Parallel Processing Letters

TERMINATING ALTERNATOR

SANDEEP S. KULKARNI*
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824, USA

Received August 2004
Revised July 2007
Communicated by S. G. Akl

ABSTRACT

In this paper, we present an program that enables the transformation of a non-terminating alternator into a terminating alternator. Our solution is stabilization preserving and has the potential to preserve maximal concurrency (if available) provided by the non-terminating alternator. It can also be used to transform a program that is stabilizing in interleaving semantics into a program that is stabilizing in powerset semantics. We also discuss how the terminating alternator can be used to enable a process to gain an understanding of system stability.

1. Introduction

The problem of alternator was introduced in [8,9,10] to transform programs in interleaving semantics – where actions are executed in an interleaving manner into powerset semantics– where any subset of enabled actions are executed concurrently. Moreover, these solutions are especially useful since they ensure that if the alternator is executed on certain topologies in maximum-parallelism semantics – where all enabled actions are executed concurrently, then the program reaches states where the number of enabled processes is always maximal. Furthermore, starting from such states, in the maximum-parallelism model, the set of enabled processes is always maximal.

However, the alternators in [8,9,10,13] suffer from the fact that they do not preserve *fixpoint* property. In other words, even if the original program in interleaving semantics reaches a fixpoint state where none of its actions are enabled, the transformed program does not reach a fixpoint. Rather, the alternator continues to execute forever. Fixpoint preservation is desirable for several reasons. For one, it allows use of Stabilization by layers [6,11] that require that all components in

*Address: 3115 Engineering, East Lansing MI 48864
Email: sandeep@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 Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

a hierarchy, except the last one, reach a fixpoint. Also, the fixpoint preservation property is desirable in terms of simplicity and communication overhead [12].

While existing solutions for self-stabilizing mutual exclusion (e.g., [15,16]) provide the fixpoint preservation property, they do not provide other properties that alternators in [8,13] provide. For example for linear topology, the alternator in [8] ensures that if the program is executed with maximum parallelism semantics, then it has a suffix where in all states the set of enabled processes is maximal. In [13], the same property is also provided for bipartite graphs. Thus, these alternators can improve the efficiency of the transformed program.

Based on this motivation, we focus on transforming a given non-terminating alternator into a terminating alternator. Our transformation preserves the stabilization property of the non-terminating alternator. And, it has the potential to preserve the maximal concurrency of the given non-terminating alternator.

Organization of the paper. This paper is organized as follows: First, we describe the model and problem statement in Section 2. Then, in Section 3, we present our transformation program. We illustrate our transformation program with an example in Section 4. Finally, we discuss an application of terminating alternator in Section 5 and conclude 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 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. A state of the program is obtained by assigning each variable in the program some value in its domain. We say that an action is enabled in a state if its guard evaluates to true in that state. For simplicity of presentation, we introduce a notion of priority for each action; an action of priority n is executed only when guards of all actions of priority greater than n evaluate to false. Thus, priority can be modeled by appropriately changing guards of actions with lower priority. Given a program, its computations are determined by the type of semantics used in its execution. We consider three types of semantics, described next.

Interleaving semantics. Given a program p, its computation in the interleaving semantics is of the form $\langle s_0, s_1, ... \rangle$ where state s_i, $i > 1$, is obtained by executing any *one* action that is enabled in s_{i-1}.

Maximum-Parallelism semantics. Given a program p, its computation in the maximum-parallelism semantics is of the form $\langle s_0, s_1, ... \rangle$ where state s_i, $i > 1$, is obtained by executing *all* actions (at most one per process) that are enabled in s_{i-1}.

Powerset semantics. Given a program p, its computation in the powerset semantics is of the form $\langle s_0, s_1, ... \rangle$ where state s_i, $i > 1$, is obtained by executing *any subset of* actions (at most one per process) that are enabled in s_{i-1}.

Legitimate states of a program are states from where all its computations sat-

isfy its specification. The notion of stabilization [5,6] requires that any program computation eventually reaches its legitimate states and then remains in legitimate states forever. Formally,

Stabilization. We say that a program is stabilizing [5,6] iff starting from an arbitrary state, the program eventually recovers to legitimate states and once the program reaches legitimate states, it continues to be in legitimate states.

The problem of alternator was introduced in [8,9,10] to convert a self-stabilizing program in interleaving semantics into a self-stabilizing program into powerset semantics. To achieve this conversion, the actions of process j in original programs are restricted to execute only when the alternator at j can execute. Towards this end, the alternator has one *privileged action* per process. An action of j in the original program (in interleaving semantics) is executed only if the guard of its privileged action also evaluates to true. To ensure that the conversion preserves stabilizing fault-tolerance, the specification of the non-terminating alternator is defined as follows:

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

- **Non-interference.** If the privileged action at a process is enabled then the privileged actions of its neighbors are not enabled.
- **Alternation.** If the privileged action of one process is executed twice in any computation then, in between, its neighbors have executed their privileged action at least once.
- **Progress (and Non-termination).** In any computation, the privileged actions of a process execute infinitely often. □.

Observation 1 Let p' be the program obtained by removing actions of one process, say j, from an alternator program. Any computation of p' has a suffix where no privileged action is enabled.

Proof. This observation follows from the alternation requirement. □.

Specification of (terminating) alternator. To obtain the specification of the terminating alternator, we introduce a variable *requesting.j* that denotes whether some action of j in the original program (in interleaving semantics) is to be executed. Thus, if the original program reaches a fixpoint then for any process j, *requesting.j* is false. Now, we revise the progress requirement so that the alternator actions at j need not be executed when *requesting.j* is false. Furthermore, we require that if the original program reaches a fixpoint then so does the alternator. Thus, the specification of the terminating alternator is as follows:

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

- **Non-interference.** Same as non-terminating alternator.
- **Alternation.** Same as non-terminating alternator.

- **Progress.** In any computation where *requesting.j* is continuously true, eventually the privileged action at j is executed.
- **Termination.** In any computation where *requsting.j* is false for *all* processes, eventually, the computation reaches a fixpoint, i.e., all actions (including the privileged actions) of the program are disabled. □.

We assume that if *requesting.j* ever becomes true then it remains true until j executes its privileged action. Also, the alternator program cannot modify *requesting.j*; it can only read it.

3. Transformation Program

Our transformation program takes two programs, one for non-terminating alternator and one for stabilizing tree correction, as input. We make the following assumptions about these programs.

- Alternator program satisfies the specification of the non-terminating alternator.
- The tree program has the following properties:
  - Each process j has a unique ID, say $ID.j$.
  - Each process j has a variable, say *root.j*, *root.j* is the ID of the process that j believes to be the root.
  - Each process j has a variable *P.j* that is the parent of j in the tree. The program constructs a tree that is rooted at the node with highest ID.
  - The program eventually it reaches a fixpoint where none of its actions are enabled. Moreover, the program reaches a fixpoint state only if the *root* value of all processes equal the highest ID process.
  - The only operations performed on the ID variable are (1) assignment and comparison for equality of one ID variable with another, or (2) comparison of one ID variable with another using $<$ operator. For example, the tree construction program can evaluate predicates such as $ID.j < ID.k$. However, it cannot evaluate formulae such as $ID.j - ID.k$, $ID.j + 5$, $ID.j + ID.k$, $ID.j = 5$, or $ID.j < 5$. Moreover, the same requirement also applies to any other variable (e.g., *P.j* and *root.j*) that is updated based on the ID value of any process. Note that this property can be easily checked by *syntactic* analysis of the tree program. Finally, since collecting a list of IDs in an array, set or sequence, checking for membership for a given ID in a set of IDs can be performed using the first operation (assignment/comparison), for simplicity of presentation, we allow these operations as well.
  - Both programs are self-stabilizing in the powerset semantics.
  - The variables in both programs are disjoint (or appropriate renaming is done to achieve this disjointness).

To obtain the terminating alternator, we modify the tree and alternator program to achieve the following properties:

- If there is any process that wants to execute its privileged action, then the tree should be rooted at some process that wants to execute its privileged action.
- If tree is rooted at a process that does not want to execute its privileged action, then actions of the (non-terminating) alternator should be disabled.

To achieve these properties, we perform three steps: (1) modifying the tree program, (2) modifying the alternator, and (3) introduce actions that allow the process to declare its intention to execute its privileged action.

Step 1: Ensure that tree is rooted at a process (if one exists) that wants to execute its privileged action. We modify the tree program so that the root process will always be a process that wants to execute its privileged action. We achieve this by modifying the ID of the process; if process j is interested in executing its privileged action, its ID would be of the form $\langle 1, ID.j\rangle$. Otherwise, its ID would be of the form: $\langle 0, ID.j\rangle$. Based on the nature of the tree computation, the root would eventually be of the form $\langle 1, x\rangle$ iff at least one process is interested in executing its privileged action. Thus, to modify the tree program, we proceed as follows: First, we replace the variable $ID.j$ from the tree program by a pair $\langle l.j, ID.j\rangle$, where $l.j$ is a new variable with domain $\{0, 1\}$. Of the two operations permitted on ID variables, the semantics of the first operation is straightforward. For the second operation, the comparison of two IDs, we use lexicographic comparison. Thus, $\langle l.j, ID.j\rangle < \langle l.k, ID.k\rangle$ iff $(l.j < l.k) \ \vee \ ((l.j = l.k) \wedge (ID.j < ID.k))$. These revised actions are included to obtain the terminating alternator. We define that these actions are of priority 1. (Note that since the tree program reaches a fixpoint where the root has the highest ID, it follows that if there is any process that wants to execute its privileged action, then the tree should be rooted at some process that wants to execute its privileged action.)

Step 2: Stop the alternator if tree is rooted at a process that does not want to execute its privileged action. We modify the actions of the non-terminating alternator and include the modified actions in the terminating alternator. We introduce a special value, $\perp$, to the domain of ID variables. We also define that $\perp$ is smaller than any ID variable. Thus, $root.j > \langle 1, \perp\rangle$ is true iff $root.j$ is of the form $\langle 1, x\rangle$. Now, we restrict each action of process j in the non-terminating alternator to execute only when $root.j > \langle 1, \perp\rangle$. And, the privileged action of j in terminating alternator is the action obtained by modifying the corresponding action in the non-terminating alternator. We define that these privileged actions are of the next higher priority, say 2, and the other alternator actions are of priority 0. (Note that this ensures that if the tree eventually reaches a state where the root value is of the form $\langle 0, x\rangle$ then all actions of alternator will be disabled.)

Step 3: Declare intent to execute privileged action. Finally, we also introduce two additional actions for each process j. These actions have the highest priority, say 3.

$$requesting.j \wedge l.j = 0 \quad \longrightarrow \quad l.j := 1;$$
$$\neg requesting.j \wedge l.j = 1 \quad \longrightarrow \quad l.j := 0;$$

Remark. The priorities assigned above ensure that if $requesting.j$ ever becomes true (respectively, false), eventually $l.j$ is set to 1 (respectively, 0). Afterwards, if the privileged action of j is enabled, it can be executed. If the privileged action at j is disabled then the tree actions will execute and satisfy the predicate $root.j > \langle 1, \perp \rangle$. Hence, privileged action of j can be eventually executed. Finally, from the non-interference condition of the alternator, the privileged actions cannot be continuously enabled, it allows the tree correction actions at j to execute. Hence, if eventually $l.j$ is set to 0 for all processes then the tree reconstruction will restore the tree to be rooted at $\langle 0, x \rangle$. At this point, all alternator actions would be disabled.

3.1. Correctness Proof

To show the correctness of the terminating alternator identified above, we prove the following three theorems. The first theorem proves that the program obtained after transformation satisfies the specification of the terminating alternator. The second theorem proves that the transformed program preserves maximal concurrency if $requesting.j$ is continuously true for all processes. Finally, the third theorem shows that the transformed program can be used to convert a program in interleaving semantics into a program in powerset semantics.

Theorem 2 The terminating alternator obtained by the transformation from Section 3 satisfies the specification of the terminating alternator.

Proof. It is straightforward to observe that the Non-interference and Alternation requirements of the non-terminating alternator are preserved while obtaining the terminating alternator. Hence, we focus on Progress and Termination.

Progress. Consider the case where $requesting.j$ is continuously true for some process j and the privileged action is never executed by j. Since $requesting.j$ is true, $l.j$ would be set to 1. Furthermore, based on Observation 1, we observe that eventually all alternator actions will be disabled. Hence, the tree actions will eventually execute until a fixpoint is reached. Based on the property of the tree program, the root of the tree would be of the form $\langle 1, x \rangle$ where x is the ID of some process. Thus, $(\forall k : root.k > \langle 1, \perp \rangle)$ will be true. Now, the restriction imposed on the actions of the non-terminating alternator is lifted, i.e., if the guard of some action in the non-terminating alternator is true then the corresponding action in the terminating alternator can execute. Moreover, based on the non-termination property of the non-terminating alternator, eventually, the guard of some privileged action evaluates to true. This contradicts the previous observation that all alternator actions are disabled. Hence, Progress is satisfied.

Termination. If the original program reaches a fixpoint, i.e., $\forall k : requesting.k = false$ then eventually the program reaches a state where $\forall k : l.k = 0$. Hence, the tree correction will cause the *root* values of all processes to be of the form $\langle 0, x \rangle$. In such a state, the actions added for the non-terminating alternator are disabled. Thus, only tree actions can execute. Since the tree is guaranteed to reach a fixpoint with a root value of the form $\langle 0, x \rangle$, the terminating alternator reaches a fixpoint. □.

Theorem 3 Let c be a computation of the terminating alternator. If c has a suffix where *requesting* variable of all processes is continuously true then c satisfies the specification of the non-terminating alternator.

Proof. If $\forall k : requesting.k$ is true continuously then $l.k = 1$ is also continuously true. Thus, the $\langle l.j, ID.j\rangle$ value for any process j is unchanged. Hence, the tree would eventually remain at its fixpoint. Therefore, $root.j > \langle 1, \bot\rangle$ would be true for all processes. Finally, if an action of the non-terminating alternator is enabled in some state then the corresponding action of the terminating alternator is also enabled in the corresponding state. Thus, the theorem follows. □.

Preserving maximality of non-terminating alternator. Consider the case where the non-terminating alternator ensures that if the alternator is executed with maximum parallelism semantics, then it has a suffix where in all states the set of enabled processes is maximal. Now, based on the above theorem if *requesting* variable is always true in the computation of the terminating alternator then under maximum-parallelism semantics, that computation has a suffix where in all states the set of enabled processes is maximal. As mentioned above, such maximality property is provided for linear graphs by the alternator in [8]. And, this property is provided by the alternator in [13]. Thus, the use of these alternators in obtaining terminating alternator will preserve the property.

Remark on implementation. Based on the above theorem, if *requesting* variable of all processes is continuously true and the non-terminating alternator provides maximal concurrency then so does the terminating alternator. However, if a process suspects that the program has reached a fixpoint (e.g., because $root.j$ is of the form $\langle 0, x\rangle$), then maximal concurrency may not be provided. We can utilize this fact in the implementation. Observe that if $requesting.j$ alternates between true and false then the tree correction may disable the alternator actions, if the tree correction causes the process to switch to a tree rooted at $\langle 0, x\rangle$. To avoid this problem, we modify the program as follows: If $requesting.j$ ever becomes true and $l.j$ is set to 1 then $l.j$ remains 1 until some threshold number of alternator actions are executed. This will force tree construction where $root.j > \langle 1, \bot\rangle$ is true for all processes. In that case, the alternator actions can execute freely. We can also modify the above program so that when $root.j < \langle 1, \bot\rangle$ becomes true, the alternator continues to execute a certain threshold of privileged actions. Thus, if the original program has not yet terminated then eventually $root.j > \langle 1, \bot\rangle$ will be true, allowing the actions of the non-terminating alternator to execute. With this modification, the activity in the tree reconstruction could be reduced at the expense of the delay in reaching the fixpoint. Moreover, if the thresholds equal ∞, the terminating alternator behaves in the same way as the non-terminating alternator. Thus, it is possible to obtain a tradeoff between the period of maximal concurrency and the delay for termination.

Theorem 4 Given a program that is stabilizing in the interleaving semantics, if it is modified so that its actions execute only when the privileged action at the corresponding process can execute, then the resulting program is stabilizing fault-tolerant in powerset semantics.

Proof. Based on the proof of Theorem 2, if an action of the original program needs to be executed, it will be eventually executed. Also, for a computation of the

terminating alternator in powerset semantics, there is a corresponding computation of the non-terminating alternator. Hence, eventually Non-interference will be satisfied. Subsequently, the execution of multiple actions of the original program can be effectively serialized. Thus, the theorem follows. □.

4. Example

In this section, we present an example for terminating alternator that is based on the linear alternator [8] and the stabilizing tree correction program [2]. First, we describe the actions of these programs. Then, we apply the transformation program from Section 3 to obtain the program for terminating alternator.

Non-terminating alternator. The solution for alternator in [8] is for a linear topology where the processes $1..n$ are arranged in a line. Each process, say j, maintains a variable $x.j$ with domain $\{0,1\}$. The program contains one action per process; this is the privileged action. Process j executes its action when its value differs from its predecessor (if one exists) and it matches the successor (if one exists). Upon execution, it toggles the value of $x.j$. Thus, the actions of the non-terminating alternator are as follows: (In this program, j ranges from $2..n-1$.)

$$
\begin{array}{lcl}
x.1 = x.2 & \longrightarrow & x.1 := 1 - x.1 \\
x.j \neq x.(j-1) \quad \wedge \quad (x.j = x.(j+1)) & \longrightarrow & x.j := 1 - x.j \\
x.(n-1) \neq x.n & \longrightarrow & x.n = 1 - x.n
\end{array}
$$

Theorem 5 Starting from any state, under maximum-parallelism semantics, the program eventually reaches a state where the number enabled processes is maximal [8]. Moreover, after such a state is reached, this property is preserved if the program is executed in maximum-parallelism semantics. □.

Tree Program. We consider the program from [2] where process j maintains $ID.j$ (unique ID as required), $root.j$ (the ID of the process that j believes is the root), $P.j$ (parent of j in the tree), $N.j$ (set of neighbors of j), $root.j.k$ (a copy of the $root.k$ maintained at j), $d.j$ (distance of j from the root in the tree), and $d.j.k$ (copy of $d.k$ maintained at j). Of these, all variables except $d.j$ and $d.j.k$ are updated based on ID value of some process. Hence, they are required to follow the constraints identified in Section 3. It also maintains a constant, D, that is (an upper bound) on the diameter of the network.

The program contains four actions. The first action considers the case where local constraints on variables of j are violated. For example, if $root.j$ is smaller than the ID of j, distance of a root process from itself is non-zero, or $P.j$ is not a neighbor of j then j resets its variables to obtain local consistency. The second action compares whether j agrees with its parent, i.e., their root values are identical and $d.j$ is one more than $d.(P.j)$. If these constraints are violated, j updates its variables accordingly. The third action allows j to change its parent; if some process is providing a higher root value or a lower distance to the current root, j changes its parent. Finally, the fourth action allows j to update its own copies based on the values of variables at k. The actions of the program are as follows:

$(root.j < ID.j) \ \vee$
$(P.j = j \ \wedge \ (root.j \neq ID.j \ \vee \ d.j \neq 0)) \ \vee$
$(P.j \notin (N.j \cup \{ID.j\}) \ \vee \ d.j > B)$
$\longrightarrow \quad root.j, P.j, d.j := ID.j, ID.j, 0$

$P.j = ID.k \ \wedge \ ID.k \in N.j \ \wedge \ d.j < D \ \wedge$
$(root.j \neq root.j.k \ \vee \ d.j \neq d.j.k + 1)$
$\longrightarrow \quad root.j, d.j \ := root.j.k, d.j.k + 1$

$(root.j < root.j.k \ \wedge \ ID.k \in N.j \ \wedge \ d.j.k < D) \ \vee$
$(root.j = root.j.k \ \wedge \ ID.k \in N.j \ \wedge \ d.j.k + 1 < d.j)$
$\longrightarrow \quad root.j, P.j, d.j := root.j.k, ID.k, d.j.k + 1$

$ID.k \in N.j \ \wedge$
$(root.k \neq root.j.k \ \vee \ P.k \neq P.j.k \ \vee \ d.k \neq d.j.k)$
$\longrightarrow \quad root.j.k, P.j.k, d.j.k := root.k, P.k, d.k$

Theorem 6 The above program ensures [2] that (1) starting from an arbitrary state, the program eventually recovers to states where the parent (P) relation forms a rooted tree and (2) a fixpoint is reached only if the value of *root* variable equals the highest ID of the process. □.

Remark. One could argue that the above tree program is an *overkill* in the sense that a much simpler tree construction program can be designed if one assumes the linear topology used by the non-terminating alternator. However, we have chosen this example to illustrate our claim that *any* non-terminating alternator (e.g., [8,9,10,13]) and *any* tree construction program (cf. [7]) can be used to design a terminating alternator if the conditions from Section 3 are satisfied.

Terminating alternator. By applying the transformation program in 3, the program for terminating alternator is as follows:

Actions of priority 3

$requesting.j \wedge l.j = 0 \quad \longrightarrow \quad l.j := 1;$
$\neg requesting.j \wedge l.j = 1 \quad \longrightarrow \quad l.j := 0;$

Actions of priority 2

$root.1 > \langle 1, \bot \rangle \ \wedge \ x.1 = x.2 \quad \longrightarrow \quad x.1 := 1 - x.1$
$root.j > \langle 1, \bot \rangle \ \wedge \ x.j \neq x.(j-1) \ \wedge \ (x.j = x.(j+1)) \quad \longrightarrow \quad x.j := 1 - x.j$
$root.n > \langle 1, \bot \rangle \ \wedge \ x.(n-1) \neq x.n \quad \longrightarrow \quad x.n = 1 - x.n$

Actions of priority 1

$(root.j < \langle l.j, ID.j \rangle) \ \vee$
$(P.j = j \ \wedge \ (root.j \neq \langle l.j, ID.j \rangle \ \vee \ d.j \neq 0)) \ \vee$
$(P.j \notin (N.j \cup \{\langle l.j, ID.j \rangle\}) \ \vee \ d.j > B)$
$\longrightarrow \quad root.j, P.j, d.j := \langle l.j, ID.j \rangle, \langle l.j, ID.j \rangle, 0$

$$P.j = \langle l.k, ID.k\rangle \ \wedge \ \langle l.k, ID.k\rangle \in N.j \ \wedge \ d.j < D \ \wedge$$
$$(root.j \neq root.j.k \ \ \vee \ \ d.j \neq d.j.k + 1)$$
$$\longrightarrow \quad root.j, d.j \ := root.j.k, d.j.k + 1$$

$$(root.j < root.j.k \ \ \wedge \ \ \langle l.k, ID.k\rangle \in N.j \ \ \wedge \ \ d.j.k < D) \ \vee$$
$$(root.j = root.j.k \ \ \wedge \ \ \langle l.k, ID.k\rangle \in N.j \ \ \wedge \ \ d.j.k + 1 < d.j)$$
$$\longrightarrow \quad root.j, P.j, d.j := root.j.k, \langle l.k, ID.k\rangle, d.j.k + 1$$

$$\langle l.k, ID.k\rangle \in N.j \ \wedge$$
$$(root.k \neq root.j.k \ \vee \ P.k \neq P.j.k \ \vee \ d.k \neq d.j.k)$$
$$\longrightarrow \quad root.j.k, P.j.k, d.j.k := root.k, P.k, d.k$$

5. Application of Terminating Alternator

One application of the terminating alternator is to allow a process to get an understanding of system stability. It is well known that no process can ever conclude that if overall system state is legitimate [6]. However, the terminating alternator can allow a process to identify what percentage of time is spent in legitimate states and what percentage of time is spent outside legitimate states. To illustrate this issue, consider a stabilizing program that reaches a fixpoint. In this case, in legitimate states, no program actions are enabled. Thus, if a program action is executed then it implies that the system is not in a legitimate state.

Note that for an arbitrary stabilizing program, even if one process executes an action, other processes may never notice that the system was outside legitimate states. This is due to the fact that many self-stabilizing programs (e.g., [16]) ensure that if the fault affects a small set of processes then only a small set of processes are involved in the recovery to legitimate states. Hence, processes not involved in recovery may not be aware of the fault.

By contrast, the addition of terminating alternator to a program can ensure that all processes are aware of a fault. Towards this end, we observe that whenever j needs to execute an action, it sets $l.j$ to 1. Now, if $l.j$ remains 1 until t executions of the alternator where t is the number of steps necessary for the recovery of the tree program, it will force all processes to change to a root process of the form $\langle 1, x\rangle$. Hence, all processes would be aware of the fault. And, if multiple faults occur *concurrently* then they would be detected as a single state perturbation. Moreover, once a fixpoint is reached, the *root* value of a process would be of the form $\langle 0, x\rangle$. Thus, by identifying whether *root* value is of the form $\langle 1, x\rangle$, a process can get an understanding of how frequently the system is in legitimate states.

The above approach can also be applied for certain programs that do not reach a fixpoint. To illustrate this issue, consider the case where a program actions can be partitioned into *closure actions* and *convergence actions* [3]. Closure actions are (meant to be) executed in legitimate states and convergence actions are executed only for recovery from illegitimate states. For the case where closure and convergence actions do not overlap (i.e., the only actions necessary for recovery are the convergence actions and the convergence actions are disabled inside the legitimate states), the terminating alternator could be used to gain understanding of how long the system is in a legitimate state. Towards this end, we use the terminating

alternator only for the convergence actions.

6. Conclusion

In this paper, we presented a transformation program for obtaining a terminating alternator based on a non-terminating alternator. Our transformation was based on adding a stabilizing silent tree correction program. We showed how this terminating alternator could be used to convert a program in interleaving semantics into a program in powerset semantics. We showed that this conversion preserves stabilizing fault-tolerance. Furthermore, we argued how the transformation has the potential to preserve maximal concurrency (if available) provided by the non-terminating alternator.

Problems closely related to the alternator problem include local mutual exclusion and dining philosophers [1,4,14,15,16]. Similar to the alternators, these solutions ensure that when a process executes its privileged action, none of its neighbors can execute their privileged action. The main difference between these solutions and the alternator is that these programs aim to achieve local recovery. For example, if only a small set of processes are interested in critical section, these solutions try to ensure that only a small subset of processes need to be involved. By contrast, the goal of the alternator is to ensure that everyone is involved and there is some correlation between the executions of neighboring processes. Hence, as discussed in Section 5, the terminating alternator can be used to get an understanding of how frequently the system is in legitimate states.

Acknowledgments

We would like to thank the reviewers of the paper for their comments on the paper.

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] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[3] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant comput ing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[4] 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.

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

[6] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[7] Felix C. Gartner. A survey of self-stabilizing spanning tree construction algorithms. Technical Report IC/2003/38, Swiss Federal Institute of Technology (EPFL), 2003.

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

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

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

[11] T. Herman. *Adaptivity Through Distributed Convergence*. PhD thesis, University of Texas at Austin, 1991.

[12] Lisa Higham and Colette Johnen. Self-stabilizing implementation of atomic register by regular register in network framework. Technical Report 1449, LRI, 2006.

[13] Sandeep S. Kulkarni, Chase Bolen, John Oleszkiewicz, and Andrew Robinson. Alternator in read/write model. *Information Processing Letters*, 2005.

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

[15] 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.

[16] 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.