The Linear Alternator

Mohamed G. Gouda and Furman Haddix

Department of Computer Sciences
The University of Texas at Austin, Austin, TX 78712

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 a linear alternator and verify its correctness. We also show that this alternator can be used in transforming any linear system that is stabilizing assuming a serial execution, to one that is stabilizing assuming a concurrent execution.

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.

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 that 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.

$b \neq c$ --> $b := c$

$b \neq c$ --> $c := b$

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 together, then this system will stay in states where $b \neq c$ indefinitely.

In order 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 the system to one that is stabilizing under the assumption of concurrent execution. For example, consider the above system that is stabilizing under serial execution, but not under concurrent execution, to states where b = c. This system can be transformed by

adding a third boolean variable x and modifying the two actions of the system to become as follows.

$b \neq c \wedge x$ --> $b := c$

$b \neq c \wedge \textbf{not}\, x$ --> $c := b$

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

Examples of system transformations for achieving stabilization under the assumption of concurrent execution are reported in [4] and in the current paper. The transformation in [4] 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 the current paper ensures that no conflicting actions (namely no actions that read or write common variables) are enabled for execution simultaneously. This transformation is based on a class of systems called linear alternators. We specify linear alternators in Section 2, and prove their stabilization properties in Section 3. Then in Section 4, we show how to use linear alternators to transform any linear system that is stabilizing under the assumption of serial execution to one that is stabilizing under concurrent execution.

2 Specification of the Linear Alternator

A linear alternator is an array of n processes p[i : 0..n-1], where $n \geq 2$. Process p[0] has a right neighbor p[1], process p[n-1] has a left neighbor p[n-2], and each other process p[i] has both a left neighbor p[i-1] and a right neighbor p[i+1]. Each process p[i] in the alternator has one boolean variable b[i] and one action. The action of process p[i] is of the form

<guard> --> <statement>

where <guard> is a boolean expression over the variable of p[i] and those of the neighbors of p[i], and <statement> is an assignment statement of the form b[i] := **not** b[i].

The processes of the linear alternator are defined as follows.

```
process p[0]
var     b[0]              :     boolean
begin
        b[0] = b[1]       -->   b[0] := not b[0]
end

process p[i : 1..n-2]
var     b[i]              :     boolean
begin
        b[i-1] ≠ b[i] = b[i+1]    -->   b[i] := not b[i]
end

process p[n-1]
var     b[n-1]            :     boolean
begin
        b[n-2] ≠ b[n-1]   -->   b[n-1] := not b[n-1]
end
```

A state of the alternator is a mapping of a value, false or true, to every variable b[i] in 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.

Lemma 1:

For each state s, the action of at least one process in the alternator is enabled at state s. []

A concurrent transition (or maximal transition, respectively) of the 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 maximal computation, respectively) of the 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 maximal transition, respectively).

(The fact that each concurrent computation and each maximal computation is infinite follows from Lemma 1 above.)

An alternating state of the alternator is a state s where either the actions of all even processes (namely processes p[0], p[2], ...) are enabled at s, or the actions of all odd processes (namely processes p[1], p[3], ...) are enabled at s.

Each alternating state of the alternator can be represented by a regular expression over the alphabet {F, T}, where F stands for false, and T stands for true. For example, an alternating state sa where

b[0] = false, b[1] = false, b[2] = true, b[3] = true,
b[4] = false, b[5] = false, b[6] = true, b[7] = true,
...

can be represented using the regular expression

sa = $(FFTT)^*$ (FF + FFT + FFTT + FFTTF)

Note that the shortest string defined by this regular expression, namely FF, consists of two symbols. This is because the alternator consists of at least two processes.

The alternator has only two alternating states where the actions of all even processes are enabled. These two states, sa and sc, can be represented by the following regular expressions.

sa = $(FFTT)^*(FF + FFT + FFTT + FFTTF)$

sc = $(TTFF)^*(TT + TTF + TTFF + TTFFT)$

Also, the alternator has only two alternating states where the actions of all odd processes are enabled. These two states, sb and sd, can be represented by the following regular expressions.

sb = $(TFFT)^*(TF + TFF + TFFT + TFFTT)$

sd = $(FTTF)^*(FT + FTT + FTTF + FTTFF)$

From this discussion, the alternator has only four alternating states: sa, sb, sc, and sd. These four states are involved in the following four maximal transitions of the alternator:

(sa, sb), (sb, sc), (sc, sd), and (sd, sa).

Therefore, the infinite sequence of states sa, sb, sc, sd, sa, ... constitutes a maximal computation of the alternator.

It is straightforward to show that the alternator satisfies the following three interesting properties.

i. Non-interference:
At each state of the alternator, if a process has an enabled action, then no neighbor of that process has an enabled action.

ii. Progress:
Along each concurrent computation of the alternator, the action of each process is executed infinitely often.

iii. Stabilization:
Each maximal computation of the alternator has an infinite suffix where each state is alternating.

3 Stabilization of the Linear Alternator

To prove that the alternator is stabilizing to an alternating state along any maximal computation, we need to state the following three definitions.

A process p[i] in the alternator is X-marked at a state s iff $0 < i < n-1$ and $b[i-1] = b[i] = b[i+1]$ at state s.

A process p[i] in the alternator is Y-marked at a state s iff $0 < i < n-1$ and $b[i-1] \neq b[i] \neq b[i+1]$ at state s.

A process p[i] in the alternator is Z-marked at a state s iff p[i] is neither X-marked nor Y-marked at state s. In other words, p[i] is Z-marked at s iff one of the following three conditions holds: $i = 0$, $i = n-1$, or $0 < i < n-1$ and $b[i-1] \neq b[i+1]$ at s.

Lemma 2:
At each state, each process in the alternator is X-marked, Y-marked, or Z-marked. []

Lemma 3:

If at a state s every process in the alternator is Z-marked, then s is an alternating state. []

Stabilization of the alternator can be established by proving the following two propositions for any maximal computation s.0, s.1, ... of the alternator.

1. For every s.i in the computation, the number of X-marked processes at s.i is at least the number of X-marked processes at s.(i+1), and the number of Y-marked processes at s.i is at least the number of Y-marked processes at s.(i+1).

2. For every s.i in the computation, there is a later s.j such that the number of X-marked processes at s.i is either zero or less than the number of X-marked processes at s.j, and the number of Y-marked processes at s.i is either zero or less than the number of Y-marked processes at s.j.

From these two propositions, the maximal computation s.0, s.1, ... has a state s.k where the number of X-marked processes is zero, and the number of Y-marked processes is zero. Thus, every process in the alternator is Z-marked at state s.k (from Lemma 2), and s.k is an alternating state (from Lemma 3). This shows that the alternator is stabilizing to an alternating state along any maximal computation.

It remains now to prove the correctness of propositions 1 and 2. We carry out these proofs for X-marked processes only. (The proofs for Y-marked processes are similar by symmetry.)

Let s be an arbitrary state of the alternator, and let p[i] be an X-marked process at state s. Each of the two neighbors of p[i] is either X-marked or Z-marked at s. Thus at state s, the X-marked processes can be grouped into intervals of consecutive X-marked processes, and each of the intervals ends at the left by a Z-marked process, and ends at the right by a Z-marked process. Consider any such interval at state s. This interval can be represented as follows.

ZXX ... XZ

We need to prove that the number of X-marked processes in this interval cannot increase, and that it will eventually decrease along any maximal computation. The proof is by considering the following three cases concerning the structure of the interval.

i. The right Z-marked process in the interval is p[n-1]: In this case, the alternator makes a maximal transition from state s to state s' such that the interval

ZXX ... XZ at s

is replaced by the interval

ZZX ... XZ at s'.

Therefore, the number of X-marked processes at s is more than that at s'.

ii. The right Z-marked process in the interval is followed by a Y-marked process: In this case, the

alternator makes a maximal transition from state s to state s' such that the interval

ZXX ... XZY at s

is replaced by the interval

ZZX ... XZY at s'.

Therefore, the number of X-marked processes at s is more than that at s'.

iii. The right Z-marked process in the interval is followed by another Z-marked process: In this case, the alternator makes a maximal transition from state s to state s' such that the interval

ZXX ... XZZ at s

is replaced by the interval

ZZX ... XXZ at s'.

Therefore, the number of X-marked processes at s is the same as that at s', but the interval of X-marked processes has shifted one position to the right.

The shift to the right by the interval in case iii can continue until the right Z-marked process is p[n-1] (and case i applies), or until the right Z-marked process is followed by a Y-marked process (and case ii applies). In either case, the number of X-marked processes in the interval starts to decrease until it becomes zero.

In summary, the number of X-marked processes in a state cannot increase and will eventually decrease to zero along any maximal computation. Using a similar argument, we can show that the number of Y-marked processes in a state cannot increase and will eventually decrease to zero along any maximal computation. Therefore, starting from any state,

the alternator will eventually reach, in any maximal computation, a state where each process is Z-marked. This state is alternating. This completes the proof that the alternator is stabilizing to an alternating state along any maximal computation.

4 Use of the Linear Alternator

In this section, we describe how to use the linear alternator in transforming any linear 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 linear process array q[i : 0..n-1], where n ≥ 2. Process q[0] has a right neighbor q[1], process q[n-1] has a left neighbor q[n-2], and each other process q[i] has both a left neighbor q[i-1] and a right neighbor q[i+1]. Each process q[i] has k variables and m actions. Each action in a process q[i] is of the form

<guard> --> <statement>

where <guard> is a boolean expression over the variables of q[i] and those 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 set S of the states of the process array q[i : 0..n-1] is 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 serially stabilizing to S iff every serial computation of the process array has an infinite suffix where each state is in S.

In 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 the linear alternator p[i : 0..n-1] in Section 2.

Each process q'[i] can be constructed from process q[i] and process p[i] in the linear alternator, as follows. First, a copy of the boolean variable

b[i] in process p[i] is added to process q[i]. Second, guard G.i of the action of process p[i] is added as a conjunct to the guard of every action in process q[i]. Third, statement S.i of the action of process p[i] is added to the statement of every action in process q[i]. The resulting process array q'[i : 0..n-1] is defined as follows.

```
process q'[i : 0..n-1]
var     <variable 0>          :     <type 0>,
        ...
        <variable k-1>        :     <type k-1>,
        b[i]                  :     boolean
begin
        G.i ∧ <guard 0>       -->   S.i ; <statement 0>
[]      ...
[]      G.i ∧ <guard m-1>     -->   S.i ; <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.

Lemma 4:

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 5:

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 maximal transition, and of a concurrent and maximal computation, that were given in Section 2 for the linear alternator p[i : 0..n-1], can also be given for the process array q'[i : 0..n-1].

An alternating state of the process array q'[i :0..n-1] is a state s where either exactly one action in every even process (namely processes q'[0], q'[2], ...) is enabled at s, or exactly one action in every odd process (namely processes q'[1], q'[3], ...) is enabled at s.

It is straightforward to show that the process array q'[i : 0..n-1] satisfies the following three properties.

i. Non-interference:
At each state of the process array q'[i : 0..n-1], if a process has an enabled action, then no neighbor of that process has an enabled action.

ii. Progress:
Along each concurrent computation of the process array q'[i : 0..n-1], an action of each process is executed infinitely often.

iii. Stabilization:
Each maximal computation of the process array q'[i : 0..n-1] has an infinite suffix where each state is alternating.

Note that these properties of the process array q'[i : 0..n-1] are similar to the properties of the linear alternator discussed in Section 2. Actually, the fact that q'[i : 0..n-1] satisfies these three properties follows from the fact that p[i : 0..n-1] satisfies similar properties.

Let S be a closed set of states of the process array q[i : 0..n-1]. The extension of S 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 a closed set of states of the process array q[i : 0..n-1], and let S' be the extension of S. The process array q'[i : 0..n-1] is concurrently stabilizing to S' iff every concurrent computation of the process array has an infinite suffix where each state is in S'.

Theorem 1:
If the process array q[i : 0..n-1] is serially stabilizing to S, then the process array q'[i : 0..n-1] is concurrently stabilizing to S', where S' is the extension of S. []

5 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 linear alternator, and discussed how to use this alternator in transforming any linear system that is stabilizing assuming serial execution into one that is stabilizing assuming concurrent execution.

Currently, we are developing alternators with more general topologies (rather than mere linear topology). These results will be reported in [3].

References

[1] Anish Arora, Paul Attie, Mike Evangelist, and Mohamed G. Gouda, "Convergence of Iteration Systems", Distributed Computing, Volume 7, pp. 43 - 53, 1993.

[2] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller, "On Relaxing Interleaving Assumptions", Proceedings of the MCC Workshop on Self-Stabilization, Austin, Texas, 1989.

[3] Furman Haddix, "Alternating Parallelism and the Stabilization of Cellular Systems", Ph. D. Dissertation in progress, Department of Computer Sciences, the University of Texas at Austin, Austin, Texas, 1997.

[4] Masaaki Mizuno and Hirotsugu Kakugawa, "A Timestamp Based Transformation of Self-Stabilizing Programs for Distributed Computing Environments" Proceedings of the International Workshop On Distributed Algorithms (WDAG). Also published in Lecture Notes on Computer Science, Volume 1151, pp. 304 - 321.