Transformations of Mutual Exclusion Algorithms from the Cache-coherent Model to the Distributed Shared Memory Model

Hyonho Lee
Department of Computer Science
University of Toronto

Abstract

We present two transformations that convert a class of local-spin mutual exclusion algorithms on the cache-coherent model to local-spin mutual exclusion algorithms on the distributed shared memory model without increasing their time complexity. Our first transformation uses registers and test-and-set objects, and does not increase the number of busy-waiting periods. The second transformation uses only registers, but contains two busy-waiting periods for each busy-waiting period of the input algorithm. We carefully define the class of mutual exclusion algorithms that are applicable to our transformations, and formally prove the correctness of our transformations.

1 Introduction

For most mutual exclusion algorithms, busy-waiting is inevitable since a process must wait in the entry section while another process is in the critical section. In shared memory systems, a waiting process keeps accessing shared memory during a busy-waiting period. Such busy-waiting results in an unbounded number of memory accesses, which may cause unbounded network traffic. In shared memory systems with processes having their own local memory, algorithms can adopt a *local-spinning* strategy: all busy waiting periods consist of only read events of local variables. In such algorithms, a busy-waiting period does not generate unbounded amount of network traffic. In shared memory systems with local-spinning, the number of *remote memory accesses (RMAs)* performed is a major contributor to the time taken by algorithms, and it is used as a measure of time complexity.

There are two major shared memory models for distributed systems with local memory: the *distributed shared memory* (DSM) model and the *cache-coherent* (CC) model. In the DSM model, each process has its own local memory, and each shared variable is physically located in the local memory of one fixed process. Processes are connected with each other via a network, and each time a process accesses a shared variable that is local to another process, it must traverse the network.

In the CC model, all shared variables are stored in a common memory that is accessible to all processes, but not local to any process. In addition, each process has its own local cache. When a process accesses a shared variable for the first time, it copies the shared variable to its local cache, so it generates network traffic. If the process accesses the shared variable again without changing its value, more network traffic is not generated, unless the shared variable was updated by another process in the meantime. When a process updates a shared variable, it invalidates all cached copies of the shared variable except its own. The number of remote memory accesses in the CC model is the number of accesses of a shared variable that change the value of the variable or is by a process that does not have a valid cached copy of the shared variable.

In the CC model, the system must keep all caches consistent, so the CC model is considered to be more expensive than the DSM model. However, the CC model allows many processes to locally read the same variable at the same time. Therefore, it is usually easier to design a local-spin algorithm on the CC model than on the DSM model. Also, if an algorithm is local-spin on the DSM model, it is also local-spin on the CC model.

A number of different N-process local-spin mutual exclusion algorithms using different objects have been proposed. Yang and Anderson presented a tree-based local spin mutual exclusion algorithm on the DSM model using only registers, in which each process performs $O(\log N)$ remote memory accesses to enter and leave a critical section [1]. A lower bound proof by Anderson and Kim shows that any local-spin mutual exclusion algorithm using only registers or comparison-based objects such as $compare\&swap$ requires $\Omega(\log N/\log\log N)$ RMAs on the CC or DSM model [2]. This impossibility result suggests that any mutual exclusion algorithm with constant time complexity must use objects other than registers and comparison-based objects.

There are several local-spin mutual exclusion algorithms on the CC model using *fetch*&$store$ or *fetch*&inc that perform a constant number of RMAs for each entry to the critical section[3, 4]. These algorithms are based on the *queue-lock* strategy: when a process enters the entry section, it is enqueued at the end of the queue, and when the critical section is available, the process at the head of the queue enters the critical section. Thus, such an algorithm also satisfies the *first-come-first-serve* (FCFS) property.

Anderson and Kim presented a constant RMA mutual exclusion algorithm using *fetch*&ϕ on the CC model and converted it to the DSM model [5]. As part of their transformation (Transformation AK), they use any instance of a two-process mutual exclusion algorithm.

They claim that Transformation AK can be applied to a class of mutual exclusion algorithms designed for the CC model. However, we found that if Transformation AK [5] using the two-process mutual exclusion by Yang and Anderson [1] is applied to the constant RMA algorithm for the CC model by T. Anderson [3], then the resulting algorithm is incorrect. The problem is a race between the busy-waiting process and the unlocking process: Suppose that process p is waiting for q to unlock p using the spin variable S. Although S is local to p on the CC model, it may not be local to p on the DSM model. If S is not local to p on the DSM model, then, in the transformed algorithm, p must spin on a local spin variable S' instead of S. Process p must communicate with process q, so that q knows which spin variable to access to unlock p. In the resulting algorithm, processes p and q communicate with one another using a two-process mutual exclusion algorithm. In the two-process mutual exclusion algorithm by Yang and Anderson, it is possible that process p does not busy-wait on the spin variable, but q accesses the spin variable. This unlocking event by q may interfere with a later invocation of p that waits for another process r to unlock p using the same spin variable, allowing the resulting algorithm to violate mutual exclusion.

In this paper, we present two new transformations that do not have this problem. Both convert a certain class of local-spin mutual exclusion algorithm on the CC model to local-spin mutual exclusion algorithms on the DSM model, without increasing their time complexity.

Our first transformation (Transformation HL1) is based on the modified version of T.Anderson's algorithm [3] that appeared in [6]. The only new objects introduced by Transformation HL1 are registers and *test*&*set* objects. Many mutual exclusion algorithms with constant time complexity use some *fetch*&ϕ object in addition to registers. Since any *fetch*&ϕ object can be used to implement a *test*&*set* object, the transformed versions of these algorithms can be implemented without using additional types of objects. Another advantage of Transformation HL1 is that, unlike Transformation AK, any DSM algorithm resulting from Transformation HL1 does not have more busy-waiting periods than the algorithm from which it originates.

Our second transformation (Transformation HL2) uses only registers. Thus, if the original algorithm uses only registers, then the transformed algorithm also uses only registers. Transformation HL2 is the first such transformation from the CC model to the DSM model.

An interesting question is whether there is a difference between the CC model and the DSM model in terms of time complexity. Recently [7], Danek and Hadzilacos showed that the N-process two-session group mutual exclusion can be solved with $O(\log N)$ RMAs on the CC model, but, on the DSM model, an optimal group mutual exclusion algorithm requires $\Omega(N)$ RMAs. This result indicates that the CC model is indeed more powerful than the DSM model for this problem.

However, Kim and Anderson proved in [8] that, with semi-synchrony (i.e. there is a bound on the time to execute an event), the ordinary mutual exclusion problem can be solved on the DSM model with a constant number of RMAs for each entry to the critical section, whereas the CC model requires $\Omega(\log\log N)$ RMAs for each entry to the critical section. Thus, we can see that the CC model is less powerful than the DSM model in some circumstances. It is not clear whether the CC model is more powerful than the DSM model for solving the mutual exclusion problem in an asynchronous environment.

We begin, in Section 2, by defining the models. In Section 3, we carefully define the class of algorithms to which Transformations HL1, and HL2 apply. We also present Transformations HL1 and HL2, and prove that the mutual exclusion algorithm produced by Transformation HL1 when applied to an algorithm in this class is correct and does not perform more RMAs than the original algorithm.

2 Preliminaries

We only consider asynchronous shared memory systems with no process failures. Let $\mathcal{P}$ be a set of all processes and $N = |\mathcal{P}|$. There are *private* and *shared* variables in the system. Private variables are accessed only by their owner, but shared variables can be accessed by many processes. In this paper, all private variables start with a lower case letter, and all shared variables start with an upper case letter. Process p's private variable var is denoted by var_p. On the DSM model, each shared variable is local to only one process. We use $Var[p]$ to denote a shared variable that is local to process p in the DSM model.

When a process accesses a shared variable, it performs a certain operation on the variable. This is called an *event*. Note that, in this paper, accessing a private variable is not considered as an event. In this paper, only registers and $test\&set$ objects are considered. Registers have $read$ and

$write$ operations, and $test\&set$ objects have $test\&set$ and $reset$ operations. All operations are *atomic*. The $test\&set$ operation returns the value of the object and sets the value of the object to one. The $reset$ operation sets the value of the object to zero. *Fetch*$\&\phi$ operations, also called *read-modify-write* operations, are defined as follows.

```
fetch&$\phi$ ($Var, value$) {
        $t := Var$
        $Var := \phi(t, value)$
        **return** $t$ }
```

Here, ϕ is a function whose range excludes the initial value, NIL. The function ϕ does not necessarily depend on one or both of its parameters. For example, for the *fetch&inc* operation, $\phi(t, value) = t + 1$. For *fetch&store*, $\phi(t, value) = value$, and for *test&set*, $\phi(t, value) = 1$. *Test&set* is the simplest *fetch*$\&\phi$ operation and can be simulated by any *fetch*$\&\phi$ object as follows:

```
test&set ($Var$) {
        $t :=$ fetch&$\phi(Var, 1)$
        **if** $t =$ NIL **then return** $0$ **else return** $1$ }
```

A *configuration* is a state of the system. It consists of the state of all processes and shared variables. The state of a process is the value of its private variables. A process's *program counter* is a private variable that indicates which line of its algorithm the process performs next. The value of variable V in configuration C is denoted by $val(V, C)$, and the program counter of process p in configuration C is denoted by $pc(p, C)$.

Each line of our transformations contain exactly one event. Thus, we can specify an event by a process and its program counter. For example, $(p, \mathbf{1})$ indicates the event in which process p performs line **1** of the algorithm. Since a process can invoke a mutual exclusion algorithm several times, event $(p, \mathbf{1})$ may occur several times. Therefore, we need to distinguish different invocations by the same process. In this paper, p^i indicates the ith invocation by a process p.

An *execution segment* is an alternating sequence of configurations and events starting and ending with a configuration. An execution segment is *admissible* if, for each contiguous subsequence $C, (p, \mathbf{a}), C'$ that it contains, $pc(p, C)$ = **a** and C' is the configuration obtained by performing $(p, \mathbf{a})$ in configuration C. An admissible execution segment starting with an initial configuration is called an *admissible execution*.

3 Transformations from the CC model to the DSM model

Each of our transformations consists of two code fragments. The transformation replaces any busy-waiting period of the input algorithm with the first fragment, and replaces any unlocking event of the input algorithm with the second fragment. These new fragments introduce a new set of variables, $\mathcal{U}$. In this paper, $\mathcal{V}$ represents the set of all private and shared variables in the input algorithm excluding the program counters.

In the input algorithm, when a process waits for another process to unlock its busy-waiting period, we call the former process the *successor* and the latter process the *predecessor*. The predecessor and the successor communicate with each other using a shared spin variable. The set of all spin variables is $\mathcal{S} = \{S_i \mid i \in I\}$. When a process enters the entry section, it chooses the index $i \in I$ of the spin variable it will use.

3.1 Requirement of the input algorithm

Our transformations, as well as Transformation AK, can be applied to a certain class of mutual exclusion algorithms. We describe this class of input algorithms by four requirements. Let p^x, q^y, r^z be different invocations, where p, q, and r may be the same or different processes. Let **X** denote the set of all lines in the input algorithm that contain busy-waiting periods, and let **Y** denote the set of all lines in the input algorithm that perform unlocking events. Let **X+1** be the set of all lines in the input algorithm that follow a busy-waiting period.

Requirement 1. *All spin variables are registers.*

Requirement 2. *Each busy-waiting period in the algorithm is a period of time during which a process performs "**await** $S = a$", where $S \in \mathcal{S}$ and a is a non-NIL value. (Usually, the initial value of S is NIL. However, one spin variable may have an initial value a, so that the first invocation of "**await** $S = a$" exits the busy-waiting period without waiting.)*

Requirement 3. *When two processes are in busy-waiting periods at the same time, they use different spin variables during those busy-waiting periods. In particular, if process p performs "**await** $S = a$" while process q performs "**await** $S' = a'$", then $S \neq S'$. This requirement is more formally described as follows: For any admissible execution of the input algorithm, there does not exist a configuration C such that $pc(p, C)$, $pc(q, C) \in$ **X** and $val(i_p, C) = val(i_q, C)$, where p and q are different processes and i_p, $i_q \in I$.*

Requirement 4. *For each busy-waiting period, there is a unique unlocking event that makes the busy-waiting period terminate. More formally, consider an admissible execution $\alpha = C_0, E_1, C_1, \ldots$ of the input algorithm. Suppose that $pc(p, C_j) \in$ **X** for $j = k, \ldots, k'$, and $pc(p, C_{k'+1}) \in$ **X+1**. Let $u = val(i_p, C_k)$. If l is the largest index less than k such that $pc(q, C_l) \in$ **X+1** and $val(i_q, C_l) = u$ for some process*

*q, then we require that there is a unique event $E_d = (r, y)$, where $y \in$ **Y**, $value(i_r, C_{d-1}) = u$, and $l < d < k'$.*

*Now, suppose that there is no such index l. If $val(S_u, C_0) = NIL$, then we require that there is a unique event E_d = (r, **y**) such that **y** $\in$ **Y**, $val(i_r, C_{d-1}) = u$ and $0 < d < k'$; otherwise, there must be no such event E_d.*

Although these requirements look difficult to check, most queue-based CC model mutual exclusion algorithms [3, 4] satisfy the requirements. Requirement 2 can be slightly generalized by replacing the condition $S = a$ by the disjunction of a finite number of such conditions. This is called **Requirement 2′.** Transformation AK can be applied to some algorithms that satisfy Requirement 2′ instead of Requirement 2, e.g. the *fetch*&ϕ mutual exclusion algorithm by Anderson and Kim [5]. If we modify Transformation HL1 and HL2 slightly, then they can also be applied to the same class of algorithms.

A process only reads a spin variable in its busy-waiting period. Until that process writes to some shared variable, no other process can detect whether it has finished spinning. Therefore, we have the following observation.

Observation 1. *Let C, E, C' be an admissible execution segment of the input algorithm, and let E = (p^x, **x**), where **x** $\in$ **X**. Then, for all processes $q \neq p$, $C \approx_q C'$.*

The next result follows from this observation and Requirement 3.

Lemma 2. *For any admissible execution of the input algorithm, there does not exist a configuration C such that $pc(p, C)$, $pc(q, C) \in$ **X** $\cup$ **X+1** and $val(i_p, C) = val(i_q, C)$, where p and q are different processes and i_p, $i_q \in I$.*

Requirement 4 and the definition of admissible execution imply the following lemma.

Lemma 3. *Let $\alpha = C_0, E_1, C_1, \ldots$ be an admissible execution of the input algorithm. Suppose that there exist a configuration C_j and a process p such that $val(i_p, C_j) = u$ and $pc(p, C_j) \in$ **X**. If there exists a process q and a configuration $\check{C}$ before C_j such that $pc(q, \check{C}) \in$ **X+1** and $val(i_q, \check{C}) = u$, then let $\hat{C}$ be the last configuration among such $\check{C}$; otherwise let $\hat{C} = C_0$.*

*Then, there does not exist more than one invocation r^x for which there exists a configuration C between $\hat{C}$ and C_j in α such that $pc(r, C) \in$ **Y** and $val(i_r, C) = u$. Moreover, if the initial value of S_u is a, then there do not exist an invocation r^x and a configuration C between C_0 and C_j in α such that $pc(r, C) \in$ **Y** and $val(i_r, C) = u$.*

Proof. Let r^x and s^y be two invocations. Suppose, to obtain a contradiction, that there exist configurations C and C' between $\hat{C}$ and C_j in α such that $val(i_r, C) = val(i_s, C') = u$, $pc(r, C)$ = **y** $\in$ **Y**, and $pc(s, C')$ = **y′** $\in$ **Y**.

Let α' be the prefix of α that ends with C_j. Let α'' be the extension of α' after (r^x, **y**) and (s^y, **y′**) are added to the end of α' if they are not already in α'. Then, α'' is also an admissible execution.

Since the input algorithm satisfies lockout freedom, there exists a configuration C_k in the extension of α'' such that $val(i_p, C_k) = u$ and $pc(p, C_k) \in$ **X+1**. Then, there exist two events (r^x, **y**) and (s^y, **y′**) between $\hat{C}$ and C_k. However, this contradicts Requirement 4.

Similarly, if the initial value of S_u is a and there exists such r^x, then there is an admissible execution of the input algorithm that violates Requirement 4. □

Lemma 4. *Let $\alpha = C_0, E_1, C_1, \ldots$ be an admissible execution of the input algorithm. Then, there do not exist processes p and q and a configuration C_j such that $val(i_p, C_j) = val(i_q, C_j) = u$, $pc(p, C_j)$ = **x** $\in$ **X+1**, and $pc(q, C_j)$ = **y** $\in$ **Y**.*

Proof. Suppose that such p, q, and C_j exist. Let α' be the prefix of α ending with C_j, and let $\hat{C}$ be the last configuration before C_j in α' such that, for some process r, $val(i_r, \hat{C}) = u$ and $pc(r, \hat{C}) \in$ **X+1**. If there is no such configuration, let $\hat{C} = C_0$. Then, by Requirement 4, there exist a unique event E_d = (s^z, **y**) such that **y** $\in$ **Y**, $val(i_s, C_{d-1}) = u$, and C_d is between $\hat{C}$ and C_j.

Let α'' be obtained by adding (q, **y**) and the resulting configuration C_k to the end of α'. Then α'' is admissible. Then, $val(i_p, C_k) = u$ and $pc(p, C_k) \in$ **X+1**. But, in α'', there are two events (s^z, **y**) and (q, **y**) between $\hat{C}$ and C_k. This violates Requirement 4. □

3.2 Transformation HL1

In Transformation HL1, the set of the introduced variables, $\mathcal{U}$, is $\{$ $waiter_p$, $Signal[i]$, $Want[i]$, $Lock[p, i]$ $\mid$ $p \in \mathcal{P}$ and $i \in I$ $\}$. Transformation HL1 replaces line **x** of the input algorithm with lines **x.a** to **x.f** and line **y** of the input algorithm with lines **y.g** to **y.j**. Figure 1 describes Transformation HL1 in detail.

Transformation HL1 can be described informally as follows. Let p be a process. In Transformation HL1, p uses its local shared variable $Lock[p, i]$ as a spin variable instead of S_i, the spin variable of the input algorithm. The predecessor of p must know the identity of p so that it can access $Lock[p, i]$. In Transformation HL1, the successor, p, writes its identity to $Want[i]$, and p's predecessor reads $Want[i]$ to get the identity of its successor.

However, if the predecessor of p reads $Want[i]$ before p writes its identity, the predecessor does not get the identity of p by reading $Want[i]$. In this case, the predecessor does not access $Lock[p, i]$ and p should not spin on $Lock[p, i]$.

```
Shared variables:
    $Want[i] \in \{\text{NIL}, 0, \ldots, N-1\}$ initially NIL
    $Signal[i] \in \{0, 1\}$ if $S_i$ is initially $a$, then initially 1.
                          otherwise, initially 0.
    $Lock[p, i] \in \{\text{LOCKED, UNLOCKED}\}$
                initially LOCKED

instead of performing:
    x:      await $S_i = a$
process $p$ performs:
    x.a:    $Want[i] := p$
    x.b:    if $test\&set(Signal[i]) = 0$ then
    x.c:        await $Lock[p, i]$ = UNLOCKED
                    /* local spin */ fi
    x.d:    $Signal[i] := 0$ /* reset */
    x.e:    $Lock[p, i]$ := LOCKED
    x.f:    $Want[i]$ := NIL

instead of performing:
    y:      $S_i := a$
process $q$ performs:
    y.g:    $S_i := a$
    y.h:    if $test\&set(Signal[i]) = 1$ then
    y.i:        $waiter := Want[i]$
    y.j:        $Lock[waiter, i]$ := UNLOCKED fi
```

Figure 1. Transformation HL1

The $test\&set$ object $Signal[i]$ is used for p and its predecessor to know whether p must busy-wait.

The initial value of $Signal[i]$ depends on the initial value of S_i, the spin variable of the input algorithm. If S_i is initially set to a, the value that makes the first busy-waiting period unlocked, then the initial value of $Signal[i]$ is 1. Hence, the first process that busy-waits on S_i in the input algorithm does not perform the busy-waiting period in the transformed algorithm, since the busy-waiting period of the input algorithm is unlocked initially.

Otherwise, the initial value of $Signal[i]$ is 0. Thus, among process p and the predecessor of p, whoever first performs $test\&set$ operation on $Signal[i]$ receives 0, and the other receives 1. Process p writes its identity to $Want[i]$ before performing the $test\&set$ operation, and the predecessor of p reads $Want[i]$ after performing the $test\&set$ operation.

If the predecessor of p receives 0 and p receives 1 from $Signal[i]$, then the predecessor does not access $Lock[p, i]$ and p does not spin on $Lock[p, i]$. If the predecessor of p receives 1 and p receives 0 from $Signal[i]$, then the predecessor can get the identity of p by reading $Want[i]$. Hence, the predecessor can access $Lock[p, i]$ to unlock p.

Proof of correctness of Transformation HL1

We will prove the correctness of Transformation HL by a simulation proof. Let $\mathcal{B}$ be the set of all admissible executions of the transformed algorithm.

First, we define a simulation function from a configuration C of the transformed algorithm to a configuration C' of the input algorithm such that process p is in the critical section in C if and only if p is in the critical section in C'. Then, using the simulation function we defined, for each $\beta \in \mathcal{B}$, we find an admissible execution α of the input algorithm that simulates β in the sense that it has the same trace as β. Given that the input algorithm satisfies mutual exclusion and lockout freedom, we can prove that every admissible execution of the transformed algorithm also satisfies mutual exclusion and lockout freedom.

Definition 1. *For each configuration C of an admissible execution of the transformed algorithm, let $f(C)$ be the configuration of the input algorithm defined as follows:*

P1. *For any process p such that $pc(p, C) \notin \{$**x.a**$, \ldots,$ **x.f**, **y.g**$, \ldots,$ **y.j**$\}$, $pc(p, f(C)) = pc(p, C)$.*

P2. *If $pc(p, C) \in \{$**x.a**, **x.b**, **x.c**$\}$, then $pc(p, f(C))$ = **x**.*

P3. *If $pc(p, C) \in \{$**x.d**, **x.e**, **x.f**$\}$, then $pc(p, f(C))$ = **x+1**.*

P4. *If $pc(p, C) \in \{$**y.g**, **y.h**, **y.i**, **y.j**$\}$, then $pc(p, f(C))$ = **y**.*

P5. *For any variable v except spin variables used in the input algorithm, the value of v is the same in both C and $f(C)$. That is, for any $v \in \mathcal{V} - \mathcal{S}$, $val(v, f(C)) = val(v, C)$.*

P6. *For any spin variable $S_u \in \mathcal{S}$, if there does not exist a process p such that $pc(p, C) \in \{$**y.h**, **y.i**, **y.j**$\}$ and $val(i_p, C) = u$, then $val(S_u, f(C)) = val(S_u, C)$. Otherwise, let p be the process that last executed line **y.g** among those such that $pc(p, C) \in \{$**y.h**, **y.i**, **y.j**$\}$ and $val(i_p, C) = u$. Then, $val(S_u, f(C)) = val(S_u, C')$ where C' is the configuration just before p last executed line **y.g**.*

For each $\beta \in \mathcal{B}$, let $C_0, C_1, \ldots$ be the sequence of configurations in β and let $\mathcal{F}(\beta)$ be the set of all admissible executions α of the input algorithm such that the sequence $f(C_0), f(C_1), f(C_2), \ldots$ with consecutive duplicates removed is a subsequence of α. We say that event e is an *interfering event* at configuration C, if $e = (p^x,$ **x.a**$)$ and there is another process q such that $val(i_q, C) = val(i_p, C)$ and $pc(q, C) \in \{$**x.a**$, \ldots,$ **x.f**$\}$, or if $e = (p^x,$ **y.g**$)$ and there exists another process q such that $val(i_q, C) = val(i_p, C)$ and $pc(q, C) \in \{$**y.g**$, \ldots,$ **y.j**$\}$.

First, we consider only $\beta \in \mathcal{B}$ without any interfering events. Let $\mathcal{B}' \subset \mathcal{B}$ such that $\mathcal{B}' = \{\beta \mid \beta$ does not have an interfering event$\}$. We will show that, for each $\beta \in \mathcal{B}'$, $\mathcal{F}(\beta)$ is not empty. Then, we will prove that there is no admissible execution of the transformed algorithm in $\mathcal{B} - \mathcal{B}'$. Thus, we will show that $\mathcal{B}' = \mathcal{B}$.

Observation 5. *Let $\beta \in \mathcal{B}'$. Let C be a configuration in β and let u be an index. Then, there exists at most one process p such that $pc(p, C) \in \{$**x.a**$, \ldots,$**x.f**$\}$ and $val(i_p, C) = u$. Similarly, there exists at most one process p such that $pc(p, C) \in \{$**y.g**$, \ldots,$**y.j**$\}$ and $val(i_p, C) = u$.*

From this observation, we have the following lemma.

Lemma 6. *Let $\beta \in \mathcal{B}'$, and let C and C' be any consecutive configurations of β such that, for all processes p, either $pc(p, f(C)) = pc(p, f(C'))$ or $pc(p, C) \notin \{$**y.h**$,$**y.i**$,$**y.j**$\}$, where **y** $\in$ **Y**. Then, for any variable $v \in \mathcal{V}$, if $val(v, C) = val(v, C')$, then $val(v, f(C)) = val(v, f(C'))$.*

Proof. Let $v \in \mathcal{V}$ be such that $val(v, C) = val(v, C')$. If $v \in \mathcal{V} - \mathcal{S}$, then, by P5 of Definition 1, $val(v, f(C)) = val(v, C)$ and $val(v, f(C')) = val(v, C')$, so $val(v, f(C)) = val(v, f(C'))$.

Consider $v = S_u \in \mathcal{S}$. In this case, we look at the program counters of all processes p such that $val(i_p, C) = u$. If, for all such processes p, neither $pc(p, C)$ nor $pc(p, C')$ is in $\{$**y.h**, **y.i**, **y.j**$\}$, then $val(S_u, f(C)) = val(S_u, C)$ and $val(S_u, f(C')) = val(S_u, C')$ by P6 of Definition 1. Thus, $val(S_u, f(C)) = val(S_u, f(C'))$. Now, suppose that there exists a process p such that $val(i_p, C) = u$ and at least one of $pc(p, C)$ and $pc(p, C')$ is in $\{$**y.h**$,$**y.i**$,$**y.j**$\}$. Then, by Observation 5, there exists exactly one such p. Let C'' be the configuration just before p last executed line **y.g**.

First, suppose that $pc(p, C) \in \{$**y.h**, **y.i**, **y.j**$\}$. Then, $pc(p, f(C')) = pc(p, f(C)) =$ **y**. Since C' is the configuration follwing C, $pc(p, C') \in \{$**y.h**, **y.i**, **y.j**$\}$. Then, by P6 of Definition 1, $val(S_u, f(C)) = val(S_u, C'') = val(S_u, f(C'))$. Otherwise, only $pc(p, C')$ is in $\{$**y.h**, **y.i**, **y.j**$\}$. Then, $pc(p, C) =$ **y.g** and $pc(p, C') =$ **y.h**. Note that $C = C''$ since C is the configuration just before p performs line **y.g**. Thus, by P6 of Definition 1, $val(S_u, f(C')) = val(S_u, C) = val(S_u, f(C))$. □

Now we show that, for an admissible execution $\beta = C_0, E_1, C_1, E_2, C_2, \ldots$ of the transformed algorithm, there exists an admissible execution α of the input algorithm such that $\alpha \in \mathcal{F}(\beta)$. We do this inductively using the following two lemmas.

Lemma 7. *If C_0 is an initial configuration of the transformed algorithm, then $f(C_0)$ is an initial configuration of the input algorithm.*

Proof. Let C_0 be an initial configuration of the transformed algorithm. Consider any process p. If the transformation does not apply to the first line, then $pc(p, C_0) =$ **1**. If the transformation does apply to the first line, then $pc(p, C_0) =$ **1.a** or **1.g**. In either case, $pc(p, f(C_0)) =$ **1** by P2 and P4 of Definition 1.

Also, no variable in $\mathcal{V}$ is changed by f. Since the initial values of variables in $\mathcal{V}$ are the same in both the transformed algorithm and the input algorithm, and for each process p, $pc(p, f(C_0)) =$ **1**, $f(C_0)$ is an initial configuration of the input algorithm. □

Now we prove the induction step.

Lemma 8. *Let $\beta = C_0, E_1, C_1, \ldots, C_{j-1}, E_j, C_j \in \mathcal{B}'$, and let β' be the prefix of β ending with C_{j-1}. Then, if $\mathcal{F}(\beta')$ is not empty, then $\mathcal{F}(\beta)$ is also not empty.*

Proof. Let $\alpha \in \mathcal{F}(\beta')$ that ends with $f(C_{j-1})$. Suppose that E_j is performed by invocation p^x and $u = val(i_p, C_j)$. There are four cases depending on the operation performed by p^x.

Case 1: E_j is (p^x, **x.a**), (p^x, **x.b**) with $test\&set$ $(Signal[u])$ = 0, (p^x, **x.c**) with $Lock[p, u]$ = LOCKED, (p^x, **x.d**), (p^x, **x.e**), (p^x, **x.f**), (p^x, **y.g**), (p^x, **y.h**) with $test\&set(Signal[u])$ = 1, or (p^x, **y.i**). In these cases, $pc(p, f(C_{j-1})) = pc(p, f(C_j))$ by Definition 1. Thus, for all processes $p \in \mathcal{P}$, $pc(p, f(C_{j-1})) = pc(p, f(C_j))$. Among these events, only (p^x, **y.g**) changes the value of a shared variable in $\mathcal{V}$. In the other cases, for all $v \in \mathcal{V}$, $val(v, C_{j-1}) = val(v, C_j)$, and hence, by Lemma 6, $val(v, f(C_{j-1})) = val(v, f(C_j))$.

If E_j = (p^x, **y.g**), then for all $v \in \mathcal{V} - \{S_u\}$, $value(v,\ C_{j-1}) = value(v,\ C_j)$, so by Lemma 6, $val(v, f(C_{j-1})) = val(v, f(C_j))$. The value of S_u is set to a by (p^x, **y.g**), so $val(S_u, C_{j-1})$ may not be equal to $val(S_u, C_j)$. But, $pc(p, C_j) =$ **y.h**, so by P6 of Definition 1, $val(S_u, f(C_j)) = val(S_u, C_{j-1}) = val(S_u, f(C_{j-1}))$. Therefore, in both cases, $f(C_j) = f(C_{j-1})$, so $\alpha \in \mathcal{F}(\beta)$.

Case 2: E_j = (p^x, **x.b**) with $test\&set(Signal[u])$ = 1. In this case, we show that $(f(C_{j-1}), (p^x,$ **x**$), f(C_j))$ is an admissible execution segment of the input algorithm. Then, $\alpha, (p^x,$ **x**$), f(C_j)$ is an admissible execution of the input algorithm that is in $\mathcal{F}(\beta)$. By P2 of Definition 1, $pc(p, f(C_{j-1})) =$ **x**. Let D be the configuration obtained by performing event $(p^x,$ **x**$)$ in configuration $f(C_{j-1})$. We will show that $D = f(C_j)$ by proving that, for all variables $v \in \mathcal{V}$, $val(v, D) = val(v, f(C_j))$ and for all processes $p \in \mathcal{P}$, $pc(p, D) = pc(p, f(C_j))$.

Since p^x receives 1 from $Signal[u]$ in E_j, p^x skips line **x.c**. Note that, for all processes $q \neq p$, $pc(q, C_{j-1}) = pc(q, C_j)$ since p is the only process that takes a step from C_{j-1} to C_j. Also, neither $pc(p, C_{j-1})$ nor $pc(p, C_j)$ is in $\{$**y.h**$,$**y.i**$,$**y.j**$\}$ since $pc(p, C_{j-1}) =$ **x.b** and $pc(p, C_j) =$ **x.d**. Since E_j changes only $Signal[u]$, which is not in $\mathcal{V}$, $val(v, C_j) = val(v, C_{j-1})$ for all $v \in \mathcal{V}$. Therefore, by Lemma 6, $val(v, f(C_j)) = val(v, f(C_{j-1}))$ for all $v \in \mathcal{V}$. Since event $(p^x,$ **x**$)$ does not change any variable, $val(v, D) = val(v, f(C_{j-1}))$ for all $v \in \mathcal{V}$. Hence, for all $v \in \mathcal{V}, val(v, D) = val(v, f(C_j))$.

For all processes $q \neq p$, $pc(q, f(C_{j-1})) = pc(q, f(C_j))$ since $pc(q, C_{j-1}) = pc(q, C_j)$. Since only p^x takes a step in $(p^x, \mathbf{x})$, for all processes $q \neq p$, $pc(q, D) = pc(q, f(C_{j-1}))$. Thus, for all processes $q \neq p$, $pc(q, D) = pc(q, f(C_j))$.

Now, we will show that $pc(p, D) = pc(p, f(C_j))$. By Definition 1, $pc(p, f(C_j))$ = **x+1**. In order for $pc(p, D)$ to be **x+1**, p^x must read $S_u = a$ in event $(p^x, \mathbf{x})$. Hence, the value of S_u at $f(C_{j-1}))$ should be a. We consider the following cases to show $val(S_u, f(C_{j-1})) = a$.

Case 2-1: Invocation p^x is the first invocation that performed **x.a** with a particular value of u. Then no invocation performs **x.a** with the same value of u until p^x performed line **x.f** by Observation 5. Thus, no invocation except p^x performs line **x.b** or **x.d** with the same value of u before C_{j-1}.

Case 2-1-1: There does not exist an invocation that performed line **y.h** with the same value of u before C_{j-1}. Then $(p^x, \mathbf{x.b})$ is the first event that accesses $Signal[u]$, since $Signal[u]$ is accessible in only lines **x.b**, **x.d**, and **y.h**. Thus, p^x got the initial value of $Signal[u]$ in event $(p^x, \mathbf{x.b})$. Since p^x received 1 from $Signal[u]$, the initial value of $Signal[u]$ is 1. But, this holds only if the initial value of S_u is a.

If no process performed **y.g** with the same value of u prior to C_{j-1}, then the value of S_u remains unchanged, so $val(S_u, C_{j-1}) = a$. In C_{j-1}, there is no process with the same value of u whose program counter is in $\{\mathbf{y.h}, \mathbf{y.i}, \mathbf{y.j}\}$. Thus, by Definition 1, $val(S_u, f(C_{j-1})) = val(S_u, C_{j-1}) = a$. If there exists a process q with the same value of u that performed **y.g** before C_{j-1}, then the value of S_u may be changed by q. Since q did not perform line **y.h** before C_{j-1}, $pc(q, C_{j-1})$ = **y.h**. By Observation 5, no proces other than q has its program counter in $\{\mathbf{y.h}, \mathbf{y.i}, \mathbf{y.j}\}$. Then, by P6 of Definition 1, the value of S_u in $f(C_{j-1})$ is the value of S_u just before q performed **y.g**.

By the Lemma 3, no process with the same value of u performed line **y.g** before q did. Thus, the value of S_u in $f(C_{j-1}))$ is 1, which is the initial value of S_u. Therefore, $val(S_u, f(C_{j-1})) = a$.

Case 2-1-2: There exists an invocation that performs line **y.h** with the same value of u before C_{j-1}. Let q^y be such an invocation, and let C_k be the configuration just before q^y performs line **y.h**. Then, $pc(q, f(C_k))$ = **y** and $pc(p, f(C_{j-1}))$ = **x**. Thus, by Lemma 3, there is no other invocation with the same value of u whose program counter is **y** between $f(C_k)$ and $f(C_{j-1})$. Hence, no process with the same value of u performed **y.h** between C_k and C_{j-1}.

Since q^y is the first invocation that accesses $Signal[u]$, it gets the initial value from $Signal[u]$ in line **y.h**. Note that, by Requirement 4 of the input algorithm, if the initial value of S_u was a, then there is no process with the same value of u whose program counter is **y** before $f(C_{j-1})$. This contradicts our assumption that q^y exists. Hence, the initial value of S_u is not a. Therefore, the initial value of $Signal[u]$ is 0. So, q^y receives 0 from $Signal[u]$, skips lines **y.i** and **y.j**, and finishes **y** in α.

Finally, the value of S_u was set to a when q^y performed line **y.g**. By Lemma 3, there is no other process with the same value of u whose program counter is **y.g** between C_k and C_{j-1}. Thus, the value of S_u remains unchanged during this period of time. Therefore, $val(S_u, C_{j-1}) = a$ and $val(S_u, f(C_{j-1})) = a$ by P6 of Definition 1.

Case 2-2: There exists a previous invocation r^z that performed **x.a** using the same variable u. Then r^z performed **x.f** before p^x performs **x.a**, since there are no interfering events in β'. Thus, when r^z performed **x.d**, the value of $Signal[u]$ was set to 0. Let r^z be the last such invocation.

Since p^x gets 1 from $Signal[u]$ and **x.b**, **x.d**, and **y.h** are the only lines that can change the value of $Signal[u]$, there exists another invocation q^y that performed **y.h** before p^x performed line **x.c** and after r^z performed line **x.d**. Note that, by Requirement 4, there is only one invocation that performed **y.h** before p^x performed line **x.c** and after r^z performed line **x.d**. Thus, q^y performed **y.h** and received 0 from $Signal[u]$. Hence, the value of S_u was set to a when q^y performed line **y.g**.

Note that, by Lemma 3, no other invocation performed line **y.g** before C_{j-1}, but after q^y performed line **y.g**. Thus, during this period of time, no invocation changes the value of S_u, so $val(S_u, C_{j-1}) = a$. Also, q^y skipped lines **y.i** and **y.j**, so at C_{j-1}, no process with the same value of u had a program counter in $\{\mathbf{y.g}, \mathbf{y.h}, \mathbf{y.i}\}$. Thus, P6 of Definition 1, $val(S_u, f(C_{j-1})) = val(S_u, C_{j-1}) = a$.

Case 3: E_j = (p^x, **x.c**) with $Lock[p, u]$ = UNLOCKED. By Definition 1, $pc(p, f(C_{j-1}))$ = **x** and $pc(p, f(C_j))$ = **x+1**. For all processes $q \neq p$, $pc(q, C_j) = pc(q, C_{j-1})$, so $pc(q, f(C_j)) = pc(q, f(C_{j-1}))$. Event E_j does not change the value of any variable. Also, only p's program counter is changed from C_{j-1} to C_j, and $pc(p, C_{j-1})$,$pc(p, C_j)$ $\notin$ $\{\mathbf{y.g}, \mathbf{y.h}, \mathbf{y.i}\}$. Thus, for all $v \in \mathcal{V}$, $val(v, C_j) = val(v, C_{j-1})$. Hence, by lemma 6, $val(v, f(C_j)) = val(v, f(C_{j-1}))$.

Now, we show that $(f(C_{j-1}), (p^x, \mathbf{x}), f(C_j))$ is an admissible execution segment of the input algorithm, which shows that $\alpha, (p^x, \mathbf{x}), f(C_j)$ is an admissible execution of the input algorithm that is in $\mathcal{F}(\beta)$. To do so, it suffices to prove that $val(S_u, f(C_{j-1})) = a$.

Consider the last configuration C_{k-1} in which $val(Lock[p, u], C_{k-1})$ = LOCKED. Suppose that invocation r^z changed the value of $Lock[p, u]$ to UNLOCKED at E_k. Thus, $k \leq j-1$, $E_k = (r^z, \mathbf{y.j})$, $val(i_r, C_{k-1}) = u$, $val(waiter_r, C_{k-1}) = p$, and p does not perform **x.e** with $i_p = u$ between C_{k-1} and C_{j-1}. Since there is no interfering event in β, no process other than r has its program counter in $\{\mathbf{y.g}, \mathbf{y.h}, \mathbf{y.i}, \mathbf{y.j}\}$ at C_{k-1} and C_k.

Thus, $val(S_u, f(C_k)) = a$ by P6 of Definition 1 since $pc(r, C_k) =$ **y+1** and $val(S_u, f(C_k)) = val(S_u, C_k)$.

Since $E_j = (p^x,$ **x.c**$)$, p^x performed **x.b** with $Signal[u] = 0$. $E_k = (r^z,$ **y.j**$)$ implies r^z performed **y.h** with $Signal[u] = 1$. We will prove that (p^x, **x.b**) precedes (r^z, **y.h**).

Suppose, for contradiction, that (p^x, **x.b**) does not precede (r^z, **y.h**). Then, some invocation s^u resets $Signal[u]$ after (r^z, **y.h**) and before (p^x, **x.b**) by performing **x.d**. Since there is no interfering event, p^x performs **x.a** after s^u performs **x.f**. If r^z performs **y.j** before s^u performs **x.f**, then r^z performs its line **y.i** before p^x performs line **x.a**. Thus, r^z cannot read the identity of p from $Want[u]$ when it performs line **y.i**. Hence, r^z cannot access $Lock[p, u]$ in its line **y.j**, which contradicts the assumption that r^z changed the value of $Lock[p, u]$ to UNLOCKED.

If r^z performs **y.j** after s^u performs **x.f**, then there exists a configuration C' in α such that $pc(r^z, C') =$ **y** and $pc(s^u, C') =$ **x+1**. This violates Lemma 4. Therefore, (p^x, **x.b**) precedes (r^z, **y.h**).

Hence, for all $k < l < j$, $pc(p, C_l) =$ **x.c**, so no process can reset S_u. Therefore, $val(S_u, C_{j-1}) = val(S_u, C_j) = a$. No process with the same value of u has a program counter that is in $\{$**y.g**, **y.h**, **y.i**$\}$ in either C_{j-1} or C_j. Thus, By P6 of Definition 1, $val(S_u, f(C_{j-1})) = val(S_u, f(C_j)) = a$. Hence, $(f(C_{j-1}), (q^y,$ **x**$), f(C_j))$ is an admissible execution segment of the input algorithm.

Case 4: E_j = (p^x, **y.h**) with $test\&set(Signal[u]) = 0$ or E_j = (p^x, **y.j**), In these cases, $pc(p, f(C_{j-1}))$ = **y** and $pc(p, f(C_j))$ = **y+1**. Event (p^x, **y.h**) only changes the value of $Signal[u]$ and (p^x, **y.j**) only changes the value of $Lock[waiter_p, u]$. Neither of these variables are in $\mathcal{V}$. Thus, for all $v \in \mathcal{V}$, $val(v, C_{j-1}) = val(v, C_j)$ and, for all $v \in \mathcal{V} - \{S_u\}$, $val(v, f(C_{j-1})) = val(v, f(C_j))$ by Definition 1. The value of S_u was set to a by event (p^x, **y.g**). Since there is no interfering event, no other process updated the value of S_u before C_j, and there does not exist a process $q \neq p$ such that $val(i_q, C_j) = u$ and $pc(q, C_j) \in \{$**y.h**, **y.i**, **y.j**$\}$.

Thus, by P6 of Definition 1, $val(S_u, f(C_j)) = val(S_u, C_j) = a$. Since the value of S_u is set to a and no other variables in V are changed, $(f(C_{j-1}), (p^x,$ **y**$), f(C_j))$ is an admissible execution segment of the input algorithm. Therefore, $\alpha, (p^x,$ **y**$), f(C_j)$ is an admissible execution of the input algorithm that is in $\mathcal{F}(\beta)$. □

From Lemma 7 and 8, we have the following lemma.

Lemma 9. *If $\beta \in \mathcal{B}'$, then $\mathcal{F}(\beta)$ is not empty.*

Proof. If $\beta = C_0$, where C_0 is an initial configuration of the transformed algorithm, then by Lemma 7, $f(C_0)$ is also an initial configuration of the input algorithm. Thus, $f(C_0) \in \mathcal{F}(\beta)$, so $\mathcal{F}(\beta)$ is not empty.

Now, let $\beta = C_0, E_1, C_1, \ldots, C_{j-1}, E_j, C_j$, and let β' be the prefix of β that ends with C_{j-1}. Suppose, for an induction hypothesis, that $\mathcal{F}(\beta')$ is not empty. Then, by Lemma 8, $\mathcal{F}(\beta)$ is also not empty. □

Now we will show that there is no interfering event in any admissible execution of the transformed algorithm.

Lemma 10. *$\mathcal{B}$= $\mathcal{B}'$.*

Proof. Suppose, to obtain a contradiction, that there exists an admissible execution β = C_0, E_1, C_1, ...,C_{j-1}, E_j, C_j, E_{j+1}, C_{j+1}, ... that is in $\mathcal{B}-\mathcal{B}'$. Let E_j be the first interfering event of β. Let β' = C_0, E_1, C_1, ...,C_{j-1}. Then, $\beta' \in \mathcal{B}'$. Thus, by Lemma 9, $\mathcal{F}(\beta')$ is not empty.

Let $\alpha \in \mathcal{F}(\beta')$. Then, α is an admissible execution of the input algorithm such that the sequence of $f(C_0)$, $f(C_1)$, $f(C_2)$, ..., $f(C_{j-1})$ with consecutive duplicates removed is a subsequence of α. Let p^x be the invocation that performs E_j. Since E_j is the first interfering event in β, it is either $(p^x,$ **x.a**$)$ or $(p^x,$ **y.g**$)$.

Case 1: $E_j = (p^x,$ **x.a**$)$. Since E_j is an interfering event, there exists an invocation q^y such that $pc(q, C_{j-1}) \in \{$**x′.a**, ..., **x′.f**$\}$ and $val(i_q, C_{j-1}) = val(i_p, C_{j-1})$, where **x′** $\in$ **X**. Then, $pc(q, f(C_{j-1}))$ = **x′** or **x′+1** by Definition 1. Since $pc(p, C_{j-1})$ = **x.a**, $pc(p, f(C_{j-1}))$ = **x** by P2 of Definition 1. Thus, in α, we have $pc(q, f(C_{j-1})) \in$ **X** $\cup$ **X+1**, $pc(p, f(C_{j-1})) \in$ **X**, and $val(i_q, C_{j-1}) = val(i_p, C_{j-1})$. This contradicts Lemma 2.

Case 2: $E_j = (p^x,$ **y.g**$)$. Since E_j is an interfering event, there exists an invocation q^y such that $pc(q, C_{j-1}) \in \{$**y′.g**, ..., **y′.j**$\}$ and $val(i_q, C_{j-1}) = val(i_p, C_{j-1})$, where **y′** $\in$ **Y**. Then, by Definition 1,$pc(q, f(C_{j-1}))$ = **y′**. Also, since $pc(p, C_{j-1})$ = **y.a**, $pc(p, f(C_{j-1}))$ = **y**. Hence, in α, we have $pc(p, f(C_{j-1}))$, $pc(q, f(C_{j-1})) \in$ **Y** and $val(i_q, C_{j-1}) = val(i_p, C_{j-1})$, which violates Requirement 4 of the input algorithm. □

Since there is no admissible execution of the transformed algorithm that has an interfering event, we can extend Lemma 9 to the following lemma.

Lemma 11. *For all $\beta \in \mathcal{B}$, $\mathcal{F}(\beta)$ is not empty.*

Theorem 12. *Transformation HL1 correctly transforms any local-spin mutual exclusion algorithm on the CC model that satisfies the requirements into a local-spin mutual exclusion algorithm on the DSM model.*

Proof. We prove this theorem by contradiction. Suppose, for contradiction, that the transformed algorithm does not satisfy mutual exclusion or lockout freedom. Then, there exists an admissible execution of the transformed algorithm in which mutual exclusion or lockout freedom is violated.

Let $\beta = C_0, E_1, C_1, \ldots$ be such an execution. Then, by Lemma 11, $\mathcal{F}(\beta)$ is not empty. Let $\alpha \in \mathcal{F}(\beta)$.

Case 1: β violates mutual exclusion.

Then, in β, there exists a configuration C in which two processes p and q are in the critical section. Let **C** be the set of all lines of the critical section of the transformed algorithm, and let **z** be the first line of the exit section of the transformed algorithm. Then, $pc(p, C), pc(q, C) \in$ **C** $\cup \{$**z**$\}$. Let **C**$'$ be the set of all lines of the critical section of the input algorithm, and let **z**$'$ be the first line of the exit section of the input algorithm.

By Definition 1, $pc(p, f(C)) = pc(p, C)$ and $pc(q, f(C)) = pc(q, C)$, except when **z** $\in \{$**x.a**, **y.g**$\} \cap \{pc(p, C), pc(q, C)\}$. But, if **x.a** is the first line of the exit section of the transformed algorithm, then **x** is the first line of the exit section of the input algorithm, i.e. **z**$'$ = **x**. Also, if $pc(p, C)$ = **x.a**, then $pc(p, f(C))$ = **x** = **z**$'$, and if $pc(q, C)$ = **x.a**, then $pc(q, f(C))$ = **x** = **z**$'$. Similarly, if **y.g** is the first line of the exit section of the transformed algorithm, then $pc(p, C) =$ **y.g** implies $pc(p, f(C)) =$ **z**$'$, and $pc(q, C) =$ **y.g** implies $pc(q, f(C)) =$ **z**$'$

Thus, $pc(p, f(C)), pc(q, f(C)) \in$ **C**$' \cup \{$**z**$'\}$. Hence, p and q are in the critical section of the input algorithm in $f(C)$. Since $f(C)$ is a configuration of α, α does not satisfy mutual exclusion. This contradicts the fact that the input algorithm satisfies mutual exclusion.

Case 2: β violates lockout freedom.

In this case, β is an infinite admissible execution of the transformed algorithm that contains an invocation p^x that remains in the entry section of the transformed algorithm forever. Let C_j be the first configuration where p^x is in the entry section. Let **A** be the set of all lines in the entry section of the transformed algorithm, and let **z** be the first line of the critical section of the transformed algorithm. Let **A**$'$ be the set of all lines in the entry section of the input algorithm, and let **z**$'$ be the first line of the critical section of the transformed algorithm. Then, for all $k \geq j$, $pc(p, C_k) \in$ **A** $\cup \{$**z**$\}$.

Note that if **x.a**, …, **x.f** are in the entry section of the transformed algorithm, then **x** is in the entry section of the input algorithm and **x+1** is in the entry section or the first line of the critical section of the input algorithm. Similarly, if **y.g**, …, **y.j** are in the entry section of the transformed algorithm, then **y** is in the entry section of the input algorithm and **y+1** is in the entry section or the first line of the critical section of the input algorithm. Thus, for all $k \geq j$, $pc(p, f(C_k)) \in$ **A**$' \cup \{$**z**$'\}$.

Since $f(C_j), f(C_{j+1}), \ldots$ is an infinite subsequence of α and, for all $k \geq j$, $pc(p, f(C_k)) \in$ **A**$' \cup \{$**z**$'\}$, α does not satisfy lockout freedom, which contradicts the fact that the input algorithm satisfies lockout freedom.

In both cases, we reach a contradiction, so the transformed algorithm satisfies mutual exclusion and lockout freedom. Note that any algorithm that results from applying Transformation HL1 is a local-spin algorithm on the DSM model, since processes use local spin variables in all busy-waiting periods. Therefore, Transformation HL1 transforms any local-spin mutual exclusion algorithm on the CC model that satisfies the requirements into a local-spin mutual exclusion algorithm on the DSM model. □

3.3 Transformation HL2

Transformation HL2 is the first transformation that uses only registers. In Transformation HL2, the set of the introduced variables, $\mathcal{U}$, is $\{pre_p$, $waiter_p$, $Succ[i]$, $Pred[i]$, $Done[p, i]$, $Lock1[p, i]$, $Lock2[p, q] \mid p, q \in \mathcal{P}$ and $i \in I\}$. Transformation HL2 replaces line **x** of the input algorithm with lines from **x.a** to **x.i** and line **y** of the input algorithm with lines from **y.j** to **y.q**. Figure 2 describes Transformation HL2 in detail.

Transformation HL2 can be described informally as follows. Let p be a process, and q be the predecessor of p. In Transformation HL2, p uses two local shared variables $Lock1[p, i]$ and $Lock2[p, q]$ as spin variables instead of S_i, the spin variable of the input algorithm. The role of $Succ[i]$ in Transformation HL2 is similar to that of $Want[i]$ in Transformation HL1. In Transformation HL2, the successor, p, writes its identity to $Succ[i]$, and q reads $Succ[i]$ to get the identity of p.

However, unlike Transformation HL1, it is possible that q accesses $Lock1[p, i]$, and p does not busy-wait on $Lock1[p, i]$. To avoid this unwanted unlocking event from interfering with a later invocation of p that uses the same index i, Transformation HL2 has another busy-waiting period for p. After finishing the first busy-waiting period, process p knows the identity of its predecessor, q, because q writes its identity to $Pred[i]$ and $Lock1[p, i]$. Therefore, in the second busy-waiting period, p can use spin variable $Lock2[p, q]$, which can be accessed only by p and q. At this time, shared variable $Done[q, i]$ is used to determine whether p spins on $Lock2[p, q]$. Although it is still possible

```
Shared variables introduced:
    $Lock1[p, i] \in \{$NIL, 0, …, $N-1\}$ initially NIL
    $Lock2[p, q] \in \{$LOCKED, UNLOCKED$\}$
            initially LOCKED
    $Pred[i] \in \{$NIL, 0, …, $N-1, N\}$
            if $S_i$ is initially $a$, then initially $N$.
            otherwise, initially NIL
    $Succ[i] \in \{$NIL, 0, …, $N-1\}$ initially NIL
    $Done[p, i] \in \{$True, False$\}$ initially True
    $Done[N, i] =$ True
    where $p, q \in \{0, \ldots, N-1\}$ and $i \in I$
```

```
instead of performing:
    x:    await $S_i = a$
process $p$ performs:
    x.a:  $pre$ := NIL
          $Lock1[p, i]$ := NIL
    x.b:  $Succ[i] := p$
    x.c:  $pre := Pred[i]$
          if $pre$ = NIL then
    x.d:      while $pre$ = NIL do
                  $pre := Lock1[p, i]$ /* local spin */
              od fi
    x.e:  $Lock2[p, pre]$ := LOCKED
    x.f:  if $\neg Done[pre, i]$ then
    x.g:      await $Lock2[p, pre]$ = UNLOCKED
                  /* local spin */ fi
    x.h:  $Pred[i]$ := NIL
    x.i:  $Succ[i]$ := NIL
instead of performing:
    y:    $S_i := a$
process $q$ performs:
    y.j:  $Done[q, i]$ := False
    y.k:  $Pred[i] := q$
    y.l:  $waiter := Succ[i]$
          if $waiter \neq$ NIL then
    y.m:      $Lock1[waiter, i] := q$ fi
    y.n:  $S_i := a$
    y.o:  $Done[q, i]$ := True
    y.p:  $waiter := Succ[i]$
          if $waiter \neq$ NIL then
    y.q:      $Lock2[waiter, q]$ := UNLOCKED
              $waiter$ := NIL fi
```

Figure 2. Transformation HL2

that q accesses $Lock2[p, q]$, and p does not busy-wait, this unlocking event by q does not affect a later invocation of p with the same index i. That is because the later invocation of p cannot use $Lock2[p, q]$ unless its predecessor is also q.

3.4 Modifications of the Transformations

For some input algorithms, it is difficult to tell whether $S_i := a$ is an unlocking event or resetting event, especially if a is not a constant. In this case, we replace $Signal[i]$ with $Signal[i, a]$ in Transformation HL1, and replace $Pred[i]$, $Succ[i]$, and $Done[q, i]$ with $Pred[i, a]$, $Succ[i, a]$ and $Done[q, i, a]$ in Transformation HL2.

4 Conclusion

We showed that if a local-spin mutual exclusion algorithm on the CC model satisfies the requirements in this paper, then there is also a local-spin mutual exclusion algorithm on the DSM model that has the same time complexity. This result is another indication that the DSM model could have the same power to solve the mutual exclusion problem as the CC model.

It is open whether it is possible to transform all local-spin mutual exclusion algorithms from the CC model to the DSM model without increasing the number of RMAs they perform by more than a constant factor.

Acknowledgement: I would like to thank my supervisor Prof. Faith Ellen Fich for her guidance and careful corrections. This research is supported by Natural Sciences and Engineering Research Council of Canada.

References

[1] Jae-Heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

[2] James H. Anderson and Yong-Jik Kim. An improved lower bound for the time complexity of mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99, August 2001.

[3] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] James Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informtica*, 30(3):249–265, May 1993.

[5] James H. Anderson and Yong-Jik Kim. Local-spin mutual exclusion using fetch-and-ϕ primitives. In *The 23rd International Conference on Distributed Computing Systems*, May 2003.

[6] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 2002.

[7] Robert Danek and Vassos Hadzilacos. Local-spin group mutual exclusion algorithms. In *Proceedings of the 18th International Symposium on Distributed Computing*, pages 71–85, October 2004.

[8] Yong-Jik Kim and James H. Anderson. Timing-based mutual exclusion with local spinning. In *Proceedings of the 17th International Symposium on Distributed Computing*, pages 30–44, October 2003.