simple cycle (in the topology of the alternator) that contains node i, then returns to node i. Let the nodes of this cycle be (i; x; y; . . . ; z; i). Assume that the number of edges in this cycle is m, where m ≤ d and d is the cyclic distance of the alternator. Now consider the following facts.

node i becomes the tail of R first time	at state s.0,
node x becomes the tail of R	at state s.1,
node y becomes the tail of R	at state s.2, ...
node z becomes the tail of R	at state s.(m-1), and
node i becomes the tail of R second time	at state s.m,

where (s.0, s.1), (s.1, s.2), …, (s.(m − 1), s.m) are the sequence of maximally concurrent transitions executed by the alternator in going from state s.0 to state s.m. Thus, the values of variables v[i], v[x], v[y], . . ., and v[z] at the states s.0 through s.m satisfy the following conditions.

At s.0,	v[i] = u, for some value u in the range 0..2d–1.
At s.1,	v[i] = u, and v[x] = u–1 **mod** 2d.
At s.2,	v[i] = u, or u+1 **mod** 2d, and v[y] = u–2 **mod** 2d.
...	
At s.(m–1),	v[i] = u, or u+1 **mod** 2d, ... , or u+m–2 **mod** 2d, and v[z] = u–m+1 **mod** 2d.
At s.m,	v[i] = u, or u+1 **mod** 2d, ... , or u+m–1 **mod** 2d, and v[z] = u–m+1 **mod** 2d.

Therefore, because m ≤ d, v[i] ≠ v[z] − 1 **mod** 2d at state s.m, node i cannot be the tail node of route R at s.m. Contradiction. Thus, route R cannot gain a new tail node in

every maximally concurrent transition of the alternator, and so R eventually becomes shorter.

Note that in some (s, s′) transitions new routes may be added to G.s causing the number of routes in G.s′ to become larger than the number of routs in G.s. However, for each newly added route R′ in G.s', there is a route R in G.s such that R′ is shorter than R.

It follows from this discussion that every route in G.s (including the newly added routes) eventually becomes shorter. This continues until G.s has no directed edges. When this happens, state s is loose. Moreover, every state that comes after this loose s in the maximally concurrent computation is also loose. □

4 Use of the alternator

In this section, we describe how to use alternators in transforming any system that is stabilizing under the assumption that actions are executed serially (one at a time) into one that is stabilizing under the assumption that actions are executed concurrently (any number of them at a time).

Consider a process array q[i : 0 . . . n − 1] where n ≥ 2. The topology of array q[i : 0 . . . n − 1] is a connected undirected graph where each node represents one process in the array. Two processes in the process array q[i : 0 . . . n − 1] are called neighbors iff there is an (undirected) edge between the two nodes representing the processes.

In this process array, each process q[i] has k variables and m actions.

Each action in a process q[i] is of the form

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

where <guard> is a boolean expression over the variables of q[i] and the variables of the neighbors of q[i], and <statement> is a sequence of assignment statements that update the variables of q[i]. The processes in the process array q[i : 0 . . . n − 1] can be defined as follows.

```
process q[i : 0..n–1]
var     <variable 0> :          <type 0>,
        ...
        <variable k–1>:         <type k–1>
begin
        <guard 0>       -->     <statement 0>
[]      ...
[]      <guard m–1>  -->        <statement m–1>
end
```

A state of the process array q[i : 0 . . . n − 1] is an assignment of a value to every variable in every process in the array. The value assigned to each variable is from the domain of values of that variable. An action in the process array q[i : 0 . . . n − 1] is enabled at a state s iff the guard of that action is true at state s.

We assume that the process array q[i : 0 . . . n − 1] satisfies the following two conditions.

(i) Determinacy: At each state s, each process q[i] has at most one action that is enabled at s.
(ii) Enabling: At each state s, each process q[i] has at least one action that is enabled at s.

(The assumption that each process q[i] satisfies these two conditions is not a severe restriction. For example, if a process q[i] has two actions whose guards are "<guard 1>" and "<guard 2>" and if these two actions are enabled at some state, then replace "<guard 2>" by "**not** < guard 1 > ∧ < guard 2 >". The resulting two actions will never be enabled at the same state. Also, if none of the actions of a process q[i] is enabled at some state, then add to q[i] the action

not < guard 1 > ∧ · · · ∧ **not** < guard m > → **skip**

where < guard 1 >, . . . , < guard m > are the guards of all actions in q[i]. This added action is enabled at any state where none of the other actions is enabled.)

A serial transition of the process array q[i : 0 . . . n − 1] is a pair (s, s′) of states such that starting the process array at state s then executing the statement of one action that is enabled at s yields the process array in state s′.

A serial computation of the process array q[i : 0 . . . n − 1] is an infinite sequence s.0, s.1, . . . of states such that each pair (s.i, s.(i + 1)) of consecutive states in the sequence is a serial transition. A serial computation of the process array q[i : 0 . . . n − 1] is called weakly fair iff along the computation, if any action becomes continuously enabled, then this action is eventually executed.

A set S of the states of the process array q[i : 0 . . . n − 1] is called closed iff for every serial transition (s, s′) of the process array, if s is in S, then s′ is in S.

Let S be a closed set of the states of the process array q[i : 0 . . . n − 1]. The process array q[i : 0 . . . n − 1] is said to be serially stabilizing to S iff every serial computation of the process array has an infinite suffix where each state is in set S. The process array q[i : 0 . . . n − 1] is said to be serially stabilizing to S under weak fairness iff every weakly fair computation of the process array has an infinite suffix where each state is in set S.

In the remainder of this section, we show that if the process array q[i : 0 . . . n − 1] is serially stabilizing to S, then another process array q′[i : 0 . . . n − 1] is concurrently stabilizing to S′, where q′[i : 0 . . . n − 1] and S′ are strongly related to q[i : 0 . . . n − 1] and S, respectively. We start by showing how to construct the process array q′[i : 0 . . . n − 1] from the process

array q[i : 0 . . . n − 1] and an alternator p[i : 0 . . . n − 1] whose topology is isomorphic to that of q[i : 0 . . . n − 1].

Each process q′[i] can be constructed from process q[i] and process p[i] in the alternator as follows.

```
process q'[i : 0..n–1]                                  {n ≥ 2}
const   d :                     integer                 {cyclic distance}
var     <variable 0> :          <type 0>,
        ...
        <variable k–1> :        <type k–1>,
        v[i] :                  0..2d–1
begin
        G.i ∧ v[i] ≠ 2d–1                               --> S.i
[]      G.i ∧ v[i] = 2d–1 ∧ <guard 0>                   --> S.i ; <statement 0>
[]      …
[]      G.i ∧ v[i] = 2d–1 ∧ <guard m-1>                 --> S.i ; <statement m–1>
end
```

where

d is the cyclic distance of the alternator p[i : 0 . . . n − 1],
v[i] is the variable in process p[i] in the alternator,
G.i is the guard in process p[i] in the alternator, and
S.i is the statement in process p[i] in the alternator.

A state of the process array q′[i : 0 . . . n − 1] is an assignment of a value to very variable in every process in the array. The value assigned to each variable is from the domain of values of that variable. An action in the process array q′[i : 0 . . . n − 1] is enabled at a state s iff the guard of that action is true at state s. The next two lemmas follow from Theorem 1 and the assumption that q[i : 0 . . . n − 1] satisfies the enabling and determinacy conditions.

Lemma 1 *For each state s, at least one action in a process in the process array q′[i : 0 . . . n − 1] is enabled at s.* □

Lemma 2 *For each state s, at most one action in each process in the process array q′[i : 0 . . . n − 1] is enabled at s.* □

The definitions of a concurrent or maximally concurrent transition, and of a concurrent and maximally concurrent computation, that we gave in Section 2 for the alternator p[i : 0 . . . n − 1], can also be given for the process array q′[i : 0 . . . n − 1]. A state s of the process array q′[i :0 . . . n − 1] is called loose iff exactly one action in every process in q′[i :0 . . . n − 1] is enabled at s.

It is straightforward to show that the process array q′[i : 0 . . . n − 1] satisfies the same three properties, namely safety, progress, and stabilization, of the alternator. (Actually, the fact that q′[i : 0 . . . n − 1] satisfies each of these properties follows from the fact that the alternator satisfies the same property.)

Let S be a closed set of states of the process array q[i : 0 . . . n − 1]. The extension of S to the process array q′[i : 0 . . . n − 1] is the set S′ of all states of the process array q′[i : 0 . . . n − 1] such that for every state s′ in set S′, there is a state s in set S where every variable in q[i : 0 . . . n − 1] has the same value in s and s′.

Let S′ be the extension of S to the process array q′[i : 0 . . . n − 1]. The process array q′[i : 0 . . . n − 1] is said to be concurrently stabilizing to S′ iff every concurrent computation of the process array q′[i : 0 . . . n − 1] has an infinite suffix where each state is in S′. The following theorem is straightforward.

Theorem 5 *If the process array q[i : 0 . . . n − 1] is serially stabilizing to S under weak fairness, then the process array q′[i : 0 . . . n − 1] is concurrently stabilizing to the extension of S to q′[i : 0 . . . n − 1].* □

5 Example on use of the alternator

Consider a process array q[i : 0 . . . n − 1] where each process q[i] has a Boolean variable b[i]. The values of these b[i] variables are to be assigned such that the following two conditions hold.

(i) The number of processes, whose b[i] variables are true, is maximal.
(ii) The b[i] variables of every two neighboring processes are not both true.

A process q[i] in this array can be defined as follows.

```
process q[i : 0..n–1]
var b[i] :      boolean
begin
        (For every j, q[j] is a neighbor of q[i], ~b[j]) -->    b[i] := true
[]      (There exists j, q[j] is neighbor of q[i], b[j]) -->    b[i] := false
end
```

Note that this process array satisfies the two conditions of "determinacy" and "enabling" specified in Sect. 4.

Let S be the set of every state of this process array at which the following predicate holds:

(For every i,
(For every j, q[j] is a neighbor of q[i], ∼ b[j]) = b[i])

It is straightforward to show that this process array is serially stabilizing to S under weak fairness, but it is not concurrently stabilizing to S (even under weak fairness).

In order to design a process array that is concurrently stabilizing to S, we can combine the process array q[i : 0 . . . n − 1] with an alternator p[i : 0 . . . n − 1] of the same topology as

discussed in Sect. 4. The resulting process array q′[i : 0 . . . n − 1] is defined as follows.

```
process q'[i : 0..n–1]
const d      :   integer      {cyclic distance}
var b[i]     :   boolean,
    v[i]     :   0..2d–1
begin
      G.i ^ v[i] ≠ 2d–1                          --> v[i] := v[i] + 1
[]    G.i ^ v[i] = 2d–1 ^
      (For every j, q[j] is a neighbor of q[i], ~b[j]) --> v[i] := 0; b[i] := true
[]    G.i ^ v[i] = 2d–1 ^
      (There exists j, q[j] is neighbor of q[i], b[j]) --> v[i] := 0; b[i] := false
end
```

Recall that G.i is the guard of the single action of process p[i] in the alternator.

From Theorem 5, the process array q′[i : 0 . . . n − 1] is concurrently stabilizing to S′, where S′ is the set of every state of q′[i : 0 . . . n − 1] at which the following predicate holds.

(For every i,
(For every j, q′[j] is a neighbor of q′[i], ∼ b[j]) = b[i])

6 Low atomicity alternator

The alternator (as specified in Sect. 2) has a high atomicity. During the execution of a single action, a process p[i] checks that its priority is not less than the priority of every neighbor before it increments its variable v[i]. (Formally, the priority of a process p[i] is less than the priority of a neighbor p[j] at some state iff either (v[i] = v[j] ∧ i > j) or (v[i] = v[j] − 1 **mod** 2d) at that state.)

The atomicity of the alternator can be reduced because of the following two observations concerning any two neighboring processes p[i] and p[j] in the alternator.

(i) Ordered execution of neighbors: If the priority of p[i] is less than the priority of p[j], then p[i] should wait until p[j] increments v[j] and the priority of p[i] is no longer less than that of p[j] before p[i] increments v[i].
(ii) Concurrent execution of neighbors: If neither the priority of p[i] is less than that of p[j] nor the priority of p[j] is less than that of p[i], then p[i] and p[j] can increment their variables v[i] and v[j] concurrently (in parallel or in any order).

Based on these two observations, the low atomicity alternator can be defined as follows.

```
process p[i : 0..n–1]                                   {n≥ 2}
const d      : integer                                  {cyclic distance}
      h      : integer                                  {number of neighbors}
      ngh    : array [0..h–1] of 0..n–1                 {indices of neighbors}
var   v[i]   : 0..2d–1,
      x      : 0..h–1
begin
      (v[i] ≠ v[ngh[x]]  ∨  i < ngh[x])      ^
      (v[i] ≠ v[ngh[x]] – 1 mod 2d)          ^
      (x < h–1)                  -->         x := x + 1
[]    (v[i] ≠ v[ngh[x]]  ∨  i < ngh[x])      ^
      (v[i] ≠ v[ngh[x]] – 1 mod 2d)          ^
      (x = h–1)                  -->         x := 0; v[i] := v[i] +1 mod 2d
end
```

This alternator does not satisfy the three properties (of safety, progress, and stabilization) of the high atomicity alternator. Instead, it satisfies the following two properties:

(i) Progress of low atomicity alternator: Along each concurrent computation of the low atomicity alternator, the actions of each process are executed infinitely often.
(ii) Stabilization of low atomicity alternator: Each concurrent computation of the low atomicity alternator has an infinite suffix where the following predicate holds at each state:
(For every neighboring processes p[i] and p[j],
v[i] ≠ v[j])

It is instructive to compare this low atomicity alternator with the low atomicity refinement proposed by Nesterenko and Arora in [8]. On one hand, the alternator is simpler with two variables and two actions per process, whereas the refinement requires seven variables and four actions per process. On the other hand, the refinement requires less atomicity and is more efficient than the alternator as explained next. First, each action in the alternator reads the v variables of a neighboring process and may write the v variables of its own process, whereas each action in the refinement in [8] either reads the variables of a neighboring process or writes the variables of its own process. Second using the refinement in [8], an application process executes one application action every O(h) actions of the refinement actions, whereas using the alternator, an application process executes one application action every O(h * d) alternator actions, where h and d are the degree and cyclic distance of the application, respectively.

7 Concluding remarks

An alternator is a system that can be used in transforming any system that is stabilizing under the assumption that actions

are executed serially into one that is stabilizing under the assumption that actions are executed concurrently. In this paper, we presented a class of alternators with arbitrary topology, and discussed how to use these alternators in transforming any system that is stabilizing assuming serial execution into one that is stabilizing assuming concurrent execution.

The guard of each action in the alternator has the predicate $i < j$, where the values of i and j are in the range $0 \ldots n-1$ and n is the number of processes in the alternator. This predicate can be replaced by the predicate pr[i] < pr[j], where each pr[i] is a fixed integer, called priority, assigned to process p[i]. The priorities are assigned to processes such that for any two neighboring processes p[i] and p[j], $pr[i] \neq pr[j]$. Thus, each assigned priority is in the range $0 \ldots g$, where g is the degree of the alternator (i.e. the maximum number of neighbors for a process in the alternator).

Thus, the local state of each process in the alternator can be implemented using log(d) + log(g + 1) bits, where d is the cyclic distance of the alternator and g is the degree of the alternator. Therefore, alternators with relatively small values of d and g can be implemented cheaply in hardware.

The system transformation in this paper uses bounded memory but its stabilization property is not silent (as characterized in [3]), whereas the system transformation in [7] uses unbounded memory but its stabilization property is silent. The problem of whether there exists a system transformation that uses bounded memory and whose stabilization property is silent is closed recently in the affirmative [8].

The alternator can be viewed as a realization of the dining philosophers. In this view, each process in the alternator is a philosopher, and each philosopher p[i] can eat during any execution of its action that starts when $v[i] = 2d - 1$. Because the dining philosophers can be used in solving many synchronization problems, we believe that the alternator can be used in realizing many of these solutions.

Efficient alternators with special topologies (such as rings and hypercubes) are discussed in [5].

Acknowledgments We are thankful to Anish Arora, Masaaki Mizuno and Mikhail Nesterenko for useful discussions during and after the writing of this paper. We are also thankful to the referees for several suggestions concerning the presentation in this paper.

References

1. Arora, A., Attie, P., Evangelist, M., Gouda, M.G.: Convergence of iteration systems,. Distrib. Comput. **7**, 43–53 (1993)
2. Burns, J.E., Gouda, M.G., Miller, R.E.: On relaxing interleaving assumptions. In: Proceedings of the MCC Workshop on Self-Stabilizing Systems, Austin, Texas (1989)
3. Dolev, S., Gouda, M.G., Schneider, M.: Memory requirements for silent stabilization. In: Proceedings of the 1996 ACM Symposium on Distributed Computing Systems (PODC-96), pp. 27–34 (1996)
4. Gouda, M.G., Haddix, F.: The linear alternator. In: Proceedings of the Third Workshop on Self-Stabilizing Systems (WSS-97), International Informatics Series 7, Carleton University Press, pp. 31–47 (1997)
5. Haddix, F.: Alternating parallelism and the stabilization of cellular systems. Ph.D. Dissertation, Department of Computer Sciences, the University of Texas at Austin, Austin, Texas (1999)
6. Mizuno, M., Kakugawa, H.: A timestamp based transformation of self-stabilizing programs for distributed computing environments. In: Proceedings of the International Workshop On Distributed Algorithms (WDAG). Also published in Lecture Notes on Computer Science, vol. 1151, pp. 304–321 (1996)
7. Mizuno, M., Nesterenko, M.: A transformation of self- stabilizing serial model programs for asynchronous parallel computing environments. Inf. Process. Lett. **66**(6), 285–290 (1998)
8. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. In: Proceedings of DESC (1999)