University of Groningen

Group mutual exclusion by fetch-and-increment

Aravind, Alex A. ; Hesselink, W H

Published in:
ACM Transactions on Parallel Computing

DOI:
[10.1145/3309202](https://doi.org/10.1145/3309202)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2019

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):

Aravind, A. A., & Hesselink, W. H. (2019). Group mutual exclusion by fetch-and-increment. *ACM Transactions on Parallel Computing*, 5(4), [14]. <https://doi.org/10.1145/3309202>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Group Mutual Exclusion by Fetch-and-increment

ALEX ARAVIND, University of Northern British Columbia, Canada

WIM H. HESSELINK, University of Groningen, The Netherlands

The group mutual exclusion (GME) problem (also called the room synchronization problem) arises in various practical applications that require concurrent data sharing. Group mutual exclusion aims to achieve exclusive access to a shared resource (a shared room) while facilitating concurrency among non-conflicting requests. The problem is that threads with distinct interests are not allowed to access the shared resource concurrently, but multiple threads with same interest can.

In Blelloch et al. (2003), the authors presented a simple solution to the room synchronization problem using *fetch&add* (F&A) and *test-and-set* (T&S) atomic operations. This algorithm has $O(m)$ remote memory references (RMRs) in the cache coherent (CC) model, where m is the number of forums. In Bhatt and Huang (2010), an open problem was posed: “Is it possible to design a GME algorithm with constant RMR for the CC model using *fetch&add* instructions?” This question is partially answered in this article by presenting a group mutual exclusion algorithm using *fetch-and-increment* instructions. The algorithm is simple and scalable.

CCS Concepts: • **Computer systems organization** → **Parallel architectures; Multicore architectures;**
• **Software and its engineering** → **Multithreading; Mutual exclusion; Concurrency control;**

Additional Key Words and Phrases: Concurrency, algorithms, group mutual exclusion, constant remote memory access, fetch-and-increment, concurrent data structure, synchronization

ACM Reference format:

Alex Aravind and Wim H. Hesselink. 2019. Group Mutual Exclusion by Fetch-and-increment. *ACM Trans. Parallel Comput.* 5, 4, Article 14 (February 2019), 29 pages.

<https://doi.org/10.1145/3309202>

1 INTRODUCTION

Mutual exclusion and concurrency are two fundamental requirements in concurrent programming and distributed computing. The mutual exclusion problem ensures that no two threads can execute the critical section (CS) simultaneously. This problem has been well known for half a century (Dijkstra 1965). The group mutual exclusion (GME) or room synchronization problem is a generalization of the mutual exclusion problem, introduced by Joung in 1998. Group mutual exclusion aims at achieving exclusive access to the shared resources while facilitating suitable concurrency (Joung 1998).

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant (312162-2015).

Authors' addresses: A. Aravind, Department of Computer Science, University of Northern British Columbia, 3333 University Way, Prince George, BC, Canada, V2N 4Z9; email: csalex@unbc.ca; W. H. Hesselink, Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, Faculty of Science and Engineering, University of Groningen, P.O. Box 407, 9700 AK Groningen, The Netherlands; email: w.h.hesselink@rug.nl.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

2329-4949/2019/02-ART14 \$15.00

<https://doi.org/10.1145/3309202>

Let us first introduce the group mutual exclusion problem informally using an example from university setting. Think of the CS as a conference room that can be shared by different departments for their colloquium talks. Obviously, two different colloquium talks cannot be arranged in the conference room at the same time. Here, people (threads) interested in a same talk (forum) can enter the CS and attend the talk together concurrently; threads interested in different forums are not allowed to enter the CS concurrently. This problem also can be considered as a generalization of the readers/writers problem, where all readers can attend the same forum while ensuring each writer attends a unique forum in isolation. In terms of concurrency, the group mutual exclusion problem has some similarity to the l -exclusion problem (Fischer et al. 1989).

As resource integration and resource sharing are becoming increasingly common due to the recent multicore revolution, group mutual exclusion is an important issue in many applications. Consider implementing a parallel stack ST using an atomic *move*(k) instruction, which moves the stack pointer *sp* (up if $k > 0$, down if $k < 0$) by k positions and returns the old address of *sp* (Blelloch et al. 2003). The stack ST grows from 0 up and the pointer *sp* points to the last occupied location in the stack. The stack operation *push* places an item in ST at the position next to *sp* and moves *sp* one position up. The operation *pop*, when ST is nonempty, returns the last inserted item pointed by *sp* and pushes *sp* one position down. Using *move* instruction, we can implement *push*(x) as $ST[move(1)] = x$, and $x = pop()$ as $x = ST[move(-1) - 1]$ if $sp > 0$. Interleaving of two or more *push* operations or two or more *pop* operations is safe, because the order does not matter when the stack pointer *sp* is moving in the same direction. Interleaving of *push* and *pop* operations is not safe. Similarly, group mutual exclusion can be applied to implement several important shared data structures such as shared *first-in-first-out* (FIFO) queue and dynamic shared stack (Blelloch et al. 2003).

1.1 Related Research

Several solutions are reported in the literature to solve the group mutual exclusion problem in the shared memory context. They can be broadly classified based on the machine instructions used to access the shared variables: (a) GME algorithms that use read/write instructions and (b) GME algorithms that use higher-level atomic instructions such as *T&S*, *CAS*, and *F&A*. Each of the algorithms using read/write instructions typically modifies and extends one of the popular mutual exclusion algorithms to solve the GME problem. The GME algorithms that use only read/write atomicity are usually complex compared to the algorithms using higher-level primitives. Since the GME algorithms presented in this article require a higher level atomic instruction *F&I*, we compare them only with the GME algorithms requiring higher-level atomic instructions presented in the literature (Bhatt and Huang 2010; Blelloch et al. 2003; Danek and Hadzilacos 2004).

Finding the appropriate measure of time complexity is difficult for concurrent algorithms for the systems where nothing is assumed about the speed of the threads. A measure that is often used in this context is the number of remote memory references (RMR) (Anderson et al. 2003; Bhatt and Huang 2010). There are two popular shared memory models—distributed shared memory (DSM) and cache coherent (CC) shared memory. In DSM models, a reference to a shared variable X by a thread p is considered remote if X is at a memory module of a processor different than p . In CC models, a reference to X by a thread p is considered remote if X is not in p 's cache. It is shown in Danek and Hadzilacos (2004) that for DSM models $\Omega(n)$ RMR is required to solve the group mutual exclusion problem using primitives of any strength, where n is the total number of threads in the system. The algorithm presented in this article solves the GME problem in the CC shared memory model. We, therefore, restrict our discussion in the rest of the article to CC shared memory models.

1.2 Background Motivation for Constant RMR GME in CC Model Using *F&A*

In Bhatt and Huang (2010), the authors proposed a group mutual exclusion algorithm for the CC model using compare-and-swap (*CAS*) instructions. This algorithm involves $O(\min(k, \log n))$

RMR, where n is the total number of threads in the system, and k is the point contention. The main drawback of this algorithm is that it requires unbounded shared space. In the end, authors posed an open problem: “*Is it possible to design a constant RMR GME algorithm using fetch&Add instructions?*”

Related to concurrency, three properties, namely concurrent occupancy (Joung 1998), concurrent entering (Hadzilacos 2001), and no late entry (Blelloch et al. 2001), are most important. These are formally defined and discussed further in Section 2. Concurrent entering is a stronger version of concurrent occupancy and it allows threads to re-enter the same forum again. The no late entry property does not allow such a re-entry, and this property is required for some important applications illustrated in Blelloch et al. (2001). This distinction in concurrency creates two versions of the open problem posed in Bhatt and Huang (2010) and also stated above—one for concurrent entering and the other for no late entry.

In Blelloch et al. (2003), the authors proposed a simple algorithm using $F\&A$ and $T\&S$ instructions (the earlier version uses $F\&A$ and CAS instructions (Blelloch et al. 2001)). The algorithm employs the round-robin strategy for forum scheduling. Round-robin is a good scheduling strategy for high interactive response but not preferable from the fairness point of view. This brings us to an expanded version of the open problem in CC model: *Is it possible to design a constant RMR GME algorithm that assures high fairness and no late entry property using $F\&A$ instruction?* We present a simple GME algorithm that answers this question affirmatively. As a matter of fact, our algorithm uses the operation *fetch&increment* ($F\&I$), which is simpler than $F\&A$.

The operation $F\&I(x)$ sets the value of x to $x + 1$ and returns the old value of x . Compared to the $F\&A$ operation, $F\&I$ is easier to implement efficiently in both hardware and software (Ellen et al. 2012; Freudenthal and Gottlieb 1991). The popular approach to design concurrent data structures is to use the most powerful synchronization primitive, namely *compare-and-swap* (CAS), mainly to avoid hotspot operations such as incrementing a contented shared counter. A recent study (Morrison and Afek 2013) on a multicore system with 4 Intel Xeon E7-4870 processors shows that, when compared to using CAS loop for incrementing a contented shared counter, a less-powerful primitive *fetch-and-add* ($F\&A$) available on x86 processors can lead to 4 to 6 times performance improvement. As the x86 architectures dominate server and desktop markets, the above observation increases the practical appeal of concurrent algorithms based on $F\&A$ or $F\&I$ operations.

As noted above, establishing a suitable measure to capture the time complexity of concurrent algorithms is more difficult than that of sequential algorithms because of the intricacies of concurrent executions and the apparent delays due to synchronization requirements. In this article, next to counting RMRs, we use a concurrent time complexity measure based on the number of rounds needed to establish some property. A round is an execution fragment in which every thread does at least one step or is found disabled at least once. Counting the number of rounds occurs as a side product of the progress proof that is based on a bounded version (Hesselink 2015a) of UNITY (Chandy and Misra 1988).

1.3 Contributions and Roadmap

This article has five key contributions on group mutual exclusion: (i) It presents two new algorithms with correctness proofs; (ii) the second algorithm assures constant RMR in CC models (that partially solves the open problem posed in Bhatt and Huang (2010)¹); (iii) it introduces two additional properties, Simultaneous acceptance and Forum FCFS; and (iv) it proves that the algorithm has constant concurrent time complexity in the sense of bounded UNITY.

¹Solving the open problem posed in Bhatt and Huang (2010) completely requires the algorithm to satisfy both constant RMR and concurrent entering properties.

Our algorithms are conceptually simpler and more elegant than most of the algorithms proposed to solve group mutual exclusion. In addition to simplicity and elegance, time complexity is an important measure of algorithm, as it quantifies the economy of system resources that it requires to solve the problem. The best possible time complexity measure is constant time (denoted as $O(1)$). Constant time complexity means that the performance of the algorithm does not depend on the size of the input (the number of threads in this case). Our final algorithm assures constant RMR. The rest of the article is organized as follows.

We discuss the various properties that can be required for GME in Section 2. In Section 3, the GME algorithm with constant RMR is developed incrementally. The correctness of the resulting algorithms is verified in Sections 4 and 5. The safety property is proved in Section 4. The progress assertions with bounds on the concurrent time complexity are presented in Section 5.3. The assertions of the Sections 4 and 5 have been mechanically verified with the proof assistant PVS (Owre et al. 2001). The PVS-dump file is available for inspection in Hesselink (2015b). A limited performance study is presented in Section 7, and Section 8 contains a comparison summary. The article is concluded in Section 9.

2 THE PROBLEM STATEMENT

We consider a system of independent cyclic threads that communicate by shared variables and a set of m forums. Threads interested in a same forum can access the shared resource (shared room) concurrently. Here forum is the critical section (CS), and therefore we use forum and CS interchangeably. We assume that (i) the speed of each thread is arbitrary but finite, (ii) the time each thread spends inside the CS is arbitrary but finite, (iii) no thread can fail while executing the algorithm, and (iv) threads communicate only through shared variables. The algorithms presented in this article, in addition to *read* and *write* operations, use the atomic operation *F&I* to access the shared (communication) variables.

The code segment of the threads can be divided into two parts: the part in which the thread attends the forum (CS) and the remaining part (NCS). Structurally, a GME algorithm has two parts: an *entry section* (the code to be inserted before entering the CS) and an *exit section* (the code to be inserted after exiting the CS). These components have to be inserted appropriately in the code of all threads to solve the GME problem. The *GME problem* is to design an algorithm that satisfies several properties, suitable for different applications. We will discuss these properties next.

2.1 Concurrency Properties

To maximize the utilization of the shared resource, Joung stated: “We are seeking solutions that facilitate **concurrent entering**, meaning that if some philosophers are interested in a forum and no philosopher is interested in a different forum, then the philosophers can concurrently enter the meeting room to hold the forum” (Joung 1998). Subsequently, Kean and Moir defined this property more formally as follows:

“Concurrent Entering: If process p requests a session $p.t$ and no process q such that $q.t \neq p.t$ is executing outside its non-critical section, then p eventually reaches its critical section (even if no other process leaves its critical section)” (Keane and Moir 1999).

Later, Hadzilacos noted that:

“Unfortunately, the phrase “can concurrently enter,” although suggestive, is not precise. Joung contrasts concurrent *entering* with a different and weaker requirement that might be called “concurrent *occupancy*,” in which processes requesting the same session can occupy the CS concurrently but may delay each other during entry . . . Kean and Moir [6] gave a precise but, as we will argue, weaker definition

of concurrent entering. Their definition captures the concurrent occupancy requirement, but not the stronger one of concurrent entering” (Hadzilacos 2001).

Then, he went on to introduce a stronger definition for concurrent entering that must guarantee the later threads interested in the same forum to enter the forum within a bounded number of their own steps when no other forum is in demand. This property appears to be more demanding than Joung’s original intent.

Now, let us look at the question of whether concurrent entering should be included as a required property of the GME problem. First, some earlier solutions presented in Keane and Moir (1999), Alagarsamy and Vidsankar (1999), Takumura and Igarashi (2003), and Blleloch et al. (2003) do not satisfy this concurrent entering property. So adding this concurrent entering property now to the list of required property for the GME problem disqualifies these earlier solutions to the problem. Second, as noted in (Blleloch et al. 2003), some applications require no concurrent entering. In Blleloch et al. (2003), the authors use their GME algorithm to design concurrent algorithms to implement dual concurrent write and dynamic stacks. The authors state that “The algorithms are not correct if concurrent entering is allowed.” So adding this concurrent entering property defined by Hadzilacos (2001) to the list of required properties is also not very helpful from the perspective of some important applications where GME is applied. Finally, concurrent entering property is effective only when there are no conflicting requests. Otherwise, without a proper concurrency requirement during conflicting requests, GME appears to be a simple mutual exclusion if there are conflicting requests. Actually, the GME problem becomes non-trivial and more interesting only when there are threads competing with conflicting requests. Therefore, we need a suitable concurrency property that can be effective even if there are conflicting requests as a requirement for GME algorithms.

The property of *no late entry* introduced in Blleloch et al. (2001) is effective even if there are conflicting requests. But this is a restrictive property that prohibits concurrency after a particular moment during the period a forum is held. In this article, we propose a new concurrency property, called *simultaneous acceptance*, that we believe is effective and assures a reasonable level of concurrency even if there are conflicting requests.

2.2 Fairness Properties

Fairness is an important property of an algorithm that facilitates multiple competing threads in accessing a shared resource. When multiple forums (resources) are involved, fairness can be viewed from two perspectives: (i) fairness among threads in accessing the forums (threads’ perspective) and (ii) fairness in utilizing the forums (forums’ perspective). For GME, however, we have two cases: (i) fairness among the threads interested in a same forum and (ii) fairness among the threads interested in different forums. First-come-first-serve (FCFS) is the most widely known and used fairness metric.

FCFS fairness property for threads interested in different forums has been introduced in Hadzilacos (2001), and FCFS fairness property for threads interested in a same forum has been introduced in Jayanti et al. (2003). Here, we introduce a new property called *Forum First-Come-First-Served* (F-FCFS) for fairness among forums. This new property F-FCFS is aimed at ensuring balanced utilization of shared resources.

2.3 Properties of Group Mutual Exclusion

To make the original idea of Joung concrete, we offer three suggestions: (i) categorize the properties into three classes—base properties, concurrency properties, and fairness properties; (ii) introduce a new property called *Simultaneous Acceptance* (Property P4) into the list of concurrency properties;

and (iii) establish all the base properties and at least one of the concurrency properties as the required properties of a GME algorithm.

To present the above properties formally, we define some concepts first. The entry section begins with the choice of a forum, say, f , followed by a code fragment without waiting, called the *doorway*, followed by a *waiting section*, optionally followed by an *opening section* without waiting. Starting the doorway with choice f is called *initiating a request* of f . The thread has *requested* the forum when it is in the waiting section. The request is *answered* when the thread enters the CS. A thread interested in f is *enabled* to attend f if it can reach the CS in a bounded number of its own steps.

A forum f is *closed* when no threads interested in f are enabled to enter the CS. A forum f is *opened* when there are threads interested in f that are enabled to enter the CS; no other threads interested in f have entered the CS since the forum f was last closed. A forum f is *active* when there are threads interested in f that are in the CS. Such threads are said to be *attending* forum f . Forum f is said to be in *demand* when there are threads interested in f that have initiated requests for f .

Base Properties

- *Safety (Mutual Exclusion)* [P1]: If some thread is attending a forum, then no thread is attending a different forum at the same time (Blelloch et al. 2003; Joung 1998).
- *Liveness (Deadlock-Freedom)* [P2]: When one or more threads have requested forums, one of the requests is eventually answered (Blelloch et al. 2003; Joung 1998).
- *Individual Progress (Lockout-Freedom)* [P3]: Every request is eventually answered and the requesting thread eventually returns to the noncritical section (terminating exit) (Blelloch et al. 2003; Hadzilacos 2001; Joung 1998).

Concurrency Properties

- *Simultaneous Acceptance* [P4]: Let S be the set of threads that have completed their requests for a forum f , when a thread p with interest f enters its opening section. Then, when p completes its opening section, every thread in S enters f within a bounded number of its own steps.
- *No Late Entry* [P5]: No thread p is allowed to enter an open forum f if p initiated its request after f was opened (Blelloch et al. 2003).
- *Concurrent Entering* [P6]: If a thread p requests a forum f and no other thread requests a different forum, then p enters f within a bounded number of its own steps (Hadzilacos 2001).
- *Concurrent Occupancy* [P6']: If a thread p requests a forum f and no other thread request a different forum, then p eventually enters f (Hadzilacos 2001).

Fairness Properties

- *First-come-first-served (FCFS)* [P7]: If a thread p requests a forum before thread q initiates a request for a different forum, then the request of p is answered before the request of q (Hadzilacos 2001).
- *First-in-first-enabled (FIFE)* [P8]: If a thread p requests a forum before thread q initiates a request for the same forum and q enters the forum while p is still waiting, then p enters the forum within a bounded number of its own steps (Jayanti et al. 2003).
- *Forum first-come-first-served (F-FCFS)* [P9]: If forum f is requested before any request for forum f' is initiated, then forum f is opened before forum f' is opened.

A GME algorithm must satisfy all the base properties and at least one of the concurrency properties. The remaining properties are desirable and dependent on the intended application. For

example, both concurrent occupancy and concurrent entering properties allow not only late threads to enter an ongoing forum but also threads to re-enter the forum any number of times. The implementations of dual concurrent write and dynamic stack algorithms, explained in pages 418 and 419 of Blleloch et al. (2003), will fail if a thread is allowed to re-enter a forum. Therefore, such implementations require the no late entry property (P5). Moreover, the property P5 conflicts with both P6 and P6'. The property P5 is the complement of P4. Roughly speaking, P4 and P5 together mean that a thread can enter a forum if and only if its request was done when the forum opens, but, strictly speaking, they do not specify what happens if the thread is still in the doorway when the forum opens.

The properties P4 and P7 cannot be satisfied in conjunction because of the following scenario. Consider three different forums, f_0, f_1, f_2 . Forum f_0 is open, with some attending threads. Then thread t_1 requests forum f_1 . Then thread t_2 requests f_2 . Then thread t_3 requests f_1 . Then forum f_0 closes. By P2, one of the threads f_1 and f_2 must be opened. By P7 for the threads t_1 and t_2 , forum f_1 must be opened before f_2 . By P4, the threads t_1 and t_3 must be admitted. This violates P7 for the threads t_2 and t_3 . This shows that P7 is too restrictive if we insist on P4 (Bhatt and Huang 2010).

Property P8 (FIFE) follows from P4 and P5. This is proved as follows. Assume that thread t_0 has completed the request of forum f before thread t_1 initiates a request for f . Assume now that forum f is opened and thread t_1 enters the forum. By P5, thread t_1 initiated its request before the opening of f . Therefore, thread t_0 completed its request before the opening of f . By P4, thread t_0 enters the forum within a bounded number of its own steps.

One may note that P6 does not follow from P9. In fact, if a thread p completes a request for an open forum f while there are no unanswered requests initiated for other forums, then property P9 implies that thread p can eventually enter forum f , but it allows the possibility that forum f needs to close first with all attending threads and then reopens for thread p . In the case of incompatible properties, like P5 and P6, or P4 and P7, the question of which one is preferred depends on the application. Our two algorithms satisfy P4, P5, P8, and P9. Next, we present the proposed GME algorithms. We use different fonts in the algorithm description to distinguish between private and shared variables: Private variables are slanted and shared variables are in typewriter font.

3 THE ALGORITHMS

The algorithms proposed in this article are inspired by the scalable room synchronization algorithm of Blleloch et al. (2003). This algorithm, we refer to as BCG, is based on the idea of ticket based entry where each thread first “draws” a ticket and then “waits” for its access to the shared resource to be “granted.” If no forum is “open,” then one of the waiting threads succeeds in “opening” its forum. Then, the last thread from the current forum “determines” the next forum to be held. It uses a round-robin policy for choosing the next forum.

The algorithm BCG (reproduced as Algorithm 1 below using our notations) uses three shared integer arrays: `wait` (waiting for forum), `grant` (granted access to forum), and `done` (completed forum access), each of size m . It uses a shared Boolean bit `active` to record if any forum is active or not. We do not use `active`; instead, we use two shared integer variables `head` and `tail` to implement a virtual queue. Initially, all shared variables are 0: $\text{head} = \text{tail} = 0 \wedge (\forall f : \text{done}[f] = \text{grant}[f] = \text{wait}[f] = 0)$.

The algorithm uses the private variables f_m to hold forum id, $ticket$ to hold ticket value, and $session$ to hold the position in the queue. The initial values of the private variables are not important.

Our algorithm deviates from BCG in two ways: (i) in choosing the next forum and (ii) in checking the condition for accessing the forum. To choose the next forum, BCG uses round-robin policy, and our algorithm uses FIFO policy. These two differences make our algorithm simpler.

ALGORITHM 1: BCG (Blelloch et al. 2003) (A1)

shared variables

wait, grant, done : $[forum \rightarrow int]$.
 active : *bool*; activeRoom : *int*.

[Initially, head = tail = active = 0 $\wedge$ ($\forall f : \text{done}[f] = \text{grant}[f] = \text{wait}[f] = 0$)
 $\wedge$ activeRoom = -1];

Thread(*p*) :

```

var fm : forum, ticket, last, currWait, ar, myDone, Ffree : int ;
NCS ; choose fm  $\in$  forum ;
ticket := F&A(wait[fm], 1) + 1 ;
while (ticket - grant[fm] > 0) ;
  if T&S(active) then
    activeRoom := fm ;
    currWait := wait[fm] ;
    grant[fm] := currWait ;
  endif
endwhile
CS ;
ar := activeRoom ;
myDone := F&A(done[ar], 1) + 1 ;
if (myDone = grant[ar]) then
  Ffree := 0 ;
  for (int k := 0; newAr = ar; k < m; k++)
    newAr := (newAr + 1) % m ;
    currWait := wait[newAr] ;
    if (currWait - grant[newAr] > 0) then
      activeRoom := newAr ;
      grant[newAr] := currWait ;
      Ffree := 1 ;
    endif
  endfor
  if (Ffree = 0) then active := 0 ;
endif

```

To check the condition for accessing the forum, all the competing threads in BCG continuously check the status of both grant (to see the access is granted) and active (to check if any forum is active or not). If the forum is not active, then one of the threads that succeed randomly becomes the leader of its forum and grants the access to other threads interested in the same forum. This creates contention at both grant and active. In our algorithm, among the threads waiting for a forum (say, f), except the thread with the smallest ticket, all other threads wait for the access to their forum granted. The thread with the smallest ticket becomes the leader of f , joins the virtual queue, and waits for its turn. When it gets the turn, it grants access to the other threads waiting for f . All other threads interested in f simply wait for their access to be granted by the leader.

3.1 The Fair Group Mutual Exclusion Algorithm

The proposed FCFS GME algorithm works as follows. Each thread starts with choosing a forum (say, fm) to access and then proceeds further to draw its *ticket* from a shared counter $wait[fm]$ using *F&I* atomic operation. Among the threads requesting the same forum, the first thread and the last thread have special roles. We refer the first thread as the “leader” and others as “followers.”

Initially, the leader will have its *ticket* value equal to $grant[fm]$ and its followers will have their *ticket* value greater than $grant[fm]$. The leader joins the virtual queue by drawing its *session* number from the shared counter $tail$ and waits for its turn (i.e., for $head = session$). Followers simply wait for their access to the forum to be granted by the leader. The leader grants permission for all its follower to proceed and access the forum (by setting their $grant[fm] = wait[fm]$) when it gets its turn in the queue. After the forum access, every thread increments $done[fm]$ and determines if it is the last thread of the current session of fm . If so, then it gives the turn to the next thread in the queue by incrementing $head$. To avoid interference at the counters, especially at $tail$, $wait$, and $done$, we use *F&I* instruction. As usual, the instruction **await** B for a condition B means the atomic command that tests B and proceeds only when B holds. It is therefore equivalent to

while $\neg B$ **do** *skip* **endwhile**.

According to the principle of *single critical reference*, an atomic command reads or writes at most one shared variable (not both) (Owicki and Gries 1976, (3.1)), (Apt et al. 2009, p. 273). The assignment $grant[f] := wait[f]$ is, therefore, the sequential composition of reading $wait$ followed by writing $grant$, and other threads can do steps in between. The resulting GME algorithm that incorporates the above idea is given as Algorithm 2. We refer it to as A2.

ALGORITHM 2: Fair group mutual exclusion (A2)

shared variables

$head, tail : int ;$
 $wait, grant, done : [forum \rightarrow int].$

[Initially, $head = tail = 0 \wedge (\forall f : done[f] = grant[f] = wait[f] = 0)$];

Thread(p) :

```

var  $fm : forum, ticket, session, last : int ;$ 
NCS ; choose  $fm \in forum ;$ 
 $ticket := F\&I(wait[fm]) ;$                                 // draw ticket
await  $ticket \leq grant[fm] ;$                                 // wait for access grant, if not first
if  $ticket = grant[fm]$  then                                  // if first then become leader
     $session := F\&I(tail) ;$                                 // join the queue
    await  $head = session ;$                                   // wait for the turn
     $grant[fm] := wait[fm] ;$                                 // grant access to followers
endif
CS ;
 $last := grant[fm] - 1 ;$ 
if  $last = F\&I(done[fm])$  then
     $head++ ;$                                               // give turn to the next thread in the queue
endif

```

In contrast to our algorithm, in the algorithm of Blelloch et al. (2003), the last thread of the previous forum opens the next forum f by setting $\text{grant}[f]$ to $\text{wait}[f]$. This approach might appear appealing theoretically, as it allows all the threads to enter the forum at the same time. In this algorithm, however, all waiting threads continuously use $T\&S$ on a Boolean bit that could create a hotspot contention—a source of performance degradation (Morrison and Afek 2013). In our algorithm, the wait locations are distributed and no higher-level atomic instruction is used during wait periods.

At this point, we see that property P4 and P5 are satisfied. In fact, when thread p has passed the **await** statement of $\text{head} = \text{session}$, it is in the opening section, and all threads that have requested forum f by incrementing $\text{wait}[f]$ are enabled by the assignment $\text{grant}[f] := \text{wait}[f]$ and can therefore enter the CS after thread p has done this assignment. P5 follows from the fact that once the leader grants permission to all the threads currently waiting, no late thread can enter the forum. As explained in Section 2, P8 follows from P4 and P5. In the algorithm A2, a leader of a forum f first makes its request by joining the queue (i.e., setting $\text{session} := F\&I(\text{tail})$) and then enters the wait section. Since the queue is served in FIFO order, any request for a different forum initiated after this will be serviced only after the current requests for f is completed. Therefore, the property P9 is satisfied.

The RMR complexity of A2 is still $O(m)$. The number of remote memory references (RMRs) is determined by the **await** statements. The entry and exit sections of A2 have no loops and only two **await** statements. For each forum, the leader waits at the second **await** statement; all other threads wait at the first **await** statement. During the entry protocol of any thread, the value of $\text{grant}[fm]$ is modified precisely once: either by a different thread or by the thread itself. Therefore, the first **await** statement takes at most two memory references. In the second **await** statement, however, the value of head may change up to the maximum of $m - 1$ times before it gets the value $\text{session}.p$ and hence $O(m)$ RMR complexity. Next, we look at how to modify the algorithm A2 to obtain constant RMR.

3.2 The Fair Group Mutual Exclusion with Constant RMR

The main reason that the algorithm A2 has RMR complexity $O(m)$ is that all the leaders of the forums await on a shared variable whose value can change several times. To avoid this, we need to make the leaders await on separate shared variables that can change only a constant number of times. To achieve this goal, we use the idea similar to the one used by Anderson (1990) for mutual exclusion. Instead of waiting for $\text{head} = \text{session}$, each leader waits at a Boolean bit. We introduce a Boolean array signal of size L , where L is a constant greater than the number of forums that could be requested at any time (i.e., $L > \min(n, m)$). Here, the last thread from the previous session releases the next leader from the waiting queue by setting its signal bit true. The resulting algorithm (A3) that assures constant RMR is presented next.

The changes from the algorithm A2 to algorithm A3 are highlighted by commenting only those statements. Initially, $\text{signal}[0] = \text{true}$ to enable the first session to be opened without any waiting. Other signal bits are initialized to *false* (i.e., for $0 < i < L$, $\text{signal}[i] = \text{false}$). Then, the last thread of every session will enable the next session by setting the next signal bit in the queue *true*. The signal queue is circular, meaning that its index ranges from 0 to $L - 1$, and the position 0 is next to the position $L - 1$.

In the algorithm A3, the value of $\text{signal}[\text{session} \bmod L]$ changes at most once during the entry protocol. Hence, the number of remote memory references for this **await** statement is reduced to two. Thus, the algorithm A3 has constant RMR.

ALGORITHM 3: Fair group mutual exclusion with constant RMR (A3)**shared variables**

```

head, tail : int ;
wait, grant, done : [forum  $\rightarrow$  int] ;
signal : [below(L)  $\rightarrow$  bool].

```

[Initially, head = tail = 0 $\wedge$ ($\forall f$: done[f] = grant[f] = wait[f] = 0) $\wedge$ signal = true];

Thread(p) :

```

var fm : forum, ticket, session, last : int ;
NCS ; choose fm  $\in$  forum ;
ticket := F&I(wait[fm]) ;
await ticket  $\leq$  grant[fm] ;
if ticket = grant[fm] then
    session := F&I(tail) ;
    await signal[session mod L] ;           // wait for the turn (signal bit to be true)
    signal[session mod L] := false ;      // reset signal bit
    grant[fm] := wait[fm] ;
endif
CS ;
last := grant[fm] - 1 ;
if last = F&I(done[fm]) then
    head++ ;
    signal[head mod L] := true ;          // give turn to next in the queue
endif.

```

3.3 Algorithm A2 as a Transition System

For the sake of the verification, algorithm A2 is transformed into a transition system, given next as Algorithm A4.

In this transition system, the code is separated into atomic commands indicated by line numbers. As argued above, the reading of wait[fm] and the writing of grant[fm] in the lines 26 and 27 have to be separated into two atomic statements; a private variable *newgrant* is introduced to store the value read. In the lines 31–32, a private variable *newhead* is used for a similar reason. Line 33 has a backward jump to the NCS to model that threads can repeatedly request for the CS. Line 27 is extended with an assignment to a shared variable curfm, which does not occur elsewhere. This is a so-called ghost variable, superfluous for the algorithm but useful for the proof. It holds the value of the current forum. Next we prove the correctness of the algorithms. We use assertional proof methods, where the derivations are based on the program's state instead of its behavior. It allows the proofs to be reduced to a series of invariants that can be verified by machine.

4 SAFETY

In this section, we prove the safety property of the algorithms, which is the *mutual exclusion* property (P1). We use the notation $v.p$ to refer to the value of the private variable v of thread p . Here and henceforth, we implicitly use universal quantification over all free variables (here q). For simplicity, we restrict the attention to algorithm A2.

ALGORITHM 4: Transition system for algorithm A2

shared variables

head, tail : *int* ;
 wait, grant, done : [*forum* $\rightarrow$ *int*].

Thread(*p*) :

```

var fm : forum, ticket, session, last, newgrant, newhead : int ;
20  NCS ; choose fm  $\in$  forum ;
21  ticket := F&I(wait[fm]) ;
22  await ticket  $\leq$  grant[fm] ;
23  if ticket = grant[fm] then
24    session := F&I(tail) ;
25    await head = session ;
26    newgrant := wait[fm] ;
27    grant[fm] := newgrant ; curfm := fm ;
    endif
28  CS ;
29  last := grant[fm] - 1 ;
30  if last = F&I(done[fm]) then
31    newhead := head + 1 ;
32    head := newhead ;
    endif
33  goto 20.

```

The proof of mutual exclusion for a GME algorithm is complicated. There are several reasons for this. For example, a new session of some forum is opened by any thread that executes line 27. This must not happen when there are threads in the region 28–32. Inequalities between *ticket* and grant[*fm*] are used to prove that it does not happen.

The test in line 30 decides whether or not to initiate a new forum. This test must therefore ensure that all threads of the current session have left. We therefore need to prove that the difference grant[*f*] – done[*f*] equals the number of threads that participate in the current session of forum *f*.

We provide our correctness proof through proving a set of invariants. Indeed, the proof of mutual exclusion lists around 20 invariants of the algorithm. These invariants are obtained by beginning with the main invariant *MX*, given below.

MX: *q* at 28 $\Rightarrow$ *fm.q* = curfm.

For every predicate that must be proved, there are two possibilities: either to infer it logically from other predicates (which must then be proved) or to prove that it holds initially and is preserved under every step of the algorithm. A proof that some predicate is preserved under every step of the algorithm requires a case distinction over all steps. The proof assistant PVS is used for these purposes. Proof assistants are much better in exhaustive case distinctions than humans. Another advantage of using a proof assistant is that after a modification of the algorithm the mechanical proof of the previous version is usually a good starting point for a proof of the new version.

Algorithm 4 serves as a blueprint for the transition system used in the PVS proof (Hesselink 2015b). During the development of such a mechanized proof, it often happens that line numbers

or invariants need to change. This is done with query-replace. To avoid the introduction of errors in this way, the line numbers are two-digit numbers and the invariants have names containing a letter “q” followed by a single digit. Indeed, late in the day, we discovered that we had to separate the lines 26 and 27. This forced us to introduce a new line number and some new invariants.

4.1 Correctness Proof of Safety by Top-down Approach

In the following, we derive and prove the invariants for safety in a top-down fashion: To prove preservation of one invariant, new invariants are invented and pushed on a queue to be proved later. That is, for safety, instead of proving that *MX* is preserved by every step, we prove that it is implied by other invariants. Indeed, *MX* follows from the invariants

Iq0: $q \text{ in } \{22 \dots 32\} \wedge \text{ticket}.q < \text{grant}[fm.q] \Rightarrow fm.q = \text{curfm},$
Iq1: $q \text{ in } \{28 \dots 32\} \Rightarrow \text{ticket}.q < \text{grant}[fm.q].$

For most invariants, we prove that they are preserved in every step of the algorithm; these constituent invariants have names that end with a “q” followed by a digit. For other invariants, we prove that they are implied by the constituent invariants. The choice is a matter of convenience. Of course, during the design, we sometimes have to backtrack, because a postulated predicate turns out not to be invariant. Such dead ends have been eliminated here. The reader who does not follow the arguments could inspect the mechanical proof, see Hesselink (2015b).

The following recital follows the PVS proof script closely. Preservation of predicate *Iq0* is proved by first investigating which steps of the algorithm can falsify it. The only steps that can do this are the steps 21 and 27. This is expressed by writing that *Iq0* is *threatened only* by the steps 21 and 27. It is threatened by step 21 only when thread *q* does step 21, sets $\text{ticket}.q := \text{wait}[fm.q]$, and goes to line 22. It is preserved because of the new postulate

Iq2: $\text{grant}[f] \leq \text{wait}[f].$

Iq0 is threatened by the assignment to *curfm* in step 27 of a thread *p* when $fm.q \neq fm.p$ and *q* is in 22–32 and $\text{ticket}.q < \text{grant}[fm.q]$. It is preserved by this step because of

Iq3: $q \text{ in } \{22 \dots 32\} \wedge r \text{ in } \{26 \dots 27\} \Rightarrow \text{grant}[fm.q] \leq \text{ticket}.q.$

Note that we use the variable *p* for the (arbitrary) acting thread, and *q*, *r*, *f* for the (arbitrary) free variables in the invariants.

Predicate *Iq1* is threatened only by the steps 23 and 27. It is threatened by step 23 of *q* when the guard of line 23 is false and *q* goes to line 28. In this case, predicate *Iq1* is preserved because of

Iq4: $q \text{ at } 23 \Rightarrow \text{ticket}.q \leq \text{grant}[fm.q].$

It is preserved when thread *q* does step 27 and goes to line 28 because of

Iq5: $q \text{ at } 27 \Rightarrow \text{ticket}.q < \text{newgrant}.q \leq \text{wait}[fm.q].$

It is preserved when thread *p* $\neq$ *q* does step 27 and modifies $\text{grant}[fm.p]$ because of

Amx0: $q \text{ in } \{26 \dots 32\} \wedge r \text{ in } \{26 \dots 27\} \Rightarrow q = r.$

The name *Amx0* is chosen to suggest “additional mutual exclusion.”

Predicate *Iq2* is threatened only by step 27. It is preserved because of *Iq5*.

Predicate *Iq3* is threatened only by the steps 21, 25, 27. It is preserved by step 21 because of *Iq2*. It is preserved by step 25 because of the new predicate

Iq6: $q \text{ in } \{22 \dots 32\} \wedge r \text{ at } 25 \wedge \text{head} = \text{session}.r \Rightarrow \text{grant}[fm.q] \leq \text{ticket}.q.$

Indeed, when thread r does step 25, this threatens $Iq3$ because r goes to line 26. If q is in 22–32, then $Iq6$ and the guard of step 25 imply the consequent of $Iq3$. Therefore, step 25 preserves $Iq3$. Predicate $Iq3$ is threatened by step 27 of p when there is a thread $r \neq p$ in 26 and 27. This is excluded by $Amx0$.

Predicate $Iq4$ is threatened only by step 27. It is preserved by step 27 because of $Iq5$ and the new postulate

$Iq7: \quad q \text{ in } \{24 \dots 27\} \Rightarrow \text{ticket}.q = \text{grant}[fm.q].$

Indeed, when thread p does step 27, this threatens $Iq4$ by modifying $\text{grant}[fm.p]$, but $Iq5$ and $Iq7$ together imply that $\text{grant}[fm.p]$ increases.

Predicate $Iq5$ is threatened only by step 26. It is preserved because of

$Iq8: \quad q \text{ in } \{22 \dots 32\} \Rightarrow \text{ticket}.q < \text{wait}[fm.q].$

Predicate $Amx0$ is implied by $Iq1$, $Iq3$, and the new postulates

$Jq0: \quad q \text{ in } \{25 \dots 27\} \wedge r \text{ in } \{25 \dots 27\} \wedge \text{session}.q = \text{session}.r \Rightarrow q = r,$

$Jq1: \quad q \text{ in } \{26 \dots 27\} \Rightarrow \text{session}.q = \text{head}.$

Indeed, let thread r be in 26 and 27. By $Iq1$ and $Iq3$, thread q is not in 28–32. By $Jq0$ and $Jq1$, thread q can only be in 26 and 27 for $q = r$. This proves $Amx0$.

Predicate $Iq6$ is threatened only by the steps 21, 24, 27, and 32. It is threatened by step 21 of thread q when it goes to line 22; it is preserved because of $Iq2$. It is threatened by step 24 of thread r when it goes to line 25. It is preserved because of

$Jq2: \quad q \text{ in } \{22 \dots 32\} \wedge \text{ticket}.q < \text{grant}[fm.q] \Rightarrow \text{head} < \text{tail}.$

It is threatened in step 27 of thread p by its modification of grant . It is preserved because of the new postulate

$Jq3: \quad q \text{ in } \{26 \dots 32\} \wedge r \text{ at } 25 \Rightarrow \text{head} < \text{session}.r.$

It is threatened by modification of head in step 32. It is preserved because of

$Jq4: \quad q \text{ in } \{22 \dots 32\} \wedge r \text{ in } \{31 \dots 32\} \wedge \text{ticket}.q < \text{grant}[fm.q] \Rightarrow q = r.$

Predicate $Iq7$ is threatened only by step 27. It is preserved because of

$Jq5: \quad q \text{ in } \{22 \dots 32\} \wedge r \text{ in } \{22 \dots 32\} \wedge fm.q = fm.r \wedge \text{ticket}.q = \text{ticket}.r$
 $\Rightarrow q = r.$

Predicate $Iq8$ is inductive (not threatened by any step).

Predicate $Jq0$ is threatened only by step 24 of q or r . It is preserved because of

$Jq6: \quad q \text{ in } \{25 \dots 27\} \Rightarrow \text{head} \leq \text{session}.q < \text{tail}.$

Predicate $Jq1$ is threatened only by the modification of head in step 32. It is preserved because of $Amx0$.

Predicate $Jq2$ is threatened only by the steps 21, 27, and 32. It is preserved by step 21 because of $Iq2$. It is preserved by step 27 because of

$Jq7: \quad q \text{ in } \{26 \dots 32\} \Rightarrow \text{head} < \text{tail}.$

It is preserved by step 32 because of $Jq4$.

Predicate $Jq3$ is threatened only by steps 23, 24, 25, and 32. It is threatened when thread q executes 23 and goes to line 28. It is preserved because of $Iq4$, $Iq6$, and $Jq6$. Indeed, $Iq4$ together with

the guard of step 23 gives $ticket.q < grant[fm.q]$. Therefore, $Iq6$ and $Jq6$ together imply $head < session.r$.

Predicate $Jq3$ is threatened by step 24 of r and preserved because of $Jq7$. It is threatened by step 25 of q and preserved because of $Jq0$ and $Jq6$. It is preserved by step 32 because of the second additional mutual exclusion predicate

$$Amx1: \quad q \text{ in } \{26 \dots 32\} \wedge r \text{ in } \{31 \dots 32\} \Rightarrow q = r.$$

Predicate $Jq4$ is threatened only by the steps 21, 27, and 30. It is preserved by step 21 because of $Iq2$. It is preserved by step 27 because of $Amx1$. It is preserved by step 30 because of the “last-done” predicate

$$LaD: \quad q \text{ in } \{22 \dots 32\} \wedge ticket.q < grant[fm.q] \wedge r \text{ at } 30 \wedge last.r = done[fm.r] \\ \Rightarrow q = r.$$

Predicate $Jq5$ is threatened only by step 21. It is preserved because of $Iq8$.

Predicate $Jq6$ is threatened only by steps 24 and 32. It is preserved by step 24 because of

$$Jq8: \quad head \leq tail.$$

It is preserved by step 32 because of $Amx0$ and

$$Jq9: \quad q \text{ at } 32 \Rightarrow newhead.q = head + 1.$$

Predicate $Jq7$ is threatened only by the steps 23, 25, and 32. It is preserved by step 23 because of $Iq4$ and $Jq2$. It is preserved by step 25 because of $Jq6$. It is preserved by step 32 because of $Amx1$.

Predicate $Amx1$ is implied by $Iq1$, $Jq4$, and $Amx0$.

Before treating predicate LaD , we deal with $Jq8$ and $Jq9$. Preservation of $Jq8$ is threatened only by step 32. It is preserved because of $Jq7$ and $Jq9$. Preservation of $Jq9$ is threatened only by step 32. It is preserved because of $Amx1$.

In order to prove LaD , we first postulate the invariant

$$Kq0: \quad q \text{ at } 30 \Rightarrow last.q = grant[fm.q] - 1.$$

Predicate $Kq0$ is threatened only by step 27. It is preserved because of $Amx0$.

We now define, for every forum f , the sets of threads

$$group(f) = \{q \mid q \text{ in } \{22 \dots 30\} \wedge fm.q = f\}, \\ class(f) = \{q \in group(f) \mid ticket.q < grant[f]\}.$$

Initially, these sets are empty. The sets remain finite because in every step of the algorithm at most one thread enters $group(f)$. Their cardinalities are determined by

$$Kq1A: \quad \#group(f) = wait[f] - done[f], \\ Kq1B: \quad \#class(f) = grant[f] - done[f].$$

These predicates are proved by generalizing them. For any integer z , we define the finite set

$$class(f, z) = \{q \in group(f) \mid ticket.q < z\},$$

and we postulate the invariant

$$Kq1: \quad grant[f] \leq z \leq wait[f] \Rightarrow \#class(f, z) = z - done[f].$$

The predicates $Kq1A$ and $Kq1B$ follow from $Kq1$ by $Iq2$ and $Iq8$.

Predicate $Kq1$ is threatened only by steps 21, 27, 30. It is preserved by step 21 because of $Iq2$, $Iq8$, and $Kq1$ with $z := z - 1$. It is preserved by step 27 because of $Iq5$ and $Iq7$. It is preserved by step 30 because of $Iq1$.

Predicate LaD follows from the invariants obtained up to this point. Indeed, when r is at 30 and $last.r = done[fm.r]$, the invariants $Kq1B$ and $Kq0$ imply that the set $class(fm.r)$ has precisely one element. By $Iq1$, this element is thread r . It follows that $fm.r = curfm$ by $Iq0$.

Now, if q is in $\{22 \dots 30\}$ and $ticket.q < grant[fm.q]$, then $fm.q = curfm$ by $Iq0$, and hence $q \in class(curfm) = \{r\}$. It remains to consider the case that q is in 31 and 32, while r is at 30. This case is forbidden by $Amx1$. This concludes the proof of LaD , and thus of mutual exclusion property of $A2$. The verification of algorithm $A3$ is similar but slightly more complicated. The verification with the proof assistant PVS in Hesselink (2015b) covers both algorithms. Next we will prove the liveness property, which has several variants.

5 LIVENESS

In this section, we prove that algorithm $A2$ satisfies the progress properties $P2$ and $P3$. $P2$ is about throughput: while there are competing threads, the number of completed sessions eventually grows. $P3$ is about individual progress: when a thread is competing, it eventually becomes idle again. Liveness and individual progress are usually proved by showing the absence of deadlock and lockout. Such proofs, however, give no insight on the speed of progress. We, therefore, prefer to give constructive proofs supported by an estimate of the time complexity.

In both $P2$ and $P3$, the keyword is “eventually.” In either case, eventually is quantified in the following way. Let an *execution fragment* be defined as a nonempty finite sequence of steps of the algorithm. Two execution fragments can be concatenated if the final state of the first one is the first state of the second one. Let an execution fragment be called a *round* if, for every thread q , the fragment contains a forward step of q or a state in which thread q is disabled. In other words, it is a round iff every thread is considered in it for scheduling.

If P and Q are predicates on the state, P leads to Q within k rounds, then notation:

$$P \text{ Lt } \langle k \rangle Q$$

is defined to mean that every execution fragment that contains a concatenation of k rounds and that begins with a state in P , contains a state in Q .

To prove liveness properties, we develop a formal operation model and use it along with UNITY developed in Chandy and Misra (1988) and Misra (2001), and a quantified version of it. This quantified version is used to prove Theorem 5.1 in Section 5.3 and Theorem 5.3 in Section 5.5. In Section 5.1, we develop the formal operational model. In Section 5.2, we present UNITY, and the quantified version of it.

5.1 Formal Operational Semantics

The *state* of the system is given by the values of all shared and private variables. Up to this point, we have kept the state implicit, but formally all invariants are Boolean functions of the state. We let X be the set of all states. If P is a predicate (on the state), then it can also be regarded as the subset of X where predicate P holds. We can therefore write $P \subseteq Q$ to mean that every state that satisfies P also satisfies Q (i.e., that P implies Q). Let *start* be the initial predicate, i.e., the set of initial states.

For thread p , relation $step(p)$ is defined as the set of the pairs (x, y) of states such that in state x thread p can do a step of the algorithm that results in state y . Relation $step$ is defined as the union of the relations $step(p)$ for all threads p , together with the identity relation of the state space. An *execution* is defined to be an infinite sequence xs of states with $xs_0 \in start$, and $(xs_n, xs_{n+1}) \in step$

for all $n \in \mathbb{N}$. A predicate P is an *invariant* if and only if it contains all states of all executions. We write $X_0 \subseteq X$ for the intersection of all invariants obtained above.

An execution fragment is a nonempty finite sequence $(xs_0 \dots xs_n)$ in X_0 such that $(xs_i, xs_{i+1}) \in \text{step}$ for all i with $0 \leq i < n$. Two execution fragments can be concatenated when the final state of the first fragment equals the initial state of the second fragment.

Coming back to the algorithm, the *forward steps* are defined to be the steps 21–35. Relation $\text{fwd}(p) \subseteq \text{step}(p)$ is defined to be the set of forward steps of thread p . Thread p is therefore *enabled* in state x if and only if there is a state y with $(x, y) \in \text{fwd}(p)$. Recall that enabledness is expressed by the predicate $\text{ena}(p)$.

An *occurrence* of thread p in an execution fragment $(xs_0 \dots xs_n)$ is a number i with $0 \leq i < n$, and $(xs_i, xs_{i+1}) \in \text{fwd}(p)$ or $xs_i \notin \text{ena}(p)$. The execution fragment is called a *round* if it contains an occurrence of every thread. In other words, in the fragment, every thread is scheduled, and either executed or found to be disabled. This applies, e.g., when thread p is always at line 20.

Progress of the algorithm will be proved under the assumption that all threads do enough forward steps unless they are disabled. To put it more precisely, progress will be proved for any execution fragment that contains a concatenation of sufficiently many rounds.

5.2 UNITY and Bounded UNITY

UNITY logic was introduced by Chandy and Misra (1988). This makes it possible to prove progress in the form of leads-to relations by assertional means, i.e., solely by investigating the states of the system, the step relation, and the forward steps of the algorithm. No executions or execution fragments need to be considered for the specific algorithm. UNITY itself only has a qualitative leads-to relation. The quantitative relation $\text{Lt} \langle k \rangle$ used above is introduced in Hesselink (2015a).

UNITY logic (Chandy and Misra 1988; Misra 2001) is focused on the concept of “leads-to” ($\mapsto$). The informal meaning of $P \mapsto Q$ is “if P holds at any point in the computation, Q will hold eventually” (here “will” applies to the current point as well as the future).

Example: Individual progress of the algorithm means that a thread, say, p , in the entry protocol, will eventually reach the critical section at line 28. This is expressed by: $p \text{ in } \{21 \dots 27\} \mapsto p \text{ at } 28$.

UNITY logic begins with defining two relations, **co** and **co!**, between predicates:

$$\begin{aligned} P \text{ co } Q &\equiv \forall (x, y) \in \text{step} : x \in P \Rightarrow y \in Q, \\ P \text{ co! } Q &\equiv \exists r : P \subseteq \text{ena}(r) \wedge (\forall (x, y) \in \text{fwd}(r) : x \in P \Rightarrow y \in Q). \end{aligned}$$

$P \text{ co } Q$ means that every step that starts in P ends in Q . According to **co!**, there is a specific thread r that is able to establish Q .

UNITY logic is based on the relations **unless** and **en** (*ensures*) defined by:

$$\begin{aligned} P \text{ unless } Q &\equiv (P \wedge \neg Q \wedge X_0) \text{ co } (P \vee Q), \\ P \text{ ensures } Q &\equiv (P \text{ unless } Q) \wedge ((P \wedge \neg Q \wedge X_0) \text{ co! } Q). \end{aligned}$$

UNITY’s *leads-to* relation $\mapsto$ is defined inductively by the three rules:

- $P \text{ ensures } Q$ implies $P \mapsto Q$.
- Relation $\mapsto$ is transitive.
- For any family $(P_i)_{i \in I}$, if $P_i \mapsto Q$ for all $i \in I$, then $(\exists i \in I : P_i) \mapsto Q$.

Bounded UNITY (Hesselink 2015a) is a version of UNITY in which the leads-to relation is quantified by means of a natural number: *Pleads to Q within n rounds*, notation $P \text{ Lt } \langle n \rangle Q$, is defined to mean that every execution fragment that contains a concatenation of n rounds and has its initial state in P , contains a state in Q . The basic proof rules are

- If $P \subseteq Q$, then $P \text{ Lt } \langle n \rangle Q$ for every $n \geq 0$.
- $P \text{ ensures } Q$ implies $P \text{ Lt } \langle 1 \rangle Q$.
- If $P \text{ Lt } \langle k \rangle Q$ and $Q \text{ Lt } \langle m \rangle R$, then $P \text{ Lt } \langle k + m \rangle R$.
- For any family $(P_i)_{i \in I}$, if $P_i \text{ Lt } \langle n \rangle Q$ for all $i \in I$, then $(\exists i \in I : P_i) \text{ Lt } \langle n \rangle Q$.

The first rule is called the subset rule, the second one is the **ensures** rule, the third one is called transitivity, and the fourth one is called the Disjunction Rule.

There is also the Progress-Safety-Progress Rule (Chandy and Misra 1988):

$$\text{PSP:} \quad (P \text{ Lt } \langle n \rangle Q) \wedge (A \text{ unless } M) \Rightarrow (P \wedge A) \text{ Lt } \langle n \rangle ((Q \wedge A) \vee M).$$

We finally need a version of the Completion Rule (Chandy and Misra 1988).

$$\text{Com:} \quad (\forall i \in I : (P \text{ Lt } \langle n \rangle Q_i) \wedge (Q_i \text{ unless } R)) \Rightarrow P \text{ Lt } \langle n \rangle ((\forall i : Q_i) \vee R).$$

For UNITY itself, this rule only applies when the index set I is finite, but it is generally valid for bounded UNITY.

The soundness of these proof rules in the operational semantics of Section 5.1 has been proved mechanically (Hesselink 2015b). The set of proof rules is not complete, but they are enough for the present purposes.

5.3 Throughput

As every session of a forum ends with an incrementation of head, throughput can be measured by the growth of head. As idle threads need not be interested in the CS, tail need not grow. Therefore head cannot be expected to grow more than the current distance between head and tail because of $\mathcal{J}q8$. In the algorithm, head is the number of completed sessions and the difference $\text{tail} - \text{head}$ is the number of incomplete session requests. Throughput is therefore measured by the growth of head. It turns out that throughput is proportional to the number of rounds.

THEOREM 5.1. *For all numbers m and n , it holds that $(\text{head} = m \wedge m + n \leq \text{tail}) \text{ Lt } \langle 10 \cdot n \rangle (\text{head} = m + n)$.*

PROOF. To guide the case distinctions, we define the state-dependent sets of threads

$$\begin{aligned} \text{Ready} &= \text{Signal} \cup \text{Open} \cup \text{Session} \cup \text{Exit}, \text{ where} \\ \text{Signal} &= \{q \mid q \text{ at } 25 \wedge \text{head} = \text{session}.q\}, \\ \text{Open} &= \{q \mid q \text{ in } \{26, 27\}\}, \\ \text{Session} &= \{q \mid q \text{ in } \{22 \dots 30\} \wedge \text{ticket}.q < \text{grant}[fm.q]\}, \\ \text{Exit} &= \{q \mid q \text{ in } \{31, 32\}\}. \end{aligned}$$

As we are aiming at a proof that the value of head grows, we first observe that, by $\mathcal{J}q9$, we have

$$q \text{ at } 32 \wedge \text{head} = m \text{ ensures } \text{head} = m + 1.$$

It is easy to see that

$$q \text{ at } 31 \wedge \text{head} = m \text{ ensures } q \text{ at } 32 \wedge \text{head} = m.$$

By the **ensures** rule together with transitivity of **Lt**, these two **ensures** properties together imply that, when q is in *Exit*, head will be incremented within two rounds:

$$q \in \text{Exit} \wedge \text{head} = m \text{ Lt } \langle 2 \rangle \text{head} = m + 1.$$

By the Disjunction Rule, it follows that

$$(1) \quad \text{Exit} \neq \emptyset \wedge \text{head} = m \text{ Lt } \langle 2 \rangle \text{head} = m + 1.$$

The next point is to show that every thread leaves *Session*. This is shown by means of the variant function vf given by

$$vf(q) = (q \notin \text{Session} ? 0 \\ : pc.q \leq 27 ? 27 - pc.q \\ : 31 - pc.q).$$

Here, we use that, when $q \in \text{Session}$ holds, it cannot be in 24–27 because of $Iq7$. More precisely, using the invariants $Iq5$ and $Iq7$, we prove that, for all $n \in \mathbb{N}$,

$$vf(q) = n + 1 \text{ ensures } vf(q) = n.$$

By the **ensures** rule, transitivity, disjunction, and $Iq7$, it follows that thread q leaves *Session* within 5 rounds:

$$(2) \quad \text{true} \text{ Lt } \langle 5 \rangle q \notin \text{Session}.$$

However, it follows from $Iq2$ and $Iq3$ that, when *Session* is nonempty, no thread enters *Session*. This is formalized in

$$(3) \quad q \notin \text{Session} \text{ unless } \text{Session} = \emptyset.$$

The Completion Rule applied to (2) and (3) implies that every session ends:

$$(4) \quad \text{true} \text{ Lt } \langle 5 \rangle \text{Session} = \emptyset.$$

Looking more carefully at session closure, we see that head is not modified as long as *Session* is nonempty and that it should end with a thread at line 31. Indeed, we have

$$(5) \quad \text{Session} \neq \emptyset \wedge \text{head} = m \text{ unless } \text{Exit} \neq \emptyset \wedge \text{head} = m.$$

This is proved using $Kq0$, $Kq1B$, $Iq1$, $Jq4$ for step 32, and $Iq5$ and $Iq7$ for step 30.

Application of the PSP rule with (4) and (5) gives

$$(6) \quad \text{Session} \neq \emptyset \wedge \text{head} = m \text{ Lt } \langle 5 \rangle \text{Exit} \neq \emptyset \wedge \text{head} = m.$$

It is easy to see that opening of a session only needs two rounds:

$$(7) \quad \text{Open} \neq \emptyset \wedge \text{head} = m \text{ Lt } \langle 2 \rangle \text{Session} \neq \emptyset \wedge \text{head} = m.$$

Here $Amx0$ is used to show that head is not modified.

Finally, we have the **ensures** property

$$q \in \text{Signal} \wedge \text{head} = m \text{ ensures } q \in \text{Open} \wedge \text{head} = m.$$

The proof uses the invariant $Jq3$ to show that step 32 does not modify head. By the Disjunction Rule, this formula implies

$$(8) \quad \text{Signal} \neq \emptyset \wedge \text{head} = m \text{ Lt } \langle 1 \rangle \text{Open} \neq \emptyset \wedge \text{head} = m.$$

Combining the formulas (1), (6), (7), and (8) with transitivity, we obtain

$$(9) \quad \text{Ready} \neq \emptyset \wedge \text{head} = m \text{ Lt } \langle 10 \rangle \text{head} = m + 1.$$

The invariant $Lq1$ implies that

$$\text{head} < \text{tail} \Rightarrow \text{Ready} \neq \emptyset.$$

We can therefore conclude to

(10) $\text{head} = m < \text{tail} \quad \mathbf{Lt} \langle 10 \rangle \text{head} = m + 1.$

Now the theorem follows by induction, using transitivity of $\mathbf{Lt}$. $\square$

Note that the factor, 10, does not depend on the number of concurrently competing threads or forums.

5.4 Absence of Immediate Deadlock

Immediate deadlock is a state where there are competing threads but none of them is enabled. Absence of immediate deadlock is a first step towards the verification of liveness. When thread is in the noncritical section NCS , it need not make any progress, nor contribute to the progress of the other threads. We therefore distinguish: A thread is called *idle* when it is in the NCS (i.e., at line 20); it is called *competing* when it is not idle. A step of the algorithm is called a *forward* step when it is done by a competing thread. A thread is said to be *disabled* when it cannot do a forward step, either because it is idle or because it is at an **await** statement and the guard is false. It is easy to see that the negation *enabledness* is characterized by

$$\begin{aligned} \text{ena}(q) \quad \equiv \quad & q \text{ in } \{21 \dots 33\} \wedge (q \text{ at } 22 \Rightarrow \text{ticket}.q \leq \text{grant}[fm.q]) \\ & \wedge (q \text{ at } 25 \Rightarrow \text{head} = \text{session}.q). \end{aligned}$$

In order to prove that immediate deadlock does not occur, we postulate two additional invariants. We obviously have

$$Lq0: \quad q \text{ in } \{20 \dots 33\}.$$

The second invariant needed expresses that all session slots between head and tail are in use:

$$\begin{aligned} Lq1: \quad & \text{head} \leq z < \text{tail} \\ & \Rightarrow (\exists q : (q \text{ at } 25 \wedge \text{session}.q = z) \\ & \quad \vee (q \text{ in } \{26, 27\} \wedge \text{head} = z) \\ & \quad \vee (q \text{ in } \{22 \dots 32\} \wedge \text{head} = z \wedge \text{ticket}.q < \text{grant}[fm.q])). \end{aligned}$$

Predicate $Lq1$ is threatened only by the steps 27, 30, and 32. It is preserved at 27 because of $Iq5$ and $Iq7$. It is preserved at 30 because of $Kq0$, $Kq1B$, and $Iq2$. Indeed, $Lq1$ is threatened by step 30 when a witness q of its right-hand side jumps from 30 to 33. In this case, we have $\text{last}.q \neq \text{done}[fm.q]$ because of the guard of step 30. By $Kq0$ and $Kq1B$, it follows that $\#class(fm.q) \neq 1$. As $q \in class(fm.q)$ by $Lq1$, this implies $r \in class(fm.q)$ for some thread $r \neq q$. Thread r remains a witness of the right-hand side of $Lq1$. Predicate $Lq1$ is preserved at 32 because of $Iq9$.

Furthermore, to preclude deadlock at line 22, we postulate that every number that is at least $\text{grant}[f]$ and below $\text{wait}[f]$ is used as a ticket:

$$Kq1C: \quad \text{grant}[f] \leq z < \text{wait}[f] \Rightarrow \exists q : \text{ticket}.q = z \wedge fm.q = f \wedge q \text{ in } \{22 \dots 27\}.$$

This predicate is proved as follows. Using $Kq1$ for both z and $z + 1$, we see that $\#class(f, z) < \#class(f, z + 1)$. It follows that there is a thread $q \in class(f, z + 1)$ with $q \notin class(f, z)$. This thread q is a witness of the right-hand side of $Kq1C$ because of $Iq1$.

Absence of immediate deadlock is expressed by

LEMMA 5.2. *Assume that all threads are not enabled. Then all threads are idle.*

PROOF. As all threads are not enabled, it follows from $Lq0$ that every thread is at one of the lines 20, 22, or 25. Moreover, any thread q at line 22 satisfies $\text{grant}[fm.q] < \text{ticket}.q$, and any thread at line 25 satisfies $\text{head} \neq \text{session}.q$.

If there is a thread q at line 25, then $Jq6$ implies $\text{head} < \text{tail}$. Then $Lq1$ with $z = \text{head}$ implies that there is an enabled thread. Therefore, all threads are at the lines 20 or 22.

If there is a thread q at line 22, then $Iq8$ implies $\text{grant}[f] < \text{wait}[f]$ for $f = fm.q$. Then $Kq1C$ with $z = \text{grant}[f]$ implies there is a thread r at line 22 with $\text{ticket}.r = \text{grant}[f]$ and $fm.r = f$. This thread would be enabled. Therefore, there are no threads at line 22.

5.5 Individual Progress

We next turn to individual progress, also known as lockout freedom. If the number k of concurrently competing forums is large, then an entering thread that needs to attend a forum that is not yet requested, may have to wait a number of rounds proportional to k before it can be served. This implies, we need an upper upper bound K on the number of competing forums to quantify individual progress.

Asp 1: The number of concurrently competing forums has the upper bound K .

The speed of individual progress does depend on the upper bound K of the number of competing forums. With Assumption *Asp 1*, individual progress of a thread in Algorithm A2 is expressed by the following theorem.

THEOREM 5.3. $(q \text{ in } \{21 \dots 33\}) \text{ Lt } \langle U \rangle (q \text{ at } 20)$ holds for $U = 10 \cdot K + 23$.

PROOF. Complementary to the set *Session* introduced in the proof of Theorem 5.1, we define the waiting set

$$\text{Wat} = \{q \mid q \text{ in } \{22 \dots 27\} \wedge \text{grant}[fm.q] \leq \text{ticket}.q\}.$$

It follows from $Iq1$ that

$$(11) \quad q \text{ in } \{22 \dots 33\} \Rightarrow q \text{ at } 22 \vee q \in \text{Wat} \vee q \in \text{Session} \vee q \text{ in } \{31 \dots 33\}.$$

It will be shown that each of the four disjuncts at the right-hand side leads to the next one, while the last one leads to $q \text{ at } 20$.

By Theorem 5.1, the value of head grows. If thread q is and remains in the region 25–27, then the growth of head is bounded by $\text{session}.q$ because of $Jq6$. The PSP Rule can therefore be used to infer

$$(q \text{ in } \{25 \dots 27\} \wedge \text{session}.q = m \wedge \text{head} = m - n) \text{ Lt } \langle 10 \cdot n + 10 \rangle (q \text{ at } 28).$$

The free variable m can be eliminated with the Disjunction Rule. Lemma 6.1 together with $Jq6$ sets a bound on the difference between head and $\text{session}.q$. By $Iq1$, q at 28 implies $q \in \text{Session}$. This gives the Formula

$$q \text{ in } \{25 \dots 27\} \text{ Lt } \langle 10 \cdot K + 10 \rangle q \in \text{Session}.$$

Let *Leader* be the subset of *Wat* given by

$$\text{Leader} = \{q \mid q \text{ in } \{22 \dots 27\} \wedge \text{grant}[fm.q] = \text{ticket}.q\}.$$

A thread $q \in \text{Leader}$ can pass the guards of lines 22 and 23, and reach line 25 in three rounds. Therefore, the previous formula implies

$$(12) \quad q \in \text{Leader} \text{ Lt } \langle 10 \cdot K + 13 \rangle q \in \text{Session}.$$

For every thread $q \in \text{Wat}$, there is a thread $r \in \text{Leader}$ with $fm.r = fm.q$, because of $Iq8$ and $Kq1C$. In most cases, when thread r leaves the set *Leader*, thread q is enabled in the sense that $q \in \text{Session}$. The only exception is that thread r at 27 has $\text{newgrant}.r \leq \text{ticket}.q$. The exceptional case is resolved

in one round because thread r moves to line 28. In the exceptional case, the invariants $Kq1C$ with $z = \text{newgrant}.r$ and $Iq5$ are used to determine the leader after r . In this way, using the Formula (12), it is proved that

$$(13) \quad q \in \text{Wat} \quad \text{Lt} \langle 10 \cdot K + 14 \rangle \quad q \in \text{Session}.$$

The remaining steps are easier. It is clear that

$$(14) \quad q \text{ at } 21 \quad \text{Lt} \langle 1 \rangle \quad q \in \text{Wat}.$$

It holds that $q \in \text{Session}$ **unless** $q \text{ in } \{31 \dots 33\}$, because of $Iq5$. By Formula (2) and the PSP rule, it follows that

$$(15) \quad q \in \text{Session} \quad \text{Lt} \langle 5 \rangle \quad q \text{ in } \{31 \dots 33\}.$$

It is easy to see that

$$(16) \quad q \text{ in } \{31 \dots 33\} \quad \text{Lt} \langle 3 \rangle \quad q \text{ at } 20.$$

Theorem 5.3 follows by combining the formulas (11), (13), (14), (15), and (16) with transitivity.

6 BOUNDING INTEGERS BY WRAP-AROUND

The algorithms A2 and A3 are developed for mathematical (i.e., unbounded) integers head, tail, wait, grant, and done. Under mild assumptions, however, they can be implemented with ordinary bounded integers in two's complement representation where the smallest machine integer is the successor of the largest machine integer. Let M be the number of available machine integers, typically $M = 2^{32}$ or $M = 2^{64}$.

The variables head, tail, session are represented by unsigned integers u , with $0 \leq u < M$. The only requirement here is that L , the size of array signal, divides M . This is required, because we need $(z \bmod M) \bmod L = z \bmod L$, for every mathematical integer z . If M is a power of 2, then L should also be a power of 2.

The variables wait, grant, done, ticket need to be represented by signed integers u , with $-\frac{1}{2}M \leq u < \frac{1}{2}M$. This is required because of the inequality test in line 22. As indicated in Belloch et al. (2003), this inequality needs to be replaced by the (mathematically equivalent) test $\text{ticket} - \text{grant}[fm] \leq 0$. This has the effect that the value of $\text{grant}[fm]$ is shifted to the middle of the range of the signed integers. Based on the assumption Asp 1 (introduced in Section 5.5), we establish a relation among head, tail, and K .

LEMMA 6.1. $\text{tail} \leq \text{head} + K + 1$.

PROOF. We may assume that $\text{head} < \text{tail}$. Let HL be the set of the integers z with $\text{head} < z < \text{tail}$. Then $\#HL = \text{tail} - \text{head} - 1$. Let $W25$ be the set of the threads q at line 25 with $\text{session}.q \neq \text{head}$. By $Jq6$, the private variable session induces a function from $W25$ to HL , which is a bijection because of $Jq0$ and $Lq1$. Therefore, $\#W25 = \#HL$. Let CF be the set of the competing forums. Then $\#CF \leq K$ by Assumption 1. The private variable fm induces a function from $W25$ to CF , which is injective because of $Iq7$ and $Jq5$. This proves that $\text{tail} - \text{head} - 1 = \#HL = \#W25 \leq \#CF \leq K$. Hence the proof.

Asp 2: The number of threads competing for forum f has the upper bound $\frac{1}{2}M$.

When $M \geq 2^{32}$, this is a reasonable assumption and it is required for the following lemmas.

LEMMA 6.2. $\text{done}[f] \leq \text{grant}[f] < \text{done}[f] + M$.

PROOF. By Assumption *Asp 1*, $\#group(f) \leq \frac{1}{2}M$. By *Kq1A*, *Kq1B*, and *Iq2*, this implies $done[f] \leq grant[f] \leq wait[f] \leq done[f] + \frac{1}{2}M$

LEMMA 6.3. $grant[fm] - \frac{1}{2}M \leq ticket < grant[fm] + \frac{1}{2}M$.

PROOF. This assertion follows from the additional invariant

Kq2: $q \in group(f) \Rightarrow grant[f] - \frac{1}{2}M \leq ticket.q$.

Predicate *Kq2* is threatened only by the steps 21 and 27. It is preserved by step 21 because of *Iq2* and by step 27 because of *Iq3*, *Iq5*, and Lemma 6.2.

Note that the variables in these formulas are the unbounded mathematical variables, not their machine representations. From lemmas 6.1, 6.2, and 6.3, Algorithms A2 and A3 will preserve the correctness with the two's complement representation, where the smallest machine integer is the successor of the largest machine integer and if the statement **await** $ticket \leq grant[fm]$ is replaced by **await** $grant[fm] - ticket \geq 0$.

7 PERFORMANCE STUDY

The main appeal of our algorithms is that they are simple, assure high fairness, and the second algorithm incurs only constant RMR—an asymptotically attractive property. From the performance perspective, we do not expect a huge difference from the performance of the scalable room synchronization algorithm of Blelloch et al., particularly for small number of forums and threads. We conducted two sets of simple experimental study using two forums and four forums comparing our algorithms (Algorithm A2 referred to as FGME and Algorithm A3 referred to as FGME-CRMR) with the algorithm BCG given in Blelloch et al. (2003). The reasons for comparing FGME and FGME-CRMR with BCG are that (i) in Blelloch et al. (2003), BCG was already compared with one of the well-known algorithms by Keane and Moir, and (ii) FGME, FGME-CRMR, and BCG satisfy the same concurrency properties. In each study, we observed the number of CS entries and average wait time for each entry.

We conducted experiments on Dell Poweredge FC630 with Intel Xeon Processor E5-2650 (2 CPU with 10 cores, each core can schedule two threads at a time). Maximum 16 cores were used to assign threads (we observed fluctuation in results when we crossed this limit). For our performance study, a test harness was used to create pthread (as worker threads) and assigned to the cores. For both studies, we varied the number of threads from *forum* to maximum 32 (i.e., 2 to 32 for the two-forum case and 4 to 32 for the four-forums case). Each study consists of a number of experiment (30 in the case of two forums and 28 in the case of four forums) by varying the number of threads between *forum* and 32. It supports exchange based *T&S* (*_sync_lock_test_and_set*) and *F&A* (*_sync_fetch_and_add*) atomic operations.

As indicated above, we conducted two studies: one to observe the number of forum entries and the other to observe the average delay per forum entry. The number of forum entries for two forums and four forums, respectively, are displayed in Figure 1 and Figure 2.

The performance graphs, given in Figure 1 and Figure 2, show a similar trend that the number of CS entries decreases as the contention for CS increases. The performance of BCG, FGME, and FGME-CRMR, respectively, are represented using dark blue, purple, and red graphs. In both cases, as expected FGME and FGME-CRMR performs better than BCG. Again, as expected, the performance of FGME-CRMR is closer to or slightly better than the performance of FGME. When the average contention is low, one or two threads per forum, the number of CS entries is quite high. When the average contention is greater than 2, the number drops quickly and then decreases

Fig. 1. Number of CS entries: With two forums.

gradually. The decrease, we believe, is attributed to the overhead and extra wait due to increased contention for the CS.

We noticed some random fluctuations (slight up and down) in performance. Since the fluctuations are non-reproducible, we believe they could be due to random factors involved in the system. Overall, we observed the consistency in the trend that the performance of FGME and FGME-CRMR is above the performance of BCG in the number of CS entries. The graphs illustrating the average delay for two forums and four forums, respectively, are given in Figure 3 and Figure 4.

As we expected, the average delay per CS entry of BCG is higher than that of FGME and FGME-CRMR for both cases (two forums and four forums). Again, as expected, threads in FGME experience a slightly higher delay than that of FGME-CRMR.

Each experiment observes the average of five runs, and the duration of each run is 1s (T_r). In a run, each thread cycles between NCS and CS for the duration of T_r . For each run, every thread is modeled as follows. It starts from the entry section of its code. Then, it chooses its forum, enters the forum when allowed, does some work inside the forum, exits the forum, does some work outside the forum, and re-enters the entry section. This sequence is repeated for the duration of T_r , and at the end the run the results (total number of forum entries and average delay) for that run were computed. In the entry section, it chooses a forum randomly (the probability of choosing a forum is equally distributed). Inside the forum, it checks its forum id with the current active forum id (a self check for safety). After exiting the forum, it updates the performance parameters—entry count and delay.

Fig. 2. Number of CS entries: With four forums.

We also observed the concurrency (the number of threads in each forum) by recording from the difference between grant and wait values before granting the forum accesses to the followers. We also computed forum entries for each thread. Finally, the total number of CS entries are computed separately using concurrency and using the CS entries of each thread. These totals are then checked to see whether they tally with each other. This is another form of correctness check. We included these two checks—self-check and the check for computing forum entries—as a way to demonstrate the implementation correctness of the algorithms for our simulation studies.

8 COMPARISON SUMMARY

Since the GME algorithms presented in this article require a higher level atomic instruction *F&I*, we compare them only with the GME algorithms requiring higher-level atomic instructions presented in the literature (Bhatt and Huang 2010; Btlelloch et al. 2003; Danek and Hadzilacos 2004). First, our algorithm uses the *F&I* instruction, which is easier to implement efficiently in both hardware and software (Ellen et al. 2012; Freudenthal and Gottlieb 1991) than *T&S*, *CAS*, and *F&A*. All other algorithms in this class proposed in the literature use either *CAS*, *F&A*, and *CAS* or *F&A* and *T&S*. Second, our algorithms are simple, elegant, and scalable. Most importantly, our second algorithm assures constant RMR complexity and that partly solves the open problem posed in Bhatt and Huang (2010) as our algorithms do not satisfy concurrent entry property.

Besides the above-mentioned advantages of the algorithms proposed in this article, they have the following other differences with the algorithms presented in Bhatt and Huang (2010), Btlelloch et al. (2003), and Danek and Hadzilacos (2004).

Fig. 3. Average delay per CS entry: With two forums.

- Compared to the algorithms given in Bhatt and Huang (2010) and Danek and Hadzilacos (2004), the algorithms A2 and A3 presented in this article
 - can work with an arbitrary number of threads. This is quite appealing for most practical applications with dynamic workloads, as the shared resources in such applications are generally stable and known a priori, while the number of threads that could access them may change.
 - are easy to understand as they are expressed directly using a small set of simple instructions (read, write, $F&I$) and straightforward logic. The four algorithms presented in Bhatt and Huang (2010) and Danek and Hadzilacos (2004) have layered structure as they assume mutual exclusion algorithms and higher-level objects such as a priority queue and counters as their main components. They further assume that such higher-level objects could be implemented using another set of higher-level objects such as f -arrays.
 - satisfy concurrency and fairness properties P4, P5, P8, and P9. The algorithm presented in Bhatt and Huang (2010) satisfies P6 and P9. The first algorithm in Danek and Hadzilacos (2004) satisfies P6, P7, and P8, and the other two algorithms satisfy only P6.
 - assure RMR complexity in terms of the number of forums (the first algorithm, $O(m)$) and constant (the second algorithm, $O(1)$). The RMR complexity of their algorithm depends on the number of threads and the number of forums.
 - A major limitation of the algorithm presented in Bhatt and Huang (2010) is that it assumes infinite shared space.

Fig. 4. Average delay per CS entry: With four forums.

Table 1. Performance Summary (n - number of threads, m - number of forums)

Algorithm	Set of Threads	Atomic Operations	SA (P4)	NLE (P5)	CCE (P6)	FIFO (P7)	FIFE (P8)	F-FCFS (P9)	RMR (CC Model)
DH	Fixed	CAS, F&A	N	N	Y	N	N	N	$O(\log n \log m)$
BH	"	CAS	N	N	Y	N	N	Y	$O(\min(k, \log n))$
BCG (A1)	Arbitrary	F&A, T&S	Y	Y	N	N	Y	N	$O(m)$
A2	"	F&I	Y	Y	N	N	Y	Y	$O(m)$
A3	"	F&I	Y	Y	N	N	Y	Y	$O(1)$

Note: DH = Danek and Hadzilacos (2004), BH = Bhatt and Huang (2010), BCG = Blleloch et al. (2003).

- Compared to the algorithm given in Blleloch et al. (2003), the algorithms A2 and A3 presented in this article have several advantages.
 - A2 and A3 use a weaker primitive *F&I* (instead of *F&A* and *T&S*).
 - A2 and A3 satisfy an additional fairness property F-FCFS (P9).
 - A3 assures constant RMR complexity in CC model.

Another difference is that our algorithm satisfy the concurrency properties P4 and P5, whereas the algorithms presented in Bhatt and Huang (2010) and Danek and Hadzilacos (2004) satisfy property P6 and it is effective only when there are no conflicting requests. Overall, the algorithms A2 and A3 presented in this article are simple, scalable, and fair. A comparison summary is given in Table 1.

9 CONCLUSION

As multicore systems has emerged as a new trend in computing, concurrent programming is expected to become a mainstream activity. In concurrent programming environments, asynchronous threads sharing data structures and databases is a frequent operation. The group mutual exclusion problem primarily deals with coordinating access to such shared data structures and shared databases. Designing concurrent algorithms and concurrent data structures is generally hard. Designing concurrent algorithms with nice properties is both interesting and challenging.

In this article, we have presented two simple algorithms to solve the group mutual exclusion problem in CC shared memory models. The algorithms are efficient and have several attractive properties as described in the previous sections.

The algorithms have been verified mechanically with the proof assistant PVS. It is possible to extend this proof to the case of bounded integers with wrap-around. The arguments used in Section 6 must then be augmented with refinement techniques as proposed by Abadi and Lamport (1991).

It is possible to extend the algorithms of this article in such a way that, when a thread opens a session of a forum, it can choose a lower bound *lwb* and an upper bound *upb* for the number of threads that will attend the session that is to be opened:

```

if ticket = grant[fm] then
  await grant[fm] + lwb ≤ wait[fm] ;
  session := F&I(tail) ;
  await head = session ;
  upb := min(upb, wait[fm] − grant[fm]) ;
  grant[fm] := grant[fm] + upb ;
endif.

```

Setting the upper bound could avoid hotspot contention and maintain the quality of service. Setting the lower bound could assure minimum concurrency and hence maintain shared resource utilization above a set level. A deeper analysis of this extension is a future work.

Algorithm A3 of this article assures constant RMR in CC shared memory models. This algorithm assure the concurrency properties P4 (simultaneous acceptance) and P5 (no late entry). As a future work, we are interested in investigating the existence of GME algorithm that assures constant RMR in CC model using F&A instruction and satisfy the concurrency property P6 (concurrent entering).

The algorithms presented in the article requires *F&I* atomic operation. This limits their portability. It would be an interesting research direction to explore designing group mutual exclusion using only read and write operations that also have the properties of the algorithm proposed in this article. Going further in that direction, it would be interesting explore the ways to design group mutual exclusion algorithms that do not require atomicity at all.

ACKNOWLEDGMENTS

We sincerely appreciate Darshik Shirodaria for the performance study. We also thank Guy Blelloch, Associate Editor, for his effort on coordinating the review process. We appreciate the reviewers' constructive feedback that helped to increase the clarity of presentation.

REFERENCES

- Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theor. Comput. Sci.* 82, 2 (May 1991), 253–284. DOI: [http://dx.doi.org/10.1016/0304-3975\(91\)90224-P](http://dx.doi.org/10.1016/0304-3975(91)90224-P)
- K. Alagarsamy and K. Vidsankar. 1999. *Elegant Solutions for Group Mutual Exclusion Problem*. Technical Report. Department of Computer Science, Memorial University, St. John's, Newfoundland, Canada.
- T. E. Anderson. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 1, 1 (Jan. 1990), 6–16. DOI: <http://dx.doi.org/10.1109/71.80120>

- J. H. Anderson, Y. J. Kim, and T. Herman. 2003. Shared-memory mutual exclusion: Major research trends since 1986. *Distrib. Comput.* 16, 2–3, (2003), 75–110. DOI : <https://doi.org/10.1007/s00446-003-0088-6>
- Krzysztof R. Apt, Frank de Boer, and Ernst-Rüdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs* (3rd ed.). Springer Publishing Company, Incorporated.
- Vibhor Bhatt and Chien-Chung Huang. 2010. Group mutual exclusion in $O(\log N)$ RMR. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'10)*. ACM, New York, NY, 45–54. DOI : <http://dx.doi.org/10.1145/1835698.1835708>
- Guy E. Blelloch, Perry Cheng, and Phillip B. Gibbons. 2001. Room synchronizations. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*. ACM, New York, NY, 122–133. DOI : <http://dx.doi.org/10.1145/378580.378605>
- Guy E. Blelloch, Perry Cheng, and Phillip B. Gibbons. 2003. Scalable room synchronizations. *Theory Comput. Syst.* 36, 5 (2003), 397–430. DOI : <http://dx.doi.org/10.1007/s00224-003-1081-y>
- Mani Chandy, K. and Jeydev Misra. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley Longman, Boston, MA.
- Robert Danek and Vassos Hadzilacos. 2004. Local-spin group mutual exclusion algorithms. In *Distributed Computing*, Rachid Guerraoui (Ed.). Lecture Notes in Computer Science, Vol. 3274. Springer, Berlin, 71–85. DOI : http://dx.doi.org/10.1007/978-3-540-30186-8_6
- E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (Sep. 1965), 569–. DOI : <http://dx.doi.org/10.1145/365559.365617>
- Faith Ellen, Vijaya Ramachandran, and Philipp Woelfel. 2012. Efficient fetch-and-increment. In *Proceedings of the 26th International Conference on Distributed Computing (DISC'12)*. Springer-Verlag, Berlin, 16–30. DOI : http://dx.doi.org/10.1007/978-3-642-33651-5_2
- Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. 1989. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. Program. Lang. Syst.* 11, 1 (Jan. 1989), 90–114. DOI : <http://dx.doi.org/10.1145/59287.59292>
- Eric Freudenthal and Allan Gottlieb. 1991. Process coordination with fetch-and-increment. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, 260–268. DOI : <http://dx.doi.org/10.1145/106972.106998>
- Vassos Hadzilacos. 2001. A note on group mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*. ACM, New York, NY, 100–106. DOI : <http://dx.doi.org/10.1145/383962.383997>
- Wim H. Hesselink. 2015a. Mutual exclusion by four shared bits with not more than quadratic complexity. *Sci. Comput. Program.* 102, C (May 2015), 57–75. DOI : <http://dx.doi.org/10.1016/j.scico.2015.01.001>
- Wim H. Hesselink. 2015b. PVS proof of an algorithm for group mutual exclusion. Retrieved from <http://wimhesselink.nl/mechver/groupMX>.
- Prasad Jayanti, Srdjan Petrovic, and King Tan. 2003. Fair group mutual exclusion. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC'03)*. ACM, New York, NY, 275–284. DOI : <http://dx.doi.org/10.1145/872035.872077>
- Yuh-Jzer Joung. 1998. Asynchronous group mutual exclusion (extended abstract). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*. ACM, New York, NY, 51–60. DOI : <http://dx.doi.org/10.1145/277697.277706>
- Patrick Keane and Mark Moir. 1999. A simple local-spin group mutual exclusion algorithm. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*. ACM, New York, NY, 23–32. DOI : <http://dx.doi.org/10.1145/301308.301319>
- Jayadev Misra. 2001. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag, New York, NY.
- Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY, 103–112. DOI : <http://dx.doi.org/10.1145/2442516.2442527>
- Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs. *Acta Inf.* 6, 4 (1976), 319–340. DOI : <http://dx.doi.org/10.1007/BF00268134>
- S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. 2001. *PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference*.
- Masataka Takumura and Yoshihide. 2003. Group mutual exclusion algorithms based on ticket orders. In *Proceedings of the 9th Annual International Computing and Combinatorics Conference (COCOON'03)*. Lecture Notes in Computer Science, Vol. 2697, 232–241.

Received October 2015; revised October 2016; accepted December 2018