Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Processing Letters 93 (2005) 207–215
Information Processing Letters
www.elsevier.com/locate/ipl

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

Received 29 September 2004; received in revised form 22 November 2004

Available online 29 December 2004

Communicated by F.B. Schneider

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; Algorithms; Concurrency

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

☆ 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.

* Corresponding author.

E-mail addresses: sandeep@cse.msu.edu (S.S. Kulkarni), bolencha@cse.msu.edu (C. Bolen), oleszkie@cse.msu.edu (J. Oleszkiewicz), robin507@cse.msu.edu (A. Robinson).

URL: http://www.cse.msu.edu/~sandeep (S.S. Kulkarni).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2004.11.009

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 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 concur-