Alternators in Read/Write Atomicity

Sandeep S. Kulkarni Chase Bolen
John Oleszkiewicz Andrew Robinson
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

The alternator problem requires that in legitimate states no two neighboring processes are enabled and between two executions of a process, its neighbors execute at least once. In this paper, we present a solution for the alternator problem that has the following properties: (1) If the underlying topology is *arbitrary* and the program is executed in read/write atomicity then it is stabilizing fault-tolerant, i.e., starting from an arbitrary state, it recovers to states from where its specification is satisfied, (2) If the underlying topology is *bipartite* and the program is executed in the concurrent execution model then it provides stabilizing fault-tolerance and maximal concurrency, (3) If the underlying topology is *linear* or *tree* then the program provides both these properties, and (4) The program uses bounded state if the network size is known. To our knowledge, this is the first alternator program that achieves these properties.

Keywords : Stabilization, Alternator, Program transformation
Serial Execution Model (Interleaving Semantics)
Concurrent Execution Model (Powerset Semantics), Read/Write atomicity

1 Introduction

To simplify the presentation of stabilizing programs [5] and to facilitate their verification, these programs are often expressed in shared memory model and they are assumed to execute in interleaving semantics (*serial execution model*). Specifically, in these (high atomicity) programs, we assume that in an atomic step, a process can read the state of its neighbors and update its state. Moreover, it is assumed that the enabled actions of a program are executed one at a time.

To implement these stabilizing programs, however, it is desirable to use a less restrictive model. One such model is *concurrent execution model* where program actions are expressed in shared memory model but are assumed to execute in *powerset semantics* where any nonempty subset of enabled actions are executed at a time. Another such model is read/write atomicity where each process can atomically read the state of its neighbor or write its own state but not both. In read/write atomicity, it is not possible for a process to read the state of its neighbor (respectively all its neighbors) and write its own state in one atomic step. Hence, a process may be using old information when it updates its own state.

In [6–8], the problem of alternator was introduced as a way to transform a stabilizing program that is correct under the serial execution model into a stabilizing program that is correct under the concurrent execution model. To achieve this transformation, the solutions in [6–8] ensure that no two neighboring processes

[1]Email: {sandeep, bolencha, oleszkie, robin507}@cse.msu.edu
Web: `http://www.cse.msu.edu/~sandeep`
Tel: +1-517-355-2387
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA contract F33615-01-C-1901, ONR Grant N00014-01-1-0744, and a grant from Michigan State University.

are enabled simultaneously. This ensures that if a subset of the enabled processes execute concurrently then their concurrent execution is equivalent to their serial execution. Additionally, they also show that each process can execute infinitely often and that the alternator itself is stabilizing in concurrent execution.

One of the important properties of the alternator solutions in [6, 8] is *maximal concurrency* if the underlying topology is a line or a tree. Specifically, in these solutions, in legitimate states, the set of enabled processes is maximal, i.e., for any process j, either j or a neighbor of j is enabled. Moreover, if all these enabled processes execute at once then the resulting state is also one where the set of enabled processes is maximal. Unfortunately, the programs in [6–8] are not stabilizing in read/write atomicity. In other words, if the alternator from [6–8] is executed in read/write atomicity then it may remain in illegitimate states forever.

In this paper, we focus on the problem of refining the alternator solutions in [6,8] so that (1) they continue to provide maximum concurrency if executed in concurrent execution model and (2) their implementation in read/write atomicity is stabilizing fault-tolerant. Such a solution is especially useful in a system where processes typically execute in the concurrent execution model, although occasionally, some read/writes may be staggered. When read/write actions are staggered thus, the staggered execution may not be a computation of that program in the concurrent execution model. However, the staggered execution will be a computation of the given program in read/write atomicity. Since computation of a program in the concurrent execution model is also its computation in read/write atomicity, in the above scenario, the overall execution of the program will be in read/write atomicity. Moreover, parts of the computation will be computations in the concurrent execution model. Thus, in such systems, our solution will ensure correctness even if the read/writes are staggered and the computation is in read/write atomicity. Moreover, it will provide maximal concurrency for the part where program is executed in the concurrent execution model.

Contributions of the paper. In this paper, we present an alternator program that has the following properties:

1. If the program is executed in read/write atomicity, it is stabilizing fault-tolerant [5] even if the underlying topology is *arbitrary*. Thus, even if the alternator program is perturbed to an arbitrary state, eventually it would recover to states from where it satisfies its specification.

2. If the underlying topology is *bipartite* and the program is executed in the concurrent execution model then the program is stabilizing fault-tolerant and provides maximal concurrency in the legitimate states.

3. If the underlying topology is a tree (respectively, line), the program provides *both* properties, i.e., if executed in the concurrent execution model, it is stabilizing fault-tolerant and provides maximum concurrency in legitimate states. And, if executed in read/write atomicity, it is stabilizing fault-tolerant.

4. It uses bounded space if the network size (or an upper bound on it) is known and it does not violate fairness when the bounded counters are reset to 0.

We note that while the previous work (e.g., [1–3, 9–11]) has focused on transforming a program from serial execution model to read/write atomicity, it has not addressed the issue of maximal concurrency in the concurrent execution model. Specifically, the transformation in [9] ensures that each concurrent execution of actions is serializable. The transformation in [10] uses unbounded timestamps to order the execution of processes. The transformations in [1–3] uses bounded timestamps to order execution of actions. However, in [1], when the timestamps are reset to 0, the fairness of action executions is violated. Finally, the program in [11] is based on the solution for dining philosopher's problem and uses bounded variables. Our program uses bounded timestamps and ensures that the fairness is preserved when the timestamps are reset to 0.

Organization of the paper. The rest of the paper is organized as follows: In Section 2, we present the problem statement for the alternator; this problem statement is a slight modification of that in [6–8]. Since

our solution is based on the solution for asynchronous unison from [4], we discuss that solution in Section 3. In Section 4, we present the alternator program that is stabilizing in read/write atomicity. In Section 5, we present an alternator program that ensures that (1) its execution provides maximal concurrency in the concurrent execution model and (2) the implementation of that program in read/write atomicity is stabilizing fault-tolerant. Finally, we make concluding remarks in Section 6.

2 Model and Problem Statement

In this section, we present the program model and define problem statement for the alternator. The program consists of a set of processes. Each process consists of two types of actions: the actions of the underlying program (that are stabilizing under the serial execution model) and the actions of the alternator. Each action of a process is of the form:

$$\langle guard \rangle \longrightarrow \langle statement \rangle,$$

where $\langle guard \rangle$ is a boolean expression involving the variables of the process and the variables of its neighbors and $\langle statement \rangle$ updates the variables of the process.

The goal of our alternator program is to identify how the actions of the underlying program in serial execution model, called high atomicity actions, can be executed in read/write atomicity while ensuring that the stabilizing fault-tolerance is preserved. It follows that the alternator itself must be stabilizing fault-tolerant while executing in read/write atomicity. Therefore, we classify the variables of the alternator as public variables and private variables. We also classify the actions of the alternator as 'read actions' or 'write actions'. In the read action, the alternator is allowed to read the public variables of one of its neighbors and write its private variables (e.g. to make a copy of the variable it read). In the write action, the alternator is allowed to read and write its own public and private variables, but not those of its neighbors.

Since the alternator is intended for transforming a program that is correct under the serial execution model into the read/write atomicity, the alternator needs to identify a *privileged point* where the underlying program can execute one of its high atomicity actions by reading the state of its neighbors and writing its own state. To ensure that the execution of the two high atomicity actions does not overlap, it suffices to ensure that the alternators at neighboring processes are not at the privileged point simultaneously. Also, to ensure that each process can execute its high atomicity actions, each process should reach its privileged point infinitely often. In fact, based on the specification in [6], we require that between any two points where a process reaches its privileged point, its neighbor reaches its privileged point at least once. Thus, we define the specification of the alternator as follows:

Specification of the alternator. Starting from an arbitrary state, the program should reach states where the following conditions are satisfied.

- **Non-interference.** If a process is at its privileged point then none of its neighbors is at its privileged point.
- **Alternation.** If a process reaches its privileged point twice then its neighbors have reached their privileged points at least once. □

Note that the above specification is slightly different than that in [6–8]. Specifically, in [6–8], the non-interference condition is satisfied in all states. By contrast, the above specification requires that this condition be satisfied only in legitimate states. In read/write atomicity, the notion of privileged point depends *only* on the local state of the process. Thus, it is always possible to perturb the program to a state where two neighbors are at their privileged point. It follows that for read/write atomicity, it is impossible to guarantee non-interference in all states. Hence, we have weakened the specification accordingly.

3 Asynchronous Unison

Our solution for the alternator in read/write atomicity is based on the solution from [4]. Hence, in this section, we recall the program in [4] (cf. Figure 1) and mention its relevant properties. In this program, constant n denotes the number of processes in the system, and constant B is any natural number such that $B \geq n^2+1$. Additionally, in this program, each process, say j, maintains a public variable $x.j$ whose domain is $[0..B-1]$. It also maintains a private variable v and an array $g[w], 0 \leq w < d$ where d is the number of neighbors of j; $g[w]$, $0 \leq w < d$ denotes the w^{th} neighbor of j.

Each process j checks the clock values of its neighbors in sequence. If the clock value of j is less than that of its neighbor, say k, and the clock values of j and k are *close* then j checks the clock of its next neighbor, and so on. After j has gone through each of its neighbors successfully, it increments its clock. If there exists a neighbor, say k, such that $x.j$ and $x.k$ are *far apart* and $x.j$ is higher than $x.k$ then j resets $x.j$ to 0. Thus, the actions of process j in asynchronous unison are as shown in Figure 1:

$$
\begin{array}{lcl}
x.j \ beh \ x.g[v] \ \wedge \ v \neq d-1 & \longrightarrow & v := v+1 \\
x.j \ beh \ x.g[v] \ \wedge \ v = d-1 & \longrightarrow & x.j, v := (x.j+1) \ mod \ B, 0 \\
\exists k : 0 \leq k < d : (x.j \ far \ x.g[k] \ \wedge \ x.j > x.g[k]) & \longrightarrow & x.j, v := 0, 0
\end{array}
$$

where,

$$
\begin{array}{lll}
x.j \ beh \ x.g[v] & \text{iff} & ((x.g[v] - x.j) \ \ mod \ B) \leq n \\
x.j \ far \ x.g[v] & \text{iff} & \neg(x.j \ beh \ x.g[v]) \ \wedge \ \neg(x.g[v] \ beh \ x.j)
\end{array}
$$

Figure 1: Program for Asynchronous Unison

Now, we recall some of the properties of the above program. Theorem 3.1 states that in any state, at least one process is enabled. Theorem 3.2 states that each process will increment its clock infinitely often. And, Theorem 3.3 states that eventually the program reaches a state where the clock value of two neighboring processes differ by at most one. Moreover, in the subsequent computation, this property remains true. We refer the reader to [4] for proof.

Theorem 3.1 Every computation of the above program is infinite. □

Theorem 3.2 Every infinite computation of the above program has an infinite suffix where every clock variable $x.j$ is updated infinitely often by and only by executing $x.j := x.j + 1 \ mod \ B$. □

Theorem 3.3 Every infinite computation of the above program has an infinite suffix where every state satisfies the following condition:

(For every neighboring processes j and k

$$x.j = x.k \quad \vee \quad x.j = (x.k+1) \ mod \ B \quad \vee \quad x.k = (x.j+1) \ mod \ B)$$

□

4 Solution for the Alternator in Read/Write Atomicity

In this section, we use the program for asynchronous unison in Section 3 to obtain a program for the alternator in read/write atomicity. Observe that based on Theorem 3.3, in a legitimate state, the x values of neighboring processes differ by at most 1. Thus, the x values can be used to obtain a program for the alternator. The program in Figure 1, however, violates the specification of the alternator in three ways:

- The last two actions in asynchronous unison allow a process to read the x value of the neighbor and update its own x value. This violates the requirements of read/write atomicity.
- If the x values of two neighboring processes, say j and k, are equal then it is possible for both processes to update their x value simultaneously. This violates the first part of the alternator specification, namely, *non-interference*.
- If $x.j = x.k - 1$ then j can increment its clock twice before k increments its clock even once. This violates the second part of the alternator specification, namely *alternation*.

In the rest of the section, we first discuss our approach for dealing with the above issues and then present our program for the alternator.

Dealing with violation of read/write atomicity. Consider the second action in asynchronous unison where process j reads the x value of its last neighbor, say k. Now, if we introduce a delay between the reading of $x.k$ and updating of $x.j$, k can execute in between and increment $x.k$ at most once. Moreover, k can increment $x.k$ only if $x.j$ and $x.k$ were initially equal. Thus, if j reads $x.k$ and concludes that it can increment $x.j$, it can still do so even if k has incremented $x.k$ in between. Hence, for the second action, we can simply increment the value of v (as done in the first action). When v equals d, we can allow the process to increment its x value.

Now, consider the third action where process j reads the $x.k$ and decides to reset $x.j$ to 0. In this case, process k is disabled and cannot update $x.k$. Hence, in this action, we can set v to an *error* value and allow j to reset its clock later.

Dealing with violation of non-interference. We can achieve non-interference by introducing a tiebreak on process ID. Thus, if $x.j = x.k$ and $j < k$ then only j will be allowed to continue (to check the state of its next neighbor). In Theorem 4.2, we show that this modification does not affect the stabilizing fault-tolerance of asynchronous unison.

Dealing with violation of alternation. The modification proposed for non-interference also handles the issue of alternation. We show this in Theorem 4.4.

4.1 Solution

Based on the above modifications, we now describe our alternator program. In this program, the constants/variables d, n, B, x, g are similar that in the program for asynchronous unison (cf. Figure 1). To capture the notion of setting v to an error value, we add an additional value, $d+1$, to the domain of v. Now, we present the actions of process j in our alternator program in Figure 2.

$$
\begin{array}{lcl}
x.j\ beh'\ x.g[v] \ \wedge \ v < d & \longrightarrow & v := v + 1 \\
\exists k : 0 \leq k < d : (x.j\ far'\ x.g[k] \ \wedge \ x.j > x.g[k]) & \longrightarrow & v := d + 1 \\
v = d & \longrightarrow & x.j, v := (x.j + 1)\ mod\ B, 0 \\
v = d{+}1 & \longrightarrow & x.j, v := 0, 0
\end{array}
$$

where,

$$
\begin{array}{lll}
x.j\ beh'\ x.g[v] & \text{iff} & (0 < ((x.g[v] - x.j)\ mod\ B) \leq n) \ \vee \ (x.j = x.g[v] \ \wedge \ j < g[v]) \\
x.j\ far'\ x.g[v] & \text{iff} & \neg(x.j\ beh'\ x.g[v]) \ \wedge \ \neg(x.g[v]\ beh'\ x.j)
\end{array}
$$

Figure 2: Alternator Program in Read/Write Atomicity

4.2 Proof of Correctness

When a process executes the third action in the alternator program in Figure 2, it is at its privileged point. With this definition, we now show that the specification of the alternator is satisfied. The outline of the proof is as follows: First, we show that the program recovers to states from where the specification of asynchronous unison (Theorem 3.3) is satisfied. Then, we show that the condition '$((x.j + 1)\ mod\ B \ = \ x.k) \ \Rightarrow \ j{>}k$' is satisfied for all pairs of neighboring processes. Finally, we show that the specification of the alternator is satisfied.

Theorem 4.1 At any state of the alternator program in Figure 2, at least one process is enabled.

Proof. This proof is similar to that in [4]. Clearly, if there are two neighboring processes j and k such that $x.j\ far'\ x.k$ then either j or k is enabled. Thus, for a deadlock to occur, there must exist processes $j_1, j_2, ...j_l$ such that $(\forall i : 1 \leq i < l : x.j_i\ beh'\ x.j_{i+1})$ and $x.j_l\ beh'\ x.j_1$ are true.

Based on the definition of beh', we observe that the (circular) difference between $x.j_i$ and $x.j_{i+1}$ is at most n. Also, the number of processes in this cycle are also at most n. Since the bound on x value is

more than n^2+1, it follows that the (circular) difference between $x.j_1$ and $x.j_l$ must be more than n, i.e., $(x.j_1\ far'\ x.j_l)$ must be true. Since this is a contradiction, we can conclude that at least one process is enabled. □

Theorem 4.2 Every infinite computation of the program in Figure 2 has an infinite suffix where every state satisfies the following condition:

(For every neighboring processes j and k
$x.j = x.k \quad \vee \quad x.j = (x.k+1)\ mod\ B \quad \vee \quad x.k = (x.j+1)\ mod\ B$)

Proof. The program in Figure 2 is obtained by restricting execution of some actions from the program in Figure 1. Based on the deadlock freedom shown in Theorem 4.1, it follows that a computation of the alternator in 2 is also a computation of the asynchronous unison in Figure 1. Thus, from Theorem 3.3, the above theorem follows. □

Theorem 4.3 Every infinite computation of the program in Figure 2 has an infinite suffix where every state satisfies the following condition:

(For every neighboring processes j and $k :: ((x.j+1)\ mod\ B\ =\ x.k) \quad \Rightarrow \quad j>k$)

Proof. Consider the suffix of the computation where the x values of neighboring processes differ by at most 1. Now, consider the pairs $\langle j,k\rangle$ for which the above condition is not satisfied, i.e., $((x.j+1)\ mod\ B\ =\ x.k)$ but j is less than k. In this state, only process j can execute. Moreover, after j executes, the above condition is satisfied.

Now, consider a pair $\langle j,k\rangle$ for which the above condition is satisfied. Based on the restriction on x values from Theorem 4.2, we have $j>k$, $x.j=x.k$, or $((x.k+1)\ mod\ B\ =\ x.j)$. If $j>k$ then the above condition is always true. If $x.j=x.k$ then only the smaller ID process between j and k can execute. Thus, the above condition remains true for the pair $\langle j,k\rangle$. Finally, if $((x.k+1)\ mod\ B\ =\ x.j)$ then only k can execute. When k executes, $x.j$ equals $x.k$ and, hence, the above condition is satisfied for $\langle j,k\rangle$. It follows that if the above condition is true for $\langle j,k\rangle$ then it continues to be true. Based on the above discussion, it follows that the number of pairs for which the above condition is not satisfied decreases. Thus, eventually, the program will reach a state where the above condition is satisfied for all pairs $\langle j,k\rangle$. □

Theorem 4.4 The program in Figure 2 stabilizing fault-tolerant and it satisfies the specification of the alternator.

Proof. The legitimate states of the alternator are states where Theorem 4.2 and 4.3 are satisfied. It follows that starting from arbitrary states, the program reaches legitimate states. Now, we show that from these states, the specification of the alternator is satisfied. To show this, consider any two neighboring processes, j and k. Wlog, $j>k$. In legitimate states, if $x.j=x.k$ then only k can execute. And, if $x.j\neq x.k$ then only j can execute. Also, after j (respectively, k) executes it cannot execute again until k (respectively, j) executes. Thus, in legitimate states, the specification of the alternator is satisfied. □

Theorem 4.5 The RAM space (after ignoring the space for constants) used by the alternator program is $O(log\ n)$. □

5 Revisiting Concurrent Execution of Alternator

The alternator program introduced in the previous section is stabilizing fault-tolerant if it is executed in read/write atomicity. In this section, we identify a corresponding program whose execution in the concurrent execution model achieves maximal concurrency for linear, tree and bipartite topology. The program in Figure 2 is the implementation of this program in read/write atomicity. Thus, this program adds stabilizing tolerance in read/write atomicity to [6,8] without losing maximal concurrency.

First, we simplify the program in Figure 2 so that each process reads the state of its neighbors at once. With such simplification, process j can at once check if $x.j\ beh'\ x.g[v]$ is true for all its neighbors. If this

condition is true then j can increment its x value. On the other hand, if j finds a neighbor k such that $x.j\ far'\ x.g[k]$ and $x.j > x.g[k]$ then j resets $x.j$ to 0. Thus, the simplified program is as shown in Figure 3.

$$\begin{array}{lll} \forall v : 0 \leq v < d : x.j\ beh'\ x.g[v] & \longrightarrow & x.j := (x.j + 1)\ mod\ B \\ \exists k : 0 \leq k < d : (x.j\ far'\ x.g[k] \ \wedge\ x.j > x.g[k]) & \longrightarrow & x.j := 0 \end{array}$$

Figure 3: Program that (1) satisfies the alternator specification in read/write atomicity, and (2) provides maximal concurrency in the concurrent execution model for line, tree, and bipartite topology

It is straightforward to see that the program in Figure 2 is an implementation of the above program in read/write atomicity. Now, we show that in legitimate states, the above program provides maximal concurrency if we use concurrent execution model and the underlying topology is a line, tree, or more generally, a bipartite graph. Towards this end, we first make an observation about the legitimate states of the alternator. From Theorem 4.2, we recall that eventually the alternator program reaches states where the x values of neighboring processes differ by at most one. Moreover, eventually it reaches states where '$(x.j + 1)\ mod\ B = x.k \Rightarrow j > k$'. Thus, in legitimate states, the execution of the alternator at process j can be expressed as follows:

$$\forall k : k \text{ is a neighbor of } j : (k > j \Rightarrow x.j = x.k) \ \wedge\ (k < j \Rightarrow (x.j + 1) = x.k) \longrightarrow x.j = (x.j + 1)\ mod\ B$$

5.1 Mapping to Linear Alternator

Given a set of processes arranged in a line, we now show how the alternator program in Figure 3 can be mapped into the linear alternator program in [6]. In this mapping, we number the processes as follows: the leftmost process has the smallest ID and the IDs increase as we move from left to right. Also, we assume that the value of B used in the alternator program in Figure 3 is even. Now, for obtaining the required mapping, we introduce an auxiliary boolean variable $b.j$ which is true iff $x.j$ is even. Thus, incrementing $x.j$ in modulo B arithmetic is equivalent to complementing $b.j$. Now, the alternator at j is at its privileged point iff the following condition is true.

$$\begin{array}{ll} & \forall k : k \text{ is a neighbor of } j : (k > j \Rightarrow x.j = x.k) \ \wedge\ (k < j \Rightarrow (x.j + 1) = x.k) \\ \equiv & \quad \text{Based on the definition of } b \text{ and constraints on } x \text{ values} \\ & \forall k : k \text{ is a neighbor of } j : (k > j \Rightarrow b.j = b.k) \ \wedge\ (k < j \Rightarrow b.j \neq b.k) \\ \equiv & \quad \text{Based on the linear topology and IDs of processes} \\ & b.j = b.(rightneigbor.j) \ \wedge\ (b.j \neq b.(leftneighbor.j)) \end{array}$$

Thus, after reaching legitimate states, computations of the alternator in Figure 3 are identical to those in [6]. It follows that in the concurrent execution model, the alternator in Figure 3 provides maximal concurrency if the topology is linear.

5.2 Mapping to Tree Alternator

Now, we extend the mapping in Section 5.1 to a tree. In this mapping, we number the processes as follows: the root process has the lowest ID, and IDs increase as we move from the root to the leaves. Once again, we introduce the auxiliary boolean variable b as in previous subsection. Now, based on the mapping in Section 5.1, the alternator at j is at its privileged point iff the following condition is satisfied.

$$\begin{array}{ll} & \forall k : k \text{ is a neighbor of } j : (k > j \Rightarrow b.j = b.k) \ \wedge\ (k < j \Rightarrow b.j \neq b.k) \\ \equiv & \quad \text{Based on the linear topology and IDs of processes} \\ & (\forall k : k \text{ is a child of } j : b.j = b.k) \ \wedge\ (b.j \neq b.(parent.j)) \end{array}$$

This program is identical to that in [8] and, hence, in the concurrent execution model, the alternator program in Figure 3 provides maximal concurrency if the topology is a tree.

Based on the above discussion, for linear and tree topology, our program provides guarantees depending upon the underlying model. If read/write atomicity is used, it will satisfy the specification of the alternator. If concurrent execution model is used then, in addition to the specification of alternator, it will provide maximal concurrency. To our knowledge, this is the first alternator program that achieves both these properties.

5.3 Mapping to Bipartite Graphs

Now, we extend the above mapping for bipartite graphs. In this mapping, we embed a tree in the given graph and number the nodes as in Section 5.2. Now, if we execute the program in Figure 3 by ignoring the non-tree edges then, as shown above, maximal concurrency will be provided. Moreover, if there is a non-tree edge between j and k then |(distance between j and the root) - (distance between k and the root)| is odd. It follows that j and k will not be enabled simultaneously. In other words, non-interference and alternation will be satisfied with the original neighborhood relation.

Thus, for a bipartite graph, in the concurrent execution model, eventually, the alternator program in Figure 3 reaches states from where maximal concurrency is provided. It follows that, our program provides maximal concurrency in meshes and hypercubes, etc.

Note that for arbitrary bipartite graphs, if we ignore the non-tree edges, then the specification of alternation may be violated while executing the program in read/write atomicity. However, in the concurrent execution model, the program will satisfy the specification of the alternator and provide maximum concurrency. Thus, for arbitrary bipartite graphs, our program provides a choice: (1) ability to satisfy the specification of the alternator in read/write atomicity but to lose maximal concurrency, or (2) ability to provide maximal concurrency in concurrent execution model.

6 Conclusion

In this paper, we presented a stabilizing program for the alternator problem in read/write atomicity. Our program uses bounded timestamps whose size is $O(log\ n)$ where n is the number of processes in the system.

For arbitrary topology, starting from any state, if our program is executed in read/write atomicity then it recovers to states from where the specification of the alternator is satisfied. In addition, for the linear/tree topology, if concurrent execution model is used then, it provides maximal concurrency. Thus, our program enhances the solutions in [6, 8] by making those solutions stabilizing fault-tolerant in read/write atomicity while preserving their maximal concurrency. Moreover, the *cost* of adding stabilizing fault-tolerance in read/write atomicity, $O(log\ n)$ space, is reasonable. Also, as discussed in Section 5, to our knowledge, this is the first solution for alternator that provides maximal concurrency in the concurrent execution model and satisfies the specification of the alternator in read/write atomicity. Moreover, our program is simple to implement as it only maintains a single variable on which only simple operations (subtract the value of the variable from that of the neighbor, increment the variable by 1 and reset the variable to 0) are performed.

For arbitrary *bipartite* graphs, our algorithm provides a choice: (1) ability to satisfy the specification of the alternator in read/write atomicity but to lose maximal concurrency, or (2) ability to provide maximal concurrency but to require concurrent execution model. Whether both these properties could be obtained simultaneously in bipartite graphs (respectively, arbitrary graphs) remains open.

References

[1] G. Antonoiu and P. Srimani. Mutual exclusion between neighboring nodes in an arbitary system graph that stabilizes using read/write atomicity. *EuroPar, Lecture Notes in Computer Science*, pages 823–830, 1999.

[2] J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *Proceedings of the Thirteenth International Symposium on Distributed Computing*, 2000.

[3] S. Cantarell, A. K. Datta, and F. Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. *Symposium on Self-stabilization*, 2003.

[4] J. Couvreur, N. Francez, and M. G. Gouda. Asynchronous unison. *International Conference on Distributed Computing Systems*, pages 486–493, 1992.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[6] M. G. Gouda and F. F. Haddix. The linear alternator. *Workshop on Self-Stabilizing Systems*, 1997.

[7] M. G. Gouda and F. F. Haddix. The alternator. *Workshop on Self-Stabilizing Systems*, 1999.

[8] M. G. Gouda and F. F. Haddix. Tree embedded alternators. Presentation at Seminar on Self-stablization, Lumini, Marseille, France, 2002.

[9] M. Mizuno and H. Kakagawa. A timestamp based transformation of self-stabilizing program for distributed computing environments. *Workshop on Distributed Algorithms*, 1996.

[10] M. Mizuno and M. Nesternko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.

[11] M. Nesternko and A. Arora. Stabilization preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 2002. A preliminary version of this paper appears in DISC'99.