Self-Stabilizing Local Mutual Exclusion and Daemon Refinement *

Joffroy Beauquier† Ajoy K. Datta‡ Maria Gradinariu† Frederic Magniette†

† Laboratoire de Recherche en Informatique, Université de Paris Sud, France
‡ Department of Computer Science, University of Nevada Las Vegas

Abstract

Refining self-stabilizing algorithms which use tighter scheduling constraints (weaker daemon) into corresponding algorithms for weaker or no scheduling constraints (stronger daemon), while preserving the stabilization property, is useful and challenging. Designing transformation techniques for these refinements has been the subject of serious investigations in recent years. This paper proposes a new transformation technique for daemon refinement. The core of the transformer is a self-stabilizing local mutual exclusion algorithm. The local mutual exclusion problem is to grant a process the privilege to enter critical section if and only if none of its neighbors has the privilege. The contribution of this paper is twofold. First, we present a bounded-memory self-stabilizing local mutual exclusion algorithm for arbitrary networks, assuming any arbitrary daemon. After stabilization, this algorithm maintains a bound on the service time (the delay between two successive executions of critical section by a particular process). This bound is $\frac{n\times(n-1)}{2}$ where n is the network size. Another nice feature of our algorithm is that it satisfies the strong safety property — in any configuration, there is at least one privileged processor. Second, we use the local mutual exclusion algorithm to design two transformers which convert the algorithms working under a weaker daemon to ones which work under the distributed, arbitrary (or unfair) daemon. Both transformers preserve the self-stabilizing property. The first transformer refines algorithms written under the central daemon, while the second transformer refines algorithms designed for the k-fair ($k \geq (n-1)$) daemon.

Keywords: Local mutual exclusion, self-stabilization, transformer, unfair daemon.

1 Introduction

One of the most inclusive approaches to fault-tolerance in distributed systems is *self-stabilization* [Dij74, Dol00]. Introduced by Dijkstra [Dij74], this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior. The correctness of self-stabilizing algorithms is proven assuming some type of scheduler (or daemon) as the adversary. The two most common schedulers are the following: the central scheduler (only one process can execute an atomic step at one time) and the distributed scheduler (any nonempty subset of the enabled processes can execute their atomic steps simultaneously). Although it is easier to prove the stabilization for the algorithms working under the central scheduler, but those working under the distributed scheduler support more practical implementations. So, it makes sense to design self-stabilizing algorithms under central scheduler, prove its correctness in this model, and then transform the algorithms such that they work under the distributed scheduler preserving the self-stabilization

*A preliminary version of this work was presented at DISC'00 [BDGM00a].

and other desirable properties. One of the main goals of this paper is to design such a transformer. The dining philosophers problem [Dij71] deals with mutual exclusion among neighboring processes in a ring. The local mutual exclusion problem is the extension of this problem to any arbitrary network.

Related Work. There have been several attempts to develop transformers that transform a program written and proven under the assumption of a weak daemon to a program self-stabilizing under a strong daemon. A special class of systems, called alternator, was introduced in [GH97] for the linear topology and in [GH99] for any arbitrary topology. The idea of an alternator is the following: Every process has an integer state variable which is bounded by $2d-1$, where d is the length of the longest simple cycle in the network. One of the main features of the alternator is that no two enabled neighboring processes can have the state variable equal to $2d-1$ at the same time. The processes do some effective work only when they are in state $2d-1$. This non-interference property is used in [GH99] to design a transformer from the central daemon to the distributed daemon. The transformation idea is to compose the algorithm A (written under the central daemon) with the alternator such that the actions of the algorithm A are executed only when the alternator is in state $2d-1$. Another transformer was proposed in [MN98]. This method uses time-stamps to order the actions of any self-stabilizing algorithm. One notable feature of [MN98] is that it achieves the silent stabilization [DGS96], but the algorithm uses unbounded variables and works under a weakly fair scheduler.

Another approach in designing transformation techniques is to implement the local mutual exclusion among the neighboring processes. Distributed, but non-stabilizing solutions to the local mutual exclusion problem are presented in [CM84] and [AS90]. The alternator [GH99] can also be considered as a solution to the dining philosophers problem. However, this is not an optimal solution for the following reason: Only when one and exactly one process (say p) in a neighborhood is in state $2d-1$, p can enter the critical section. But, in this algorithm, there are many configurations where no process in some neighborhoods is in state $2d-1$, implying no process in those neighborhoods is in critical section. The algorithms in [HP89] and [Gou87] propose self-stabilizing solutions to the dining philosophers problem (and hence, to the local mutual exclusion problem). But, both solutions use a central daemon and a distinguished process to implement the token circulation. The process holding the token executes its critical section. A self-stabilizing solution to the dining philosophers problem has been proposed in [Hua00]. The algorithm in [Hua00] works under the read/write atomicity model [DIM93], but makes a very strong assumption—the links of the network are (initially) colored in a special way.

A solution to the local mutual exclusion problem for arbitrary networks is proposed in [AS99a]. This solution uses the read/write atomicity model. The proposed algorithm is not uniform. It uses an underlying spanning tree. This tree is used to to bound the timestamps—the system is periodically reset using the tree. The algorithm disables the critical section access during the reset period, and hence, is not an optimal solution. The legitimate state depends on some parameters, one of which has to be larger than the number of nodes. The authors claimed that the algorithm uses bounded memory, but did not provide any bound. The solution to the local mutual exclusion problem in tree networks is presented in [AS99b] and [JADT99].

A transformer using the local mutual exclusion has been reported in [NA99]. The main goal of this work was to refine the atomicity. The algorithm works under the read/write atomicity and uses bounded variables. However, since the algorithm uses a weakly fair daemon, although the service time is bounded, the exact bound on the service time cannot be computed (which depends on the type of daemon).

Our Contributions. We first present two solutions to the local mutual exclusion problem. The first solution uses unbounded memory. We then extend the first algorithm to design a bounded memory solution. Our algorithms are self-stabilizing under any arbitrary (unfair) distributed daemon. Moreover, after stabilization, the service time is bounded by $\frac{n(n-1)}{2}$.

We use the local mutual exclusion algorithm to design two stabilizing preserving transformers to transform algorithms written using weaker daemon into algorithms which work under the assumption of a stronger daemon. The first transformer refines algorithms written under the central daemon, while the second transformer takes as input algorithms designed for the k-fair ($k \geq (n-1)$) daemon. Both transformers convert the input algorithms to self-stabilizing algorithms which work under the assumption of arbitrary (even unfair) distributed daemon.

Outline of Paper. The rest of the paper is organized as follows: The model and specification of the problem solved in this paper are presented in Section 2. In Section 3, two local mutual exclusion algorithms are given. The two transformers are discussed in Section 4. Section 5 provides some concluding remarks.

2 Model and Specification

A distributed system is a set of state machines called processes. Each process can communicate with a subset of the processes called neighbors. We will use $\mathcal{N}.x$ to denote the set of neighbors of node x and $|\mathcal{N}.x|$ to represent the number of neighbors of x. The communication among neighboring processes is carried out using the communication registers (called "shared variables" throughout this paper). We consider distributed systems consisting of n processes where every process has a unique identifier. The system's communication graph is drawn by representing processes as nodes and the neighborhood relationship by edges connecting the nodes.

Every process p executes a program which consists of a set of shared variables and a finite set of guarded actions of the form: $\langle label \rangle :: \langle guard \rangle \longrightarrow \langle statement \rangle$, where each guard is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard evaluates to true.

A *configuration* of a distributed system is an instance of the state of the processes. A process is *enabled* in a given configuration if at least one of the guards of its program is *true*. We denote the set of enabled processes for a given configuration by E.

A distributed system can be modeled by a transition system. A transition system is a three-tuple $S = (\mathcal{C}, \mathcal{T}, \mathcal{I})$ where $\mathcal{C}$ is the collection of all the configurations, $\mathcal{I}$ is a subset of $\mathcal{C}$ called the set of initial configurations, and $\mathcal{T}$ is a function $\mathcal{T} : \mathcal{C} \longrightarrow \mathcal{C}$. A transition, also called a *computation step*, is a tuple (c_1, c_2) such that $c_2 = \mathcal{T}(c_1)$. A *computation* of an algorithm $\mathcal{P}$ is a *maximal* sequence of computation steps $e = ((c_0, c_1)\ (c_1, c_2)\ \ldots (c_i, c_{i+1}) \ldots)$ such that for $i \geq 0, c_{i+1} = \mathcal{T}(c_i)$ (a single *computation step*) if c_{i+1} exists, or c_i is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no process is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. A *fragment* of a computation e is a finite sequence of successive computation steps of e.

In a computation, a transition (c_i, c_{i+1}) occurs due to the execution of a nonempty subset of the enabled processes in configuration c_i. In every computation step, this subset is chosen by the scheduler or daemon. We refer to the following types of daemon in this paper: *central daemon* — in every computation step, only one of the enabled processes is chosen by the daemon; *k-fair daemon* — a process cannot be selected more than k times by the daemon without choosing another process

which has been continuously enabled; *weakly fair daemon* — if a process p is continuously enabled, p will be eventually chosen by the daemon to execute an action; *distributed unfair daemon* — during a computation step, any nonempty subset of the enabled processes is chosen by the daemon. We refer to the distributed unfair daemon as *stronger daemon* and all other daemons (defined above) as *weaker daemons.*

Self-Stabilization. In order to define self-stabilization for a distributed system, we use two types of predicates: the legitimacy predicate (defined on the system configurations and denoted by $\mathcal{L}$) and the problem specification (defined on the system computations and denoted by $\mathcal{SP}$).

Let $\mathcal{P}$ be an algorithm. The set of all computations of the algorithm $\mathcal{P}$ is denoted by $\mathcal{E}_{\mathcal{P}}$. Let $\mathcal{X}$ be a set and *Pred* a predicate defined on the elements of $\mathcal{X}$. The notation $x \models Pred$ means that the element $x \in \mathcal{X}$ satisfies the predicate *Pred.*

Definition 2.1 (Self-Stabilization) *An algorithm $\mathcal{P}$ is self-stabilizing for a specification $\mathcal{SP}$ if and only if the following two properties hold:*
(1) Convergence: All computations reach a configuration that satisfies the legitimacy predicate. Formally, $\forall e \in \mathcal{E}_{\mathcal{P}} :: e = ((c_0, c_1)(c_1, c_2)\dots) : \exists n \geq 1, c_n \models \mathcal{L}$.
(2) Correctness: All computations starting in a configuration satisfying the legitimacy predicate satisfy the problem specification $\mathcal{SP}$. Formally, $\forall e \in \mathcal{E}_{\mathcal{P}} :: e = ((c_0, c_1)\ (c_1, c_2)\ \dots) : c_0 \models \mathcal{L} \Rightarrow e \models \mathcal{SP}$.

Local Mutual Exclusion. The specification of the local mutual exclusion problem ($\mathcal{SP}_{LME}$) is the conjunction of the following two conditions defined in terms of "the privilege to enter the critical section": (i) *Safety:* If a process holds a privilege, then none of its neighbors holds the privilege. (ii)*Liveness:* Every process holds the privilege infinitely often.

Definition 2.2 (Strong Safety) *Let $\mathcal{P}$ be a self-stabilizing mutual exclusion algorithm. $\mathcal{P}$ satisfies the strong safety property if in any configuration there exists at least one privileged process.*

Definition 2.3 (Fairness Index) *Let $\mathcal{P}$ be a self-stabilizing mutual exclusion algorithm. $\mathcal{P}$ is considered to have a fairness index of k if in any computation of $\mathcal{P}$ under any daemon, between any two consecutive critical section executions of a process, any other process can execute its critical section at most k times.*

Definition 2.4 (Service Time) *Let $\mathcal{P}$ be a self-stabilizing local mutual exclusion algorithm. The service time of $\mathcal{P}$ is the maximum total number of critical sections executed by all other processors between two successive executions of critical section by any process without any assumption of the daemon.*

Virtual Orientation of the Communication Graph. We will use an "adjacency" relation, denoted by $\triangleright$, over the shared variables of the processes to define a virtual orientation of the communication graph. The exact definition (or implementation) of this relation will be specific to the two solutions to the local mutual exclusion problem presented in Section 3.

Definition 2.5 (Virtual Orientation) *Let $x.p$ and $x.q$ be two shared variables of two neighboring processes p and q. In the communication graph, the edge between p and q is said to be virtually oriented from p to q if and only if $x.p \triangleright x.q$. This edge is an incoming edge for q and an outgoing edge for p.*

Definition 2.6 (Privileged Process) *A process is said to be privileged in a configuration if in the communication graph, all the edges adjacent to the process are oriented toward it (i.e., all edges are incoming edges).*

3 Local Mutual Exclusion

We present two solutions to the local mutual exclusion problem in this section. We first present the unbounded space solution to help understand the ideas behind the second solution. Next, we will give the bounded solution which is our final solution. In the next section, we will use the bounded solution to design two scheduler transformers. Both solutions use the edge reversal mechanism of [BG89] to maintain the acyclic orientation of the communication graph.

3.1 Unbounded Local Mutual Exclusion Algorithm

The key feature of this algorithm is its $(n-1)$-fairness index under any arbitrary distributed daemon.

3.1.1 Algorithm ULME

Algorithm 3.1 Unbounded Local Mutual Exclusion (ULME) for process p

Constants:
$id.p$: unique integer identifier of p;
$\mathcal{N}.p$: the set of neighbors of p;
Shared Variable:
$L.p$: unbounded integer;
Local variable:
CS : boolean flag used to indicate if p is in the critical section;
Function:
$(\forall q \in \mathcal{N}.p) : p \prec q \equiv L.q \rhd L.p$

Actions:
$\mathcal{A} : \forall q \in \mathcal{N}.p, \quad p \prec q \longrightarrow$
$CS = 1;$
execute critical section;
$L.p = max\{L.q |\ q \in \mathcal{N}.p\} + 1;$

Note 3.1 *The variable CS in Algorithm ULME is not necessary to solve the local mutual exclusion problem. We added this in the code in Algorithm 3.1 because we would need this to design the transformers in Section 4.*

We borrow the definition of the "direction of an edge" from [GK93] to define the adjacency relation ($\rhd$) for Algorithm ULME (shown as Algorithm 3.1).

Definition 3.1 *For any two neighboring processes p_1 and p_2 executing Algorithm ULME, $L.p_2 \rhd L.p_1$ (i.e., p_2 is virtually oriented toward p_1) iff $(L.p_1 < L.p_2) \vee ((L.p_1 = L.p_2) \wedge (id.p_1 < id.p_2))$.*

Note 3.2 *Due to the uniqueness of the process ids, $\triangleright$ is a total order (anti-reflexive, anti-symmetric, and transitive) relation.*

The virtual orientation is used in Algorithm ULME in the following manner: A process enters its *critical section* if and only if it is privileged (Definition 2.6), i.e., all its incident edges are oriented toward it. Once the process finishes executing its critical section, it reverses all its incident edges. When p's value ($L.p$) is the local minimum, it becomes privileged. After using the privilege, p sets its value to one plus the maximum value in the neighborhood (Action $\mathcal{A}$).

3.1.2 Analysis of Algorithm ULME

In this section, we prove the correctness and convergence of Algorithm ULME. We also compute the bound for the service time and fairness.

Definition 3.2 (Legitimate Configuration) *A legitimate configuration of Algorithm ULME (i.e., a configuration which satisfies the legitimacy predicate $\mathcal{L}_{ULME}$) is a configuration that satisfies the following conditions: (i) At least one process is privileged. (ii) No two neighbors are privileged.*

Lemma 3.1 *Let G be the communication graph representing the system executing Algorithm ULME. G is acyclic.*

Proof: We prove this result by contradiction. Let $G_1 = (V_1, E_1)$ be the subgraph of G with $V_1 = \{p_1, \ldots, p_m\}$ such that the nodes in V_1 form a simple cycle. Then by Definition 3.1, the processes corresponding to V_1 satisfy the following: $L.p_1 \triangleright L.p_2 \triangleright \ldots \triangleright L.p_m \triangleright L.p_1 \Rightarrow L.p_1 \triangleright L.p_1$, which is impossible because $\triangleright$ is anti-reflexive (Note 3.2). □

Property 3.1 (Strong Safety) *During any computation of Algorithm ULME, at least one process is privileged in every configuration.*

Proof: Follows from Definitions 2.6 and 3.1, and Note 3.2, □

Lemma 3.2 *In any computation of Algorithm ULME, between every two successive executions of critical section by a process, all its neighbors execute their critical section exactly once.*

Proof: Let e be an arbitrary computation of Algorithm ULME and f_{12} a fragment of e from configuration c_1 to configuration c_2. Assume that in both c_1 and c_2, Process p is privileged, and these are the only two configurations in f_{12} where p is privileged. So, by Definition 2.6, in both c_1 and c_2, all edges adjacent to p are incoming edges. When p executes its critical section in c_1, all these edges are reversed, i.e., the adjacent edges become outgoing edges. Thus, before p becomes privileged again in c_2, all the edges must have been reversed. The edges are reversed only after a process is privileged and executes its critical section. Hence, all the neighbors of p must have been privileged at least once in f_{12}. In f_{12}, after a neighbor q of p executes its critical section and reverses the edge (i.e., makes it pointing toward p), the edge cannot be reversed again until p executes its critical section (in c_2). Thus, the neighbors of p execute their critical section exactly once in f_{12}. □

Definition 3.3 (Distance) *Let p and q be two processes. The distance between p and q, denoted by $dist(p, q)$, is the number of processes on the shortest path from p to q, plus 1. We denote by $SPdist_m.p$ and $|SPdist_m.p|$ the set of processes and number of processes, respectively, at distance i from p.*

Lemma 3.3 *In any computation of Algorithm ULME, between every two successive executions of critical section by a process p, q ($q \in SPdist_i(p)$) can execute its critical section at most i times.*

Proof: We prove this by induction on distance i.

- **Base Step:** Let us consider $i = 1$. All processes at distance 1 from p are neighbors of p. By Lemma 3.2, all neighbors of p execute their critical section exactly once.
- **Induction Step:** Assume that the lemma is true for processes at distance i from p. We will now show that it is also true for processes at distance $i + 1$ from p. $SPdist_{i+1}(p) = \bigcup_{q \in SPdist_i(p)} Neighbor(q)$. Consider a process $q \in SPdist_i(p)$. By the induction hypothesis, q can execute its critical section at most i times. During this time, any neighbor r of q (i.e., $r \in SPdist_{i+1}(p)$) can execute its critical section at most $i+1$ times — i times when q reversed the edges and once corresponding to the first execution of q.

□

Lemma 3.4 *Every computation of Algorithm ULME is infinite.*

Proof: Assume that there exists a finite computation, e. Let c be the last configuration in e. By Property 3.1, at least one process is privileged in c. So, p will execute its critical section (Action $\mathcal{A}_1$). Thus, we arrive at the contradiction. □

Lemma 3.5 (Liveness) *Every process is privileged infinitely often in every computation of Algorithm ULME.*

Proof: Let e be an arbitrary computation of Algorithm ULME. Assume that a process p is privileged only a finite number of times in e. Then by Lemma 3.2, every neighbor of p is also privileged a finite number of times. Since the communication graph is connected and acyclic (Lemma 3.1), all processes must be privileged a finite number of times. This implies that e is finite, which contradicts Lemma 3.4. □

Lemma 3.6 (Convergence) *Every computation of Algorithm ULME eventually reaches a configuration which satisfies the legitimacy predicate $\mathcal{L}_{ULME}$ (as defined in Definition 3.2).*

Proof: The proof follows from Definitions 2.6 and 3.2, and Property 3.1. □

Theorem 3.1 (Self-Stabilization) *Algorithm ULME is self-stabilizing.*

Proof: Correctness: The *safety* follows from Definition 3.1. The *liveness* is proven in Lemma 3.5.
Convergence: Proven in Lemma 3.6. □

Lemma 3.7 (Fairness Index) *The fairness index of Algorithm ULME is* $(n - 1)$.

Proof: Let e be an execution of Algorithm ULME under an unfair daemon. By Lemma 3.5, any process executes its critical section infinitely often in e. Let f be the fragment computation of e between two successive critical section executions of a process p. By Lemma 3.3, the processes at distance i from p execute their critical section at most i times in f. The maximal possible distance of a process from p is $n - 1$. Hence, in f, any process can execute its critical section at most $n - 1$ times. □

Lemma 3.8 (Service Time) *The delay between two successive executions of critical section by a process in Algorithm ULME is bounded by* $\frac{n(n-1)}{2}$.

Proof: The upper bound of the service time for a process p is given by the maximal value of the expression $\sum_{i=1}^{n-1} i.neigh_i$ where i is the distance to p and $neigh_i = |SPdist_i(p)|$. $\sum_{i=1}^{n-1} neigh_i = n-1$ where $neigh_{i+1} = 0$ if $neigh_i = 0$. The expression is maximal when $\forall i : neigh_i = 1$ for the following reason: Assume that for a particular value of i, $neigh_i$ is greater than 1. Then $neigh_{n-1}$ must be zero. Therefore, the value of the expression is reduced by $n-1$ and increased by $i < n-1$. Thus, the bound of the service time is $\frac{n(n-1)}{2}$. □

3.2 Bounded Memory Solution

The main drawback of Algorithm ULME is its unbounded memory requirement. In the following, we propose a bounded solution. The new algorithm is self-stabilizing and has a fairness index of $(n-1)$ under any arbitrary daemon.

3.2.1 Algorithm BLME

Algorithm BLME is a refined version of Algorithm ULME with bounded variables. This transformation deals with two major problems — the maintenance of the acyclic orientation of the communication graph and the choice of a bound for the variable L. Once the acyclic orientation is provided, Algorithm BLME (as shown in Algorithm 3.2) is quite similar to Algorithm ULME.

In the following, we redefine the adjacency relation $\rhd$ for the bounded integers. We then provide a bound for L.

Definition 3.4 (Cyclic Comparison) *Let x and y be two integers bounded by a positive integer $B \geq 2$. We define the relation $\rhd$ as follows:*

- $\forall x \in [0, \frac{B}{2}]$*:*
  1. $y \rhd x$ *iff* $y \in [x+1, x+\frac{B}{2}]$.
  2. $x \rhd y$ *iff* $y \in [\frac{B}{2}+x+1, B-1] \cup [0, x-1]$.
- $\forall x \in [\frac{B}{2}+1, B-1]$*:*
  1. $y \rhd x$ *iff* $y \in [x+1, B-1] \cup [0, x+\frac{B}{2}]$.
  2. $x \rhd y$ *iff* $y \in [x+\frac{B}{2}+1, x-1]$.

Figure 1 shows the (cyclic) relation between x, y_1, y_2, and y_3. In Figure 1(a), $y_1 \rhd x$, $x \rhd y_2$, and $x \rhd y_3$, and in Figure 1(b), $y_1 \rhd x$, $y_2 \rhd x$, and $x \rhd y_3$.

In the following, based on the relation defined in Definition 3.4, we redefine the virtual orientation of the communication graph using the variable L bounded by B. This orientation must provide an acyclic nature to the communication graph. We choose the value of B as follows: In order to ensure an acyclic orientation, the values of L of the neighboring processes must be different. In a completely connected graph, every node has $(n-1)$ neighbors. Therefore, the lower bound for the distance between the values of L of two neighboring processes is n. In order to avoid formation of a cycle among the nodes of the communication graph, the sum of n gaps between n processes which form the cycle must be less then B. The minimum value of B which satisfies the above condition is n^2

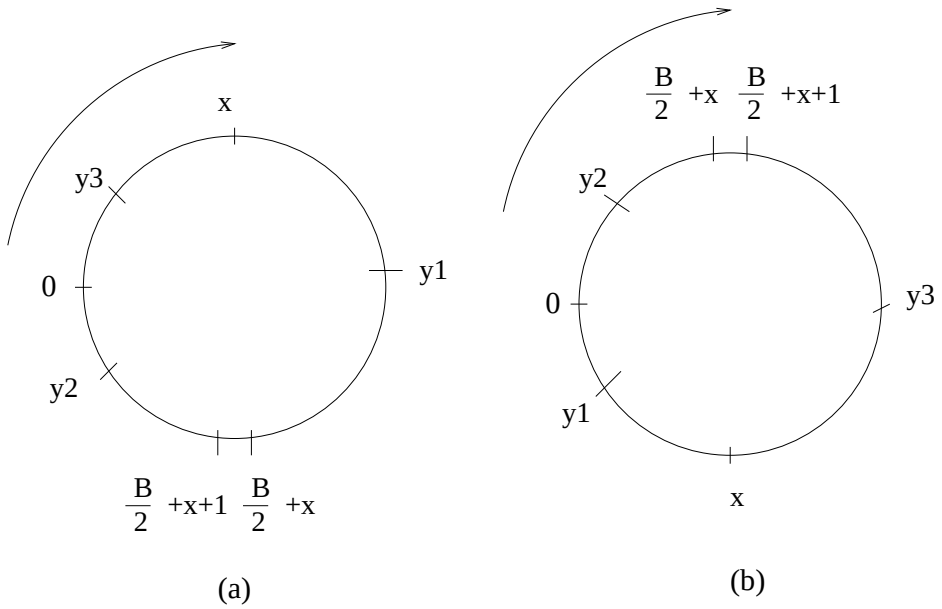

Figure 1: Cyclic Comparison.

when n is even and n^2+1 when n is odd. We give a more formal argument for the above explanation in Lemma 3.9. Note that the cyclic comparison creates an ordering among the values when all the values are in an interval bounded by n.

Definition 3.5 (Bounded Virtual Orientation) *Let p_1 and p_2 be two neighboring processes executing Algorithm BLME. p_1 is virtually oriented towards p_2 ($p_1 \rhd p_2$) if they satisfy the properties specified in Definitions 2.5 and 3.4 defined over the variable L for $B = n^2$ if n is even, and $B = n^2+1$ if n is odd.*

Note that the variable L of each process executing Algorithm BLME has values in $[0..B-1]$ where B is the constant defined in Definition 3.5. As in Algorithm ULME, Algorithm BLME also maintains an acyclic communication graph in all legitimate configurations. To achieve this characteristic of the underlying graph, we force the processes to satisfy the following property (Definition 3.6). (We will show in Section 3.2.2 how the balanced processes guarantee that the communication graph is acyclic.)

Definition 3.6 (Balanced Process) *Two neighboring processes, p_1 and p_2, running Algorithm BLME are said to be balanced with respect to each other (denoted as $p_1 \sim p_2$) if and only if $|L.p_1 - L.p_2| < n \wedge ((L.p_1 \neq L.p_2) \vee (L.p_1 = L.p_2 = 0))$. When all processes are balanced with respect to all their neighbors, we refer to this situation as a balanced configuration. Let p_1 and p_2 be two neighboring processes which are not balanced. We call these processes unbalanced and denote this condition as $p_1 \not\sim p_2$. If at least two processes are unbalanced in a configuration, we say that the system is in an unbalanced configuration.*

A process p is enabled to execute its critical section if it is balanced (Definition 3.6) with its neighbors and if all the edges adjacent to p are oriented towards p (Action $\mathcal{A}_1$). After p exits its critical section, p reverses its incident edges. This allows the neighbors of p to get a chance to execute their critical section. In a self-stabilizing setting, the system may start in an unbalanced

Algorithm 3.2 Bounded Local Mutual Exclusion (BLME) for process p

Constants:

B : n^2 if n is odd, n^2+1 if n is even.

$\mathcal{N}.p$: the set of neighbors of p;

Shared Variables:

$L.p \in [0..B-1]$;

$R.p$: boolean; reset flag

Local Variables:

CS: boolean flag used to indicate if p is in critical section;

Functions:

$MaxL(p) \equiv L.i$ such that $|L.i - L.p| \mod B = max(|L.j - L.p| \mod B, \forall j \in \mathcal{N}.p)$;

$(\forall i \in \mathcal{N}.p) : p \prec i \equiv (L.i \rhd L.p) \vee ((L.p = L.i = 0) \wedge (id.p < id.i))$;

$(\forall i \in \mathcal{N}.p) : p \sim i \equiv ((|L.p - L.i| \mod B) < n) \wedge ((L.p \neq L.i) \vee (L.p = L.i = 0))$;

Actions:

$\mathcal{A}_1$: $(R.p = 0) \wedge (\forall i \in \mathcal{N}.p,\ p \prec i \wedge p \sim i \wedge R.i = 0) \longrightarrow$
CS=1;
execute critical section;
$L.p = (MaxL(p) + 1) \mod B$;

$\mathcal{R}_1$: $(R.p = 0) \wedge (\exists i \in \mathcal{N}.p,\ (p \not\prec i \vee R.i = 1) \wedge (L.i \neq 0 \vee L.p \neq 0)) \longrightarrow$
CS=0;
$R.p = 1$;

$\mathcal{R}_2$: $(R.p = 1) \wedge (L.p \neq 0) \wedge (\forall i \in \mathcal{N}.p,\ R.i = 1) \longrightarrow$
CS=0;
$L.p = 0$;

$\mathcal{R}_3$: $(R.p = 1) \wedge (L.p = 0) \wedge (\forall i \in \mathcal{N}.p,\ L.i = 0) \longrightarrow$
CS=0;
$R.p = 0$;

configuration (Definition 3.6). Starting from this unbalanced configuration, Algorithm BLME will eventually take the system into a balanced configuration, and the communication graph becomes acyclic again. From this configuration onwards, Algorithm BLME always moves from one balanced configuration to another, and the communication graph will remain acyclic.

When a process p recognizes that (i) it is unbalanced with respect to at least one of its neighbors, or (ii) it has a neighbor i such that $R_i = 1$ (which implies that some processes are unbalanced), p executes Action $\mathcal{R}_1$ and sets its reset marker R_p to 1. p then waits until all its neighbors also reset their R to 1. At that time, p changes its L variable to 0 (Action $\mathcal{R}_2$). Again, p waits until all its neighbors reset their L to 0. When that happens, one of the processes in the neighborhood of p becomes balanced with all its neighbors. Let us assume that p is that process. p then clears R to 0 so that it can eventually execute its critical section (Action $\mathcal{R}_3$).

A process leaves the reset period (the period in which it resets its L variable to 0) after the execution of the action $\mathcal{R}_3$. The duration of this period depends on the reset actions executed by the neighboring processes.

3.2.2 Analysis of Algorithm BLME

In this section, we prove the correctness and convergence of Algorithm BLME. We also compute the bound for the service time and fairness. To prove the correctness of Algorithm BLME, we first show that starting from a legitimate configuration, any computation always maintains the legitimacy—the processes execute only Action $\mathcal{A}_1$. Finally, we prove the convergence by defining different possible scenarios in terms of edges of G and using an "edge migration" scheme.

Lemma 3.9 *Let G be the communication graph representing the system running Algorithm BLME. If the system is balanced, then G is acyclic.*

Proof: We prove this result by contradiction. Let $G_1 = (V_1, E_1)$ be the subgraph of G with $V_1 = \{p_1, \ldots, p_m\}$ such that the nodes in V_1 form a simple cycle. Let us consider the case where the values of L are not equal to 0. (For $L = 0$, the acyclic orientation is provided by the uniqueness of the identifiers.) Then by Definition 3.1, the processes corresponding to V_1 satisfy the following:

$$L.p_1 \rhd L.p_2 \rhd ... \rhd L.p_m \rhd L.p_1 \tag{1}$$

Since every process $p_i,\ i \in [1..m]$ is balanced with all its neighbors, by Definition 3.6,

$$|L.p_i - L.p_{i+1}| < n \tag{2}$$

which implies that

$$|L.p_1 - L.p_2| + |L.p_2 - L.p_3| + ... + |L.p_{n-1} - L.p_m| + |L.p_m - L.p_1| < B \tag{3}$$

Equation (1) implies that

$$|L.p_1 - L.p_2| + |L.p_2 - L.p_3| + ... + |L.p_{m-1} - L.p_m| + |L.p_m - L.p_1| = k * B \tag{4}$$

where k is a (nonzero) positive integer.

Equations (3) and (4) contradict each other. Hence, we arrived at the contradiction. □

Property 3.2 *If the system is balanced, then there exists at least one privileged process and no two neighbors are privileged.*

Definition 3.7 (Legitimate Configuration) *A legitimate configuration of Algorithm BLME (i.e., a configuration which satisfies the legitimacy predicate $\mathcal{L}_{BLME}$) is a configuration such that the following two conditions hold: (i) The system is in a balanced configuration. (ii) The value of the reset marker R of all processes is* 0.

Remark 3.1 *In any legitimate configuration of Algorithm BLME, the processes execute only Action $\mathcal{A}_1$.*

Lemma 3.10 *Let e be a computation of Algorithm BLME starting from a legitimate configuration c (i.e., c satisfies $\mathcal{L}_{BLME}$). Then any configuration reachable from c in e also satisfies $\mathcal{L}_{BLME}$.*

Proof: Consider a computation step (c, c') in e. Assume that p is one of the processes which took part in the step (c, c'). Then by Note 3.1, p executed $\mathcal{A}_1$ in this step. By Definition 3.6,

$$\forall i \in \mathcal{N}.p,\ |L.p - L.i| < n \tag{5}$$

We assumed that p executed $\mathcal{A}_1$ in c. This is possible if and only if all its incident edges are oriented toward it, i.e.,

$$\forall i \in \mathcal{N}.p,\ p \prec i \equiv \forall i \in \mathcal{N}.p,\ L.i \rhd L.p \tag{6}$$

Let i_{max} and i_{min} be the neighbors of p representing the process holding the maximal and minimal value of L, respectively, in c. Let $L.p_{old}$ and $L.p_{new}$ be the value of p in c and c', respectively. From the algorithm, $L.p_{new} = L.i_{max} + 1$. From Equation (6),

$$L.i_{min} \rhd L.p_{old} \tag{7}$$

From Equation (7), $L.i_{min}$ is at least $L.p_{old} + 1$. Using Equation (5), we get

$$|L.p_{old} - L.i_{max}| < n \tag{8}$$

Using Equations (7) and (8), we obtain the final result:

$$|L.p_{new} - L.i_{min}| \leq |L.i_{max} + 1 - (L.p_{old} + 1)| \leq |L.i_{max} - L.p_{old}| < n \tag{9}$$

□

Now, we want to prove the convergence of Algorithm BLME. We first need to define some covering edge sets of the communication graph G representing the system running Algorithm BLME.

- $M_1 = \{(p, q) \mid p$ and q are balanced and $(R.p = 0 \wedge R.q = 0)\}$.
- $M_2 = \{(p, q) \mid p$ and q are balanced, and $(R.p = 1 \vee R.q = 1)\}$.
- $M_3 = \{(p, q) \mid p$ and q are unbalanced, and $(R.p = 0 \vee R.q = 0)\}$.
- $M_4 = \{(p, q) \mid p$ and q are unbalanced and $(R.p = 1 \wedge R.q = 1)\}$.
- $MT = \bigcup_{i=1}^{4} M_i$.
- $\forall i \in \{2, 3, 4\} : M_i^0 = \{(p, q) \mid (p, q) \in M_i \wedge (L.p = 0 \vee L.q = 0)\}$.
- $M^{(0,0)} = \{(p, q) \mid L.p = 0 \wedge L.q = 0\}$.

In a legitimate configuration, all edges are in M_1, i.e., $MT = M_1$, and the only action enabled at any process is $\mathcal{A}_1$ (Note 3.1). When the system is in an illegitimate configuration, $MT \neq M_1$, and from the algorithm, at least one of Actions $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ is enabled at some process. So, our obligation now is to show that starting from such a configuration where $MT \neq M_1$, eventually, all edges will become part of M_1 again. We explain this process by using an "edge migration" process.

We will first show that starting from an illegitimate configuration, eventually, one of the processes would be able to execute one of Actions $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$, meaning that the process of convergence would eventually start. We then prove the convergence process using the edge migration process as follows (shown in Figure 2):

1. Every edge of M_i $(i \neq 1, i \neq 3)$ eventually becomes a member of M_i^0 or $M^{(0,0)}$.
2. Every edge of M_3 eventually becomes a member of M_4.
3. Every edge of $M_i^0 (i \neq 1)$ eventually moves to $M^{(0,0)}$.
4. Every edge of $M^{(0,0)}$ eventually moves to set M_{1r} which is defined as follows: M_{1r} contains all edges (p, q) such that p and q are balanced, $R.p = 0$, $R.q = 0$, the edge (p, q) was originally in a set $M_i (i \neq 1)$, and $(L.p \neq 0 \vee L.q \neq 0)$.

Note that M_{1r} is actually the same as M_1 except that the set M_{1r} is created only due to a computation starting from an illegitimate state. So, after the system is back in a legitimate configuration, M_{1r} becomes the current M_1.

Lemma 3.11 *Let e be a computation of Algorithm BLME starting from an illegitimate configuration c, i.e., where $MT \neq M_1$. Then eventually one of the actions in $\{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3\}$ will be executed in e.*

Proof: From the algorithm, in c, there must be at least one process which is enabled to execute one of the actions in $\{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3\}$. Assume that P is the set of processes which are enabled in c to execute one of the above actions. Obviously, then all processes in P are not enabled to execute $\mathcal{A}_1$. So, we need to prove that one of the processes in P will eventually be able to execute its action. We prove this by contradiction. Assume that in every configuration in e starting from c, the processes in P execute only Action $\mathcal{A}_1$, i.e., every configuration is balanced. So, the communication graph contains a cycle, which is impossible by Lemma 3.9. Therefore, there exists a configuration where no process in P can execute Action $\mathcal{A}_1$. Hence, one of the processes in P would then be able to execute one of the actions in $\{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3\}$. □

Lemma 3.12 (Convergence) *Let e be a computation of Algorithm BLME starting from a configuration c where $MT \neq M_1$. Then eventually, all edges of G will belong to M_{1r}.*

Proof: Let (p, q) be an edge which is not in M_{1r} in c. Assume that p or q executes an action in c. We will now consider the following possibilities:

(1) $(p, q) \in M_1$.

(1a) p is balanced with all its neighbors. Then p can only execute $\mathcal{A}_1$ and (p, q) would still remain in M_1.

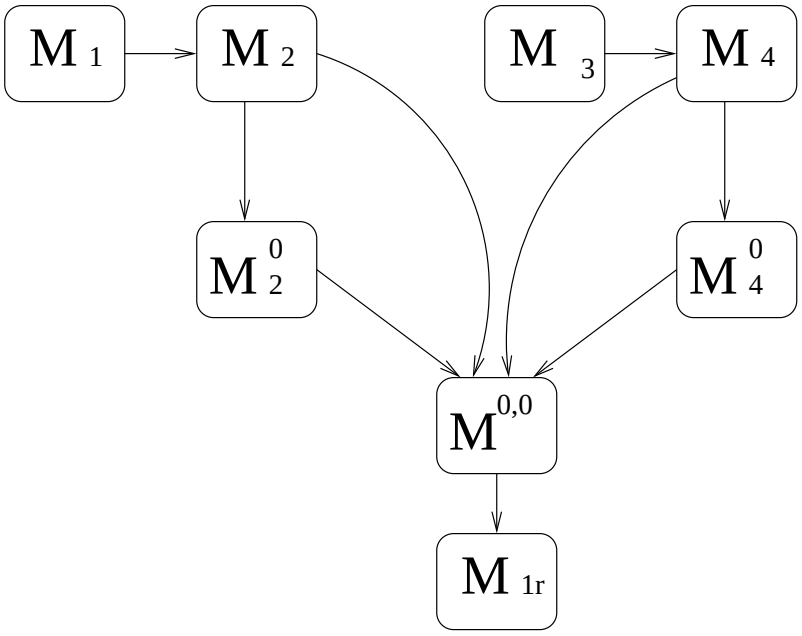

Figure 2: **Edge Migration**

(1b) p has a neighbor r such that p and r are unbalanced, or $R.r = 1$. The only possible action for p to execute is $\mathcal{R}_1$, and after its execution (which is guaranteed by Lemma 3.11), (p, q) will move to the set M_2.

Note that eventually, both p and q would satisfy Case (1b). So, we can claim that all edges (p, q) in M_1 eventually move to M_2.

(2) $(p, q) \in M_2$. There are two situations to consider:

(2a) $R.p = 1$ and $R.q = 0$. q executes $\mathcal{R}_1$. In this case, (p, q) becomes an edge of the second type (Case (2b)).

(2b) $R.p = 1$ and $R.q = 1$. p executes Action $\mathcal{R}_2$ or $\mathcal{R}_3$. After the execution of $\mathcal{R}_2$ (again guaranteed by Lemma 3.11), (p, q) migrates to M_2^0. If $\mathcal{R}_3$ is executed by p instead, (p, q) switches to $M^{(0,0)}$.

(3) $(p, q) \in M_3$.

(3a) $R.p = 0$ and $R.q = 0$. If only p or q executes its action, then the edge becomes an edge of the second type (Case (3b)). If both p and q execute their action, then (p, q) moves to M_4.

(3b) $R.p = 1$ and $R.q = 0$. q executes $\mathcal{R}_1$. (p, q) moves from M_3 to M_4.

(4) $(p, q) \in M_4$. After executing $\mathcal{R}_2$ or $\mathcal{R}_3$, the edge migrates to M_4^0 or $M^{(0,0)}$.

(5) $(p, q) \in M_2^0$ or M_4^0. After executing $\mathcal{R}_2$, (p, q) moves to $M^{(0,0)}$.

(6) $(p, q) \in M^{(0,0)}$. p and q can execute only $\mathcal{A}_1$ in this situation. When any of them executes $\mathcal{A}_1$, (p, q) switches to M_{1r}.

It is obvious from the algorithm that the set M_{1r} is a closed set. □

Remark 3.2 *The liveness (Lemma 3.5), $(n-1)$-fairness (Lemma 3.7), and service time (Lemma 3.8 proven for Algorithm ULME also hold for Algorithm BLME because the orientation of edges is changed in an identical manner in the two algorithms.*

Theorem 3.2 (Self-Stabilization) *Algorithm BLME is self-stabilizing.*

Proof: The convergence is proven in Lemmas 3.12.
The correctness follows from Property 3.2, Lemma 3.5, and Note 3.2. □

4 Daemon Refinement

In this section, we propose an application of the local mutual exclusion algorithms introduced in the previous section. We refer to the algorithms presented in the previous section as $\mathcal{A}_{LME}$. We use those algorithms to transform self-stabilizing algorithms proven under some weaker daemon (e.g., the central or k-fair daemon) called weaker algorithms to self-stabilizing algorithms which would work in the presence of the stronger daemon (i.e., the distributed unfair daemon). From now on, we refer to these algorithms as the stronger algorithms.

Transformation. The transformation technique is based on a particular composition scheme between the actions of the local mutual exclusion algorithm ($\mathcal{A}_{LME}$) and a weaker algorithm $\mathcal{W}$. Assume that $\mathcal{A}_{LME}$ and $\mathcal{W}$ have m and n actions, respectively. Let $a_{lme}g_i$ (respectively, wg_i) and $a_{lme}s_i$ (respectively, ws_i) represent the guard and statement, respectively, of ith action of $\mathcal{A}_{LME}$ (respectively, $\mathcal{W}$). The composed algorithm $\mathcal{S}$ consists of the following actions:

- $\forall i \in [1..m]$: $< a_{lme}g_i > \longrightarrow < a_{lme}s_i >$;
  if $CS = 1$ then
  if $\exists j \in [1..n] < wg_j >$ then $< ws_j >$;

Note 4.1 *The actions of the weaker algorithm are executed only when $CS = 1$, i.e., when the process is allowed to execute its critical section. The variable CS is used only to design the transformers (see Note 3.1).*

We now present some properties of the composed algorithm.

Lemma 4.1 *Any maximal computation of the composed algorithm has a maximal projection on Algorithm $\mathcal{A}_{LME}$.*

Proof: Let e be a maximal computation of the composed algorithm. Suppose that the projection of e on the $\mathcal{A}_{LME}$ algorithm is not maximal. This means that e has a suffix where no $\mathcal{A}_{LME}$ action is executed, only the weaker actions are executed. From the above definition of composition, the weaker actions are executed only in conjunction with the actions of $\mathcal{A}_{LME}$, which implies that our assumption is false. □

Lemma 4.2 *The composed algorithm is a self-stabilizing local mutual exclusion algorithm according to the specification $\mathcal{SP}_{LME}$.*

Proof: Let e be a maximal computation of the composed algorithm. Consider the maximal projection, e_{st} of e on the algorithm $\mathcal{A}_{LME}$ (the existence of the maximal projection is proven in Lemma 4.1). By assumption, a legitimate configuration for this algorithm is reached during e_{st}, and the suffix of e_{st} (let us call it e_{sts}) starting from this legitimate configuration satisfies the local mutual exclusion specification. Let e_s be the suffix of e corresponding to the projection e_{sts} on $\mathcal{A}_{LME}$. The computation e_s then also starts in a legitimate configuration for this algorithm, and satisfies local mutual exclusion specification. □

Lemma 4.3 *The fairness index of the composed algorithm is* $(n-1)$.

Proof: Let e be a computation of the composed algorithm. Assume that in e, the fairness index is not $(n-1)$. That implies that the fairness index of the projection of e on algorithm $\mathcal{A}_{LME}$ is not $n-1$, which contradicts Lemma 3.7 (where we established that the fairness index of $\mathcal{A}_{LME}$ is $(n-1)$). □

Lemma 4.4 *The projection of any maximal computation of the composed algorithm on the weaker algorithm is a maximal computation.*

Proof: Let e be a maximal computation of the composed algorithm and e_w be the projection of e on the weaker algorithm. Assume that e_w is not a maximal computation. Hence there exists a suffix of e in which the actions of the weaker algorithm are not executed, only the actions of the $\mathcal{A}_{LME}$ algorithm are executed. Let c be the first configuration occurring in this suffix. So, in c and any other configuration reachable from c in e, one of the two following conditions is true: (i) No weaker guard of any process is enabled. (ii) There exists a process p enabled to execute a weaker action, but p is never chosen by the daemon to execute. In Case (i), the prefix of e ending at c has a maximal projection on the weaker algorithm. In Case (ii), using the $(n-1)$-fairness property of the composed algorithm (see Lemma 4.3), p will eventually execute this weaker action in conjunction with $\mathcal{A}_{LME}$ critical section execution. Thus, the projection on the weaker algorithm is maximal. □

Lemma 4.5 *The projection of any maximal computation of the composed algorithm on the weaker algorithm has a maximal suffix where the neighboring processes do not execute their actions simultaneously.*

Proof: Let e be a maximal computation of the composed algorithm. From Lemma 4.2, e has a maximal suffix, f satisfying the local mutual exclusion specification. Let e_W be the projection of e on the weaker protocol. From Lemma 4.4, the computation e_W is maximal. Let f_W be the projection of f on the weaker protocol. Now, we need to show that in the computation f_W, the neighboring processes do not execute their actions simultaneously. Suppose that there exists a configuration in f_W where two neighboring processes execute their actions simultaneously. This means that there is a configuration in f such that the critical section is executed simultaneously by two neighboring processes which contradict the assumption that f satisfies the local mutual exclusion property. □

4.1 Central Daemon to Distributed Daemon

In this section, we show that we can transform a self-stabilizing algorithm working under a central daemon into a self-stabilizing algorithm under a distributed daemon. Assume that, in the composition described in Section 4, the weaker algorithm, $\mathcal{W}_C$, is a self-stabilizing algorithm for the specification $\mathcal{SP}_C$ which works under a central daemon, and $\mathcal{S}$ represents the composed algorithm.

Lemma 4.6 *In any algorithm $\mathcal{P}$, if in a computation the neighboring processes do not execute their actions simultaneously, then there must exist a corresponding computation of $\mathcal{P}$ under a central daemon.*

Proof: Let e be a computation of $\mathcal{P}$ under a distributed daemon such that the neighboring processes do not execute their actions simultaneously. Computation e_c (under central daemon) corresponding to e is obtained by constructing a fragment computation of e_c for every transition in e. Let c and c' be two successive configurations in e. Assume that in c, processes $p_1, \ldots, p_k$ execute their actions to change the configuration to c'. These processes are not neighbors of each other as per our assumption. Consider the following fragment $f = (c = c_0, c_1, c_2, \ldots c_k = c')$ where c_i is obtained from c_{i-1} after process p_i executes its action. Since processes p_i $(i \in \{1, k\})$ are not neighboring processes, the execution of p_i's action has no influence on the execution of other active processes, and hence, the fragment f satisfies the central daemon specification. Thus, we can consider the fragment f as the computation under the central daemon corresponding to the transition (c, c') in e under the distributed daemon. We can repeat the above construction process to obtain the computation e_c corresponding to e (working under the distributed daemon). □

Theorem 4.1 *Algorithm $\mathcal{S}$ is self-stabilizing for Specification $\mathcal{SP}_\mathcal{C}$ under a distributed daemon.*

Proof: From Lemma 4.2, the composed algorithm is a self-stabilizing local mutual exclusion algorithm under a distributed daemon. So, any computation of the composed algorithm has an infinite suffix which satisfies the local mutual exclusion specification. Let e be one such computation and e_W be the projection of e on the weaker algorithm. From Lemmas 4.4 and 4.5, the projection e_W is maximal and also satisfies the property that the neighboring processes do not execute their actions simultaneously. From Lemma 4.6, there is a computation e_W^c, such that e_W^c contains all the configurations which appeared in e_W and satisfies the central daemon specification. Hence there exists a function which maps any computation of the composed algorithm to a computation of the weaker algorithm under a central daemon. Thus, since the weaker algorithm is self-stabilizing for Specification $\mathcal{SP}_\mathcal{C}$ under the central daemon, the composed algorithm is self-stabilizing for Specification $\mathcal{SP}_\mathcal{C}$. □

4.2 Fair Daemon to Distributed Daemon

We propose another application of the local mutual exclusion in this section — the transformation of a self-stabilizing algorithm under a k-fair daemon into a self-stabilizing algorithm under a distributed daemon.

Assume that in the composition described in Section 4, the weaker algorithm, $\mathcal{W}_{kB}$ is a self-stabilizing algorithm for the specification $\mathcal{SP}_\mathcal{F}$ under a k-fair daemon $(\forall k \geq (n-1))$, and $\mathcal{S}$ is the composed algorithm.

Theorem 4.2 *Algorithm $\mathcal{S}$ is self-stabilizing for Specification $\mathcal{SP}_\mathcal{F}$ under any distributed daemon.*

Proof: Let e be a computation of Algorithm $\mathcal{S}$. From Lemma 4.3, we can claim that e is a $(n-1)$-fair computation. Let e_W be the projection of e on the weaker algorithm. Now, we need to prove that e_W is a computation under a k-fair $(\forall k \geq (n-1))$ daemon. Assume that e_W is not a computation under the k-fair daemon. This implies that a process p executes its actions more then k times before another process q executes. This also means that p executes its critical section more than k times before q executes in e. But, that is not possible because the fairness index of e is $(n-1)$ and $k \geq (n-1)$. □

5 Conclusions

We presented a transformation technique to transform self-stabilizing algorithms under weak daemons into algorithms which maintain the self-stabilization property and also paper under the stronger daemon, e.g., any arbitrary distributed daemon (including the unfair daemon). The key tool in designing the above is a self-stabilizing local mutual exclusion algorithm, which by itself is a major contribution of this work. One of the two local mutual exclusion algorithms presented in this paper is a bounded memory self-stabilizing solution which is proven under the unfair daemon. Another nice feature of our local mutual algorithms is that they achieve a bounded service time.

Our protocols can be easily transformed to work under the read/write atomicity model (see [BDGM00b] for this work), and hence can be extended to the message-passing environment. Another possible extension for our bounded solution is a generalization of the unison problem defined in [CFG92]. In Algorithm BLME, the difference between the values of the local variables of any two neighboring processes is bounded. Therefore, each process can start a phase with the number equal to the value of its local variable L. The main features of this extension are that the phase difference between two processes is bounded and no two neighboring processes are in the same phase.

6 Acknowledgments

We would like to thank the referees for their valuable which greatly improve the presentation of the paper. Special thanks to the referee who helped us simplify the composition scheme in Section 4.

References

[AS90] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. *31st Annual Symposium on Foundations of Computer Science*, volume I:65–74, october 1990.

[AS99a] GH Antonoiu and PK Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In *Euro-par99, Parallel Processing, Proceedings LNCS:1685*, pages 823–830, 1999.

[AS99b] GH Antonoiu and PK Srimani. Self-stabilizing protocol for mutual exclusion among neighboring nodes in tree structured distributed system. *Parallel algorithms and applications 58*, pages 215–221, 1999.

[BDGM00a] J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *DISC'00 Proceedings of the Thirteenth International Symposium on Distributed Computing*, pages 223–227, 2000.

[BDGM00b] J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. Technical Report 1251, LRI, Universite de Paris Sud, France, 2000.

[BG89] V. Barbosa and E. Gafni. Concurrency in heavily loaded neighborhood-constrained systems. *ACM Transactions on Programming Languages and Systems Vol 11 Num 4*, pages 562–584, 1989.

[CFG92] J.M. Couvreur, N. Francez, and M. Gouda. Asynchronous unison. *ICDCS92 Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 486–493, 1992.

[CM84] M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transations on Programming Languages and Systems*, pages 6(4):632–646, october 1984.

[DGS96] S. Dolev, M. Gouda, and M. Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, pages 115–138, 1971.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *ACM 17*, pages 643–644, 1974.

[DIM93] S. Dolev, A. Israeli, and S. Moran. Self-stabilizing of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[Dol00] S. Dolev. *Self-Stabilization.* The MIT Press, 2000.

[GH97] M. Gouda and F. Hadix. The linear alternator. *Proceedings of the third workshop on self-stabilizing systems (WSS-97), International Informatics Series 7, Carleton University Press*, pages 31–47, 1997.

[GH99] M. Gouda and F. Hadix. The alternator. *In Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53, 1999.

[GK93] S. Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, pages 7:55–59, 1993.

[Gou87] M. G. Gouda. The stabilizing philosopher: asymmetry by memory and by action. *Tech Rep TR-87-12, Univesity of Texas at Austin*, 1987.

[HP89] D. Hoover and J. Poole. A distributed self-stabilizing solution for the dining philosophers problem. *Information Processing Letter 41*, pages 209–213, 1989.

[Hua00] S.T. Huang. The fuzzy philoshophers. In *Workshop on Advances of Paralleland Distributed Computational Models*, page to appear, May 2000.

[JADT99] C. Johnen, L. O. Alima, A. K. Datta, and S. Tixeuil. Self-stabilizing neighborhood synchronizer in tree networks. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems (ICDCS'99)*, pages 487–494, june 1999.

[MN98] M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letter 66*, pages 285–290, 1998.

[NA99] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *DISC'99 Proceedings of the Thirteenth International Symposium on Distributed Computing*, pages 254–268, 1999.