Mutual exclusion by four shared bits with not more than quadratic complexity

Hesselink, Willem H.

Published in:
Science of computer programming

DOI:
10.1016/j.scico.2015.01.001

Publication date:
2015

Science of Computer Programming 102 (2015) 57–75

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Mutual exclusion by four shared bits with not more than quadratic complexity

Wim H. Hesselink

Dept. of Computing Science, University of Groningen, P.O. Box 407, 9700 AK Groningen, The Netherlands

ARTICLE INFO

Article history:
Received 19 February 2014
Received in revised form 4 November 2014
Accepted 5 January 2015
Available online 14 January 2015

Keywords:
Concurrent algorithm
Mutual exclusion
Fault tolerance
Time complexity
Verification

ABSTRACT

For years, the mutual exclusion algorithm of Lycklama and Hadzilacos (1991) [21] was the optimal mutual exclusion algorithm with the first-come-first-served property, with a minimal number of (non-atomic) communication variables (5 bits per thread). Recently, Aravind published an improvement of it, which uses 4 bits per thread and has simplified waiting conditions. This algorithm is extended here with fault tolerance, and it is verified by assertional methods, using the proof assistant PVS. Progress is proved by means of UNITY logic. The paper proposes a new measure of concurrent time complexity, and proves that the concurrent complexity for throughput of the present algorithm is not more than quadratic in the number of threads.

1. Introduction

Due to the advent of multiprocessors and multicore architectures, the practical relevance of concurrent algorithms is increasing dramatically. This raises the interest in the correctness of them because, as is well known, such algorithms can unexpectedly misbehave due to subtle bugs or race conditions.

Testing and model checking are important methods to find errors, but, for concurrent algorithms, they are often not able to ensure correctness. Formal verification of a concurrent algorithm usually requires many case distinctions. Therefore, even carefully drafted man-made proofs are hardly convincing. In recent years, the advance of mechanical theorem provers like ACL2, Coq, HOL, Isabelle, PVS has made it possible to prove concurrent algorithms exhaustively, and in such a way that the proof script can be inspected and replayed to verify the correctness claims of the verifier. Effective use of a prover for these purposes requires a good understanding of the methods of concurrency verification.

The present paper illustrates these possibilities by presenting a computer assisted verification of a recent improvement by Alex Aravind [5] of the mutual exclusion algorithm of Lycklama and Hadzilacos [21]. For many years, the algorithm of [21] was the “best possible one”, in the sense that it provides mutual exclusion with the first-come first-served property by means of a minimum number of communication variables: 5 bits per thread which need not be atomic.

Aravind’s improvement reduces the number of communication bits to 4, simplifies the waiting conditions, and retains the other properties of the algorithm. Moreover, the algorithm is fault tolerant in the sense that a thread may fail at any time. A failed thread goes immediately to its noncritical section, but its communication variables may get arbitrary values. After some period of time it resets them, and it may try and reenter the protocol.

The algorithm and its proof work under the assumption of sequentially consistent memory. If one wants to apply it on current hardware, memory fences are needed to prevent the hardware from reordering loads before stores [7, Section 4].

E-mail address: w.h.hesselink@rug.nl.

http://dx.doi.org/10.1016/j.scico.2015.01.001
0167-6423/© 2015 Elsevier B.V. All rights reserved.

Simplified versions of the algorithms of Lycklama–Hadzilacos and Aravind were tested with appropriate memory fences in [7, Section 17].

In the present paper, the algorithm is verified completely, including nonatomicity and fault tolerance. The verification is done entirely with assertional methods, i.e., in terms of states, the next state relation, and the forward steps done under weak fairness. The safety properties are verified by invariants and history variables. Progress of the algorithm is verified with a bounded version of UNITY logic [9,12]. In this way, we obtain explicit bounds for throughput and individual delay.

Contributions:

- Addition of fault tolerance and fault recovery.
- Verification of safety and progress.
- Explicit bounds for throughput and individual delay in terms of rounds.

1.1. Overview

Section 1.2 presents the mutual exclusion problem (MX) and the first-come-first-served property (FCFS). Section 1.3 introduces the solution of Lycklama and Hadzilacos, as improved by Aravind. In Section 1.4, we explain our time complexity for concurrent algorithms. Section 1.5 sketches the approach to verification.

Section 2 presents the algorithm with 4 shared bits. It first explains how Lycklama and Hadzilacos [21] have separated the concerns for MX and FCFS. It then presents Aravind's algorithm along these lines.

Section 3 presents the formal model for concurrent algorithms with shared memory, introduces UNITY and our bounded version of it, and then discusses atomicity and nonatomic shared variables.

In Section 4, we decorate the algorithm as presented in Section 2 with history variables and environment steps in such a way that it forms a blueprint for a transition system amenable to formal verification. We prove that this system satisfies MX and FCFS, and absence of immediate deadlock.

Progress is treated in Section 5. Here, the fault tolerance complicates matters considerably, because a frequently failing thread can obstruct the progress of nonfailing threads completely. We conclude in Section 6.

1.2. Mutual exclusion

The problem that concurrent processes may need exclusive access to some shared resource was first proposed and solved in [10]. The problem came to be known as mutual exclusion in [11]. Numerous solutions to this problem have been proposed, e.g., see the surveys [2,7,25,27].

An early and elegant solution is Lamport's bakery algorithm [18], which has three additional properties: it has the first-come-first-served property (FCFS), the shared variables used need not be atomic, and it is fault tolerant in a certain sense. The second point means that the algorithm does not assume mutual exclusion on read and write operations on shared variables. On the other hand, these shared variables hold integer values that can become arbitrarily large.

In the past, devising busy-waiting solutions to the mutual exclusion problem was primarily an academic exercise, because busy waiting is inefficient on a single processor. The advent of multiprocessors and multicore architectures, however, has spurred renewed interest in such algorithms [3, p. 133].

Similarly, algorithms for nonatomic shared variables are becoming practically relevant, because several recent systems such as smart-phones, multi-mode handsets, multiprocessor systems, network processors, graphics chips, and other high performance electronic devices use multiport memories, and such memories allow nonatomic accesses through multiple ports [17,26,28].

Mutual exclusion without FCFS does not require much shared memory. Indeed, the algorithm of Burns–Lamport [8,19] establishes mutual exclusion with only one nonatomic shared Boolean variable per thread.

Lamport's bakery algorithm for N threads gives mutual exclusion with FCFS, using only N nonatomic shared integer variables, but these variables cannot be bounded. The algorithm of [6] also gives mutual exclusion with FCFS. It uses N nonatomic variables with values $\leq N$, and $3N+2$ nonatomic shared bits.

In terms of shared-space complexity, however, the best mutual exclusion algorithm with the FCFS property, that is known to us, is the one of Lycklama and Hadzilacos [21], or rather the recent simplification by Aravind [5].

1.3. Mutual exclusion by Lycklama–Hadzilacos–Aravind

The mutual exclusion algorithm of Lycklama and Hadzilacos [21] establishes mutual exclusion with the FCFS property for an arbitrary number of threads. Per thread, it uses five shared Boolean variables, which need not be atomic.

The presentation of the algorithm splits it in two parts: an inner algorithm to establish mutual exclusion and an outer algorithm to guarantee the FCFS property. The inner algorithm is the mutual exclusion algorithm of Burns [8] and Lamport [19], which uses one shared bit per thread. The outer algorithm uses four shared bits per thread. The combined algorithm thus uses five bits per thread.

In [5], Alex Aravind simplifies the outer algorithm of [21] in such a way that it only needs three bits per thread and has simpler waiting conditions. He also makes the inner algorithm more flexible. The papers [21,5] contain behavioral correctness proofs that we are unable to follow. We give an assertional proof instead.

We also show that Aravind's version of the algorithm allows a thread to fail at any time. This is based on a fault model, described below in Section 2.5, which is slightly stronger than the one of Lamport [18].

1.4. Concurrent time complexity

Time complexity for concurrent algorithms is a difficult issue because at any time there are usually several threads that can do steps, and these steps may or may not serve the purposes of the algorithm. Indeed, we need to partition the steps of the algorithm into two classes: the environment steps that model the task of the algorithm and the forward steps that model the solution. The algorithm can only be expected to accomplish its task when sufficiently many forward steps are taken in a useful order. We make this explicit by prescribing a set of forward steps that are performed under weak fairness, which means that in every execution, from any time onward, such a step is eventually taken if it is forever enabled.

We quantify this idea by introducing rounds [12]. A round is a finite execution in which each of the forward steps is at some time either disabled or taken. We measure the concurrent time complexity of reaching some goal by giving an upper bound for the number of rounds needed. This approach is formalized in Section 3.3 in a bounded version of UNITY [9,12].

1.5. Verification

Concurrent algorithms are difficult to design because of the possible interference between actions of different threads. Testing is not the solution because incorrectness may only show up in very unlikely scenarios. It follows that verification is necessary. Verification of concurrent algorithms, however, often requires complicated arguments with large case distinctions. We therefore use the proof assistant PVS [24]. All main assertions in this paper have been proved with PVS. The proof file can be obtained from [16, Section 2]. PVS users can inspect it to check the correspondence of the proof goals with the results claimed in the paper, and to replay the proof on their own system.

For any concurrent algorithm, verification has two aspects: safety and progress. Safety means that nothing goes wrong: in this case, it amounts to mutual exclusion, the first-come-first-served property, and absence of immediate deadlock. Progress means that every thread that aims at the critical section will eventually reach it and return to the idle state. Safety is proved here with invariants and auxiliary variables, just as we used in [15] for a closely related algorithm. We prove progress with a bounded version of UNITY logic [9,12].

2. Mutual exclusion with four bits

In this section, we present the algorithm without any correctness considerations. In Section 2.1, we introduce the problem of mutual exclusion (MX) and the first-come-first-served-property (FCFS). Section 2.2 presents the algorithm as a combination of an MX algorithm and an FCFS algorithm. The MX algorithm is explained in Section 2.3. The FCFS algorithm is explained in Section 2.4. For simplicity, in these sections, we ignore the questions of fault tolerance. Fault tolerance and fault recovery are treated in Section 2.5.

2.1. Mutual exclusion described

Traditionally, mutual exclusion (MX) is modeled as follows. There are N threads, numbered from 0 upward, that communicate via shared variables and that repeatedly may compete for unique access to a shared resource. We thus use the set Thread $= \{p \in \mathbb{N} \mid p < N\}$. The threads are of the form:

```
thread (p : Thread) =
    while true do
        NCS ; Doorway ; Waiting ; CS ; Exit
    endwhile .
```

NCS and CS are given program fragments. NCS is the *noncritical section*, which need not terminate. Access to the shared resource is modeled as the critical section CS, which is guaranteed to terminate. The aim is to implement Doorway, Waiting, and Exit in such a way that the number of threads in CS is guaranteed to remain ≤ 1 (*mutual exclusion*). Thread p is said to be *idle* when it is at NCS, otherwise it is *competing*.

The first-come-first-served property (FCFS) is defined in [18] to mean that, if thread p enters Doorway while thread q is in Waiting, thread p will not enter CS before q does. It is a safety property (it would be bad when p enters before q).

The progress requirements are firstly that Doorway and Exit can be traversed without waiting, and secondly that, when there is a thread in Waiting and no thread in CS, eventually a thread will enter CS. Together with FCFS, it implies lockout freedom: every thread in Waiting eventually enters CS.

```
(i)    thread (p) =
         while true do
           NCS ; Doorway ; WaitingFcfs ;
    (†)    WaitingMx ; CS ; ExitMx ;
           ExitFcfs
         endwhile .
(ii)  thread (p) =
        while true do
          NCS ; Doorway ; WaitingFcfs ;
          WaitingMx ; ExitFcfs ; CS ; ExitMx
        endwhile .
```

Fig. 1. Two ways for splitting mutual exclusion and FCFS.

```
     thread (p : Thread) =
21     NCS ;
22     dw[p] := true ;
23     for all k ∈ Range do copy[k] := turn[k] end ;
24     turn[2p + nx] := true ;
25     dw[p] := false ;
26     for all k ∈ Range with copy[k] do  27 await ¬turn[k] end ;
28     cc[p] := true ;
29     for all thr < p with cc[thr] do
30       cc[p] := false ;
31       await ¬cc[thr] ; goto 28
       end ;
32     for all thr > p do  33 await ¬cc[thr] end ;
34     turn[2p + nx] := false ; nx := 1 − nx ;
35     for all thr ∈ Thread do  36 await ¬dw[thr] end ;
37     CS ;
38     cc[p] := false ; goto 21 .
```

Fig. 2. Aravind's algorithm for 4 shared bits.

2.2. The algorithm of Lycklama–Hadzilacos–Aravind

In the mutual exclusion algorithm of [21], MX and FCFS are accomplished by two separate algorithms that are nested as shown at (i) in Fig. 1. The line (†) holds the inner algorithm for mutual exclusion. It requires one shared bit per thread. The outer algorithm to guarantee FCFS consists of Doorway, WaitingFcfs, and ExitFcfs.

The FCFS algorithms of [21,15] require 4 shared bits per thread. Aravind [5] improved the FCFS algorithm so that it only needs three shared bits per thread but that ExitFcfs requires waiting. As a mutual exclusion algorithm should have no waiting after CS, he moved ExitFcfs to the position just before CS, thus getting the nesting shown at (ii) in Fig. 1.

The combined algorithm is presented in Fig. 2. The fragment of the commands 28 up to 33 together with command 38 is the mutual exclusion algorithm of Burns–Lamport, which is described below in Section 2.3. The part for FCFS is described in Section 2.4. It consists of Doorway in commands 22–25, WaitingFcfs in commands 26–27, and ExitFcfs in commands 34–36.

As it combines the algorithms for mutual exclusion and FCFS, the algorithm uses four shared booleans (bits) per thread. The boolean $\texttt{dw}[p]$ holds when thread p is in the Doorway. The boolean $\texttt{cc}[p]$ holds when thread p is in the inner algorithm and has not yet given priority to a lower numbered thread. The boolean $\texttt{turn}[2p+\mathsf{nx}]$ with $\mathsf{nx} \in \{0,1\}$ serve to ensure FCFS by indicating that thread p is in its waiting sections. All these bits are initially false.

We use the convention that shared variables are in typewriter font, while private variables are set in sans serif font. All threads have the same private variables. If v is a private variable, it is denoted by v in the code of p, but outside the code we write $v.p$ for the value of v of thread p.

We use labels beginning at 21 to ease proof refactoring as described in [15]. We write p **at** ℓ to indicate that thread p is at the command labeled ℓ, i.e., that $\mathsf{pc}.p = \ell$. We write p **in** L for $\mathsf{pc}.p \in L$.

2.3. Mutual exclusion implemented

The inner algorithm is a flexible version of the mutual exclusion algorithm of Burns [8] and Lamport [19]. In Fig. 2, it consists of the commands 28–33 and 38. It uses a single nonatomic Boolean shared variable per thread, called $\texttt{cc}[p]$, which is initially false. We thus use the declaration:

$$\texttt{cc}: \textbf{array}\,[\mathsf{Thread}]\ \textbf{of}\ \mathbb{B}\,.$$

The idea is that, when it starts competing, thread p makes $\texttt{cc}[p]$ true, and then inspects $\texttt{cc}[q]$ for all threads $q \neq p$, first for the threads $q < p$. When it encounters a thread $\mathsf{thr} < p$ with $\texttt{cc}[\mathsf{thr}]$, it gives precedence to thr by making $\texttt{cc}[p]$ false, waits until thr has made $\mathtt{cc}[\mathsf{thr}]$ false, and then sets $\mathtt{cc}[p]$ true and restarts its inspection of the lower threads. When it has inspected all lower threads successfully, it turns to the threads $q > p$. When it finds a thread $\mathsf{thr} > p$ with $\mathtt{cc}[\mathsf{thr}]$, it just waits until thread thr has made $\mathtt{cc}[\mathsf{thr}]$ false. When its inspection loop terminates, thread p can enter CS. After CS, it makes $\mathtt{cc}[p]$ false.

2.4. Design for FCFS

Recall from Section 2.1 that FCFS means that, if thread p enters Doorway while thread q is in Waiting, thread p will not enter CS before q does.

The FCFS algorithm uses three nonatomic Boolean shared variables per thread p. They are declared as follows:

$$\begin{aligned}&\mathsf{Range} = \{k \in \mathbb{N} \mid k < 2N\}\,,\\ &\mathtt{turn}:\ \textbf{array}\ [\mathsf{Range}]\ \textbf{of}\ \mathbb{B}\,;\\ &\mathtt{dw}:\ \textbf{array}\ [\mathsf{Thread}]\ \textbf{of}\ \mathbb{B}\,.\end{aligned}$$

Thread q is the owner of the three variables $\mathtt{dw}[q]$ and $\mathtt{turn}[2q + e]$ with $e \in \mathsf{Bit}$ where $\mathsf{Bit} = \{0, 1\}$. Note that every $k \in \mathsf{Range}$ has a unique expression $k = 2q + e$ with $q \in \mathsf{Thread}$ and $e \in \mathsf{Bit}$.

The main private variables of the threads are declared by

$$\begin{aligned}&\mathsf{copy}:\ \textbf{array}\ [\mathsf{Range}]\ \textbf{of}\ \mathbb{B}\,;\\ &\mathsf{nx}: \mathsf{Bit}\ (\text{persistent})\,.\end{aligned}$$

In command 23 of Fig. 2, thread p makes a copy of array $\mathtt{turn}$ in its private variable copy. This copy turns out to be useful, even though, because of possible interferences, we cannot claim any equality between $\mathsf{copy}.p$ and $\mathtt{turn}$.

In command 24, the thread announces that it is competing by modifying one of its bits in array $\mathtt{turn}$. After Doorway, the thread waits at commands 26–27 until all threads in its private copy have completed WaitingMx. Then it can enter the inner mutual exclusion algorithm. When it gets access to the critical section, it removes its announcement at command 34, and enters the critical section.

This would guarantee FCFS because the thread does not enter CS before all threads in its private copy have completed CS. Unfortunately, if we follow this informal description, it can lead to deadlock, as is shown by the following scenario: thread p_0 announces itself, thread p_1 observes the announcement of p_0 and announces itself, thread p_0 concludes its competing period (via CS), enters again, observes the announcement of p_1, and announces itself again. Now both p_0 and p_1 have announced and are waiting until the other removes its announcement: deadlock.

In order to break these cyclic dependencies, we give the announcements of thread p cyclic version numbers, kept in a persistent private variable nx declared above. Indeed, in the commands 24 and 34 of Fig. 2, thread p indicates its interest in the critical section at the index $2p + \mathsf{nx}.p$ in array $\mathtt{turn}$. This, however, is not enough, because while thread p_1 is in the doorway, thread p_0 may cycle several times through competing periods and then use the version number observed by p_1 again.

In order to preclude this unrestricted progress, every thread (p_1 in this scenario) waits at the end of ExitFcfs (at commands 35–36) for all threads to leave the Doorway. Indeed, thread p sets $\mathtt{dw}[p]$ in command 22 and resets it in command 25. The other threads may have to wait for $\neg\mathtt{dw}[p]$ at command 36. It turns out that, in this way, the cyclic version numbers can be bits. We come back to this in the proof of correctness.

2.5. Fault tolerance

The conclusion of the paper [21] mentions that it is possible, though complicated, to extend the algorithm of [21] with fault tolerance. For the present algorithm, this is much easier.

We use the following fault model. A faulty thread in the critical section immediately exits the critical section. There may be a period when reading its communication variables gives arbitrary values. Eventually, however, all its communication variables are reset to false. We allow the thread to reenter the protocol. This final possibility is not in Lamport's fault model [18].

The code for failure recovery is given in Fig. 3. The recovery consists of resets of the communication variables, followed by two await loops. These loops serve to guarantee that the thread that has failed, say p, does not reenter with $2p + e \in \mathsf{copy}.q$ for any $e \in \mathsf{Bit}$ and any thread q. The first loop ensures that any thread q with $2p + e \in \mathsf{copy}.q$ has set its flag $\mathtt{turn}[2q + \mathsf{nx}.q]$. Thread p waits in the second loop for these flags to be reset.

3. The model and the repertoire

We need a formal model to argue the correctness of the algorithm: its safety, its progress, and its time complexity. The basic model is described in Section 3.1. UNITY logic is introduced in Section 3.2. In Section 3.3, we present bounded UNITY, our variation with a form of concurrent time complexity. Atomicity is discussed in Section 3.4. Nonatomic shared variables are introduced in Section 3.5.

```
**recovering thread**$(p)$ :
39   dw[$p$] := false ; cc[$p$] := false ;
41   turn[$2p$] := false ; turn[$2p+1$] := false ;
43   for all thr $\in$ Thread do    44 await $\neg$ dw[thr] end ;
45   for all $k \in$ Range do    46 await $\neg$ turn[$k$] end ; goto 21 .
```

Fig. 3. Failure recovery.

3.1. The basic model

Our basic model for concurrent algorithms with shared memory is a formal version of UNITY [9]. There is a set of threads (processes) that can do steps. The *state* of the system is given by the values of all shared and private variables. The state space X of the system is the set of all states. It is the Cartesian product of a shared state space and the private state spaces of the threads. If P is a predicate on the state space X, it can also be regarded as the subset of X where 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 $\mathsf{step}(p)$ 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 $\mathsf{step}(p)$ 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 $\mathsf{xs}_0 \in \mathsf{start}$, and $(\mathsf{xs}_n, \mathsf{xs}_{n+1}) \in \mathsf{step}$ 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 that we have obtained for the algorithm.

An execution fragment of length $n \geq 0$ is a nonempty finite sequence $(\mathsf{xs}_0 \dots \mathsf{xs}_n)$ in X_0 such that $(\mathsf{xs}_i, \mathsf{xs}_{i+1}) \in \mathsf{step}$ 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.

Let the steps of the threads be partitioned in *forward steps* which serve the goal of the algorithm, and environment steps that model the task of the algorithm or possible obstructions. Relation $\mathsf{fwd}(p) \subseteq \mathsf{step}(p)$ 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 $(x, y) \in \mathsf{fwd}(p)$. Enabledness of thread p is expressed by $\mathsf{ena}(p)$:

$$x \in \mathsf{ena}(p) \equiv \big(\exists y : (x, y) \in \ \mathsf{fwd}(p)\big).$$

An *occurrence* of thread p in an execution fragment $(\mathsf{xs}_0 \dots \mathsf{xs}_n)$ is a number i with $0 \leq i < n$, and $(\mathsf{xs}_i, \mathsf{xs}_{i+1}) \in \mathsf{fwd}(p)$ or $\mathsf{xs}_i \notin \mathsf{ena}(p)$. The execution fragment is called a *round* if it contains an occurrence of every thread. In other words, in a round, every thread is scheduled at least once, and is either executed or found to be disabled.

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.

3.2. UNITY logic

UNITY logic is designed to prove progress in the form of leads-to relations between predicates. P *leads to* Q means that, for every execution fragment $(\mathsf{xs}_0 \dots \mathsf{xs}_n)$ with $\mathsf{xs}_0 \in P$ and sufficiently many rounds, there is a number k with $\mathsf{xs}_k \in Q$. UNITY logic is designed to prove this by assertional means, i.e., by only using states and step relations, and without considering executions or rounds.

Example. For a mutual exclusion algorithm as described in Section 2.1, one may want to assert that every thread p that has entered the Doorway will eventually reach the critical section CS. In UNITY, this is expressed by (p **in** Doorway) leads to (p **in** CS). □

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

$$P \ \mathbf{co}\ Q \ \equiv \forall (x, y) \in \mathsf{step} : x \in P \Rightarrow y \in Q\,,$$

$$P \ \mathbf{co!}\ Q \equiv \exists r : P \subseteq \mathsf{ena}(r) \\ \wedge \big(\forall (x, y) \in \mathsf{fwd}(r) : x \in P \Rightarrow y \in Q\big).$$

P **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 [9,22] is based on the relations **unless** and **ensures** defined by:

$$P \ \mathbf{unless}\ Q \equiv (P \wedge \neg Q \wedge X_0)\ \mathbf{co}\ (P \vee Q),$$

$$P \ \mathbf{ensures}\ Q \equiv (P\ \mathbf{unless}\ Q) \wedge \big((P \wedge \neg Q \wedge X_0)\ \mathbf{co!}\ Q\big).$$

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

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

It is easy to see that $P \mapsto Q$ implies that "P leads to Q" as it is described above. In fact, it is equivalent, as is well known, e.g., see [13] and the references given there.

Remark. The books [9,22] define **unless** and **ensures** without mentioning a set X_0. Instead, they allow invariants to be regarded as axioms, wherever one might need them. This is justified for a design methodology. For verification with a proof assistant, it is safer to refrain from introducing axioms when this can be avoided.

3.3. *Bounded* UNITY

In bounded UNITY, the leads-to property is quantified by counting the number of rounds, see Section 3.1.

For predicates P and Q, we say that P *leads to* Q *within* k *rounds* (notation P $\mathbf{Lt}\langle k\rangle$ Q), if every execution fragment that starts with a state in P and that contains a concatenation of k rounds, contains a state in Q. The number k is called the *progress bound* of the leads-to property.

It is relatively easy to prove the following rules:

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

These rules are called the subset rule, the **ensures** rule, transitivity, and the disjunction rule, respectively.

We also have the Progress–Safety–Progress rule [9, p. 65]:

(PSP) If P $\mathbf{Lt}\langle k\rangle$ Q and A **unless** M, then $(P \wedge A)$ $\mathbf{Lt}\langle k\rangle$ $((Q \wedge A) \vee M)$.

A progress proof with UNITY can often be converted into a proof with bounded UNITY. The bounds give an indication of the concurrent complexity.

Example. Consider a system with three shared variables $\mathtt{b}, \mathtt{c} : \mathsf{Bit}$, $\mathtt{n} : \mathbb{N}$, and three atomic commands, regarded as forward steps, to be executed repeatedly:

$$\begin{array}{llll}
[\,] & \mathtt{b}=0 & \rightarrow & \mathtt{b}:=1\,;\ \mathtt{c}:=1-\mathtt{c} \\
[\,] & \mathtt{c}=0 & \rightarrow & \mathtt{n}:=\mathtt{n}+\mathtt{b}\,;\ \mathtt{b}:=0 \\
[\,] & \mathtt{c}=1 & \rightarrow & \mathtt{n}:=\mathtt{n}+\mathtt{b}\,;\ \mathtt{b}:=0\,.
\end{array}$$

The state space X here is the Cartesian product $\mathsf{Bit}^2 \times \mathbb{N}$. We take $X_0 = X$. The three commands prescribe the relations $\mathsf{fwd}(p)$ with $p = 0, 1, 2$. Their guards are the enabling conditions. It is easy to verify the three **ensures** properties (for any $k \in \mathbb{N}$):

$$\begin{array}{l}
(\mathtt{b}=0\ \wedge\ \mathtt{n}\geq k)\ \mathbf{ensures}\ (\mathtt{b}=1\ \wedge\ \mathtt{n}\geq k), \\
(\mathtt{b}=1\ \wedge\ \mathtt{c}=0\ \wedge\ \mathtt{n}\geq k)\ \mathbf{ensures}\ (\mathtt{n}\geq k+1), \\
(\mathtt{b}=1\ \wedge\ \mathtt{c}=1\ \wedge\ \mathtt{n}\geq k)\ \mathbf{ensures}\ (\mathtt{n}\geq k+1).
\end{array}$$

The **ensures** rule and the disjunction rule enable us to derive:

$$(\mathtt{b}=1\ \wedge\ \mathtt{n}\geq k)\ \ \mathbf{Lt}\langle 1\rangle\ \ (\mathtt{n}\geq k+1).$$

Using transitivity and again the disjunction rule, we obtain

$$(\mathtt{n}\geq k)\ \ \mathbf{Lt}\langle 2\rangle\ \ (\mathtt{n}\geq k+1).$$

By transitivity and induction, we thus obtain progress in the form

$$(\mathtt{n}\geq k)\ \ \mathbf{Lt}\langle 2\cdot m\rangle\ \ (\mathtt{n}\geq k+m).$$

Now consider the variation in which we replace the guard $\mathtt{b} = 0$ in the first alternative by true. Then the second and third **ensures** properties are no longer valid: the first alternative can be executed too often and the other two alternatives need never be executed because they are never henceforth continuously enabled. It follows that $\mathtt{n}$ can remain constant. This variation shows the importance of the proof obligation to indicate (in **co!**) an enabled "progressive" step that remains enabled at least until some "progressive" step has been taken. □

3.4. Atomicity

In concurrency, the atomic commands of the threads can be interleaved in arbitrary ways. It is therefore important to specify the grain of atomicity of the commands. According to the *principle of single critical reference*, e.g., [23, (3.1)], [4, p. 273], an atomic command reads or writes at most one shared variable (not both), unless it is specifically provided as a call to the operating system (e.g., a semaphore operation). Actions on private variables can be added to atomic commands because they never lead to interference.

This principle sets the default. The present paper follows the principle completely. In general, we choose the atomicity of the commands as coarse as allowed by the principle. The reason for this is that an algorithm requires more, and more complicated invariants when its grain of atomicity is refined.

Every transition of the transition system corresponds to an atomic command that has a unique label. The control state of thread p is given by the program counter $\mathsf{pc}.p$, which takes values in the set of labels and determines which command is to be executed next. We thus use the labels to indicate the grain of atomicity that is used in our transition system. In other words, in the transition system, control is never at an unlabeled semicolon.

We use a pseudocode with a standard syntax for simple commands, with := for the assignment, **if then** (**else**) **end** for the conditional command and **while do endwhile** for the conditional repetition. In the pseudocode, every command implicitly modifies pc to jump to the next label unless specified otherwise.

Every repetition consists of a test for termination and a body that ends with a backward jump to the test. Consequently, control is repeatedly at the test. If the body including the test is a single atomic command, control does not leave the test until termination of the repetition. In this way, a labeled command can be a complete repetition, but only the steps of the repetition are atomic. For example, the labeled command

$$\ell : \textbf{while } B \textbf{ do } S \textbf{ endwhile};\ T$$

is modeled as

$$\ell : \textbf{if } B \textbf{ then } S;\ \textbf{goto } \ell \textbf{ else } T \textbf{ endif}.$$

In other words, in this single atomic command, B is inspected and either S or T is executed once. In the first case control stays at ℓ. Otherwise control goes to the next label.

A thread can be explicitly disabled by a command of the form

$$\ell : \textbf{await } B;\ S.$$

This command can only be executed when B holds. Execution of it means execution of S followed by a jump to the next label. The command is disabled when B is false.

3.5. Nonatomic shared variables

The algorithms of [21,5,15] truly solve mutual exclusion, in the sense that they do not rely on atomic access of the shared variables (which can be seen as mutual exclusion at a lower level). We thus have to accommodate nonatomic shared variables in our model of executions with atomic steps.

In this paper, every shared variable is an *output* variable, i.e., there is at most one thread that can write it. Lamport [20] distinguishes two kinds of nonatomic output variables: safe ones and regular ones.

An output variable is called *safe* if a read not concurrent with a write obtains the correct value, while a read concurrent with a write may obtain an arbitrary value of the correct type. Concurrent writes do not occur, because the variable is written by at most one thread. A variable is called *regular* if it is safe and a read concurrent with a write can only obtain the old value or the value that is being written. In this paper, all shared variables are only assumed to be safe.

We use the formal model for safe variables of [1]. A read action of a shared variable $\mathtt{x}$ into a private variable v is written $v := \mathtt{x}$. It can be regarded as atomic because it does not influence the shared state. For a safe shared variable, we need to indicate that reading during a write action can return any value. We therefore denote a write action of a private expression E into a safe shared variable $\mathtt{x}$ by

$$\ell : \mathtt{x} := (\text{flickering})\ E. \tag{0}$$

We model this by a nondeterministic choice

$$\ell : \big(\mathtt{x} := \mathsf{arbitrary};\ \textbf{goto } \ell \ [] \ \mathtt{x} := E\big). \tag{1}$$

```
     thread ($p$ : Thread) =
21     predec[$p$] := central ; turnset := Range ;
22     dw[$p$] := (flickering) true ;
23     while exists $k \in$ turnset do
           copy[$k$] := turn[$k$] ; turnset[$k$] := false
       endwhile ;
       cnt := shacnt ; shacnt := shacnt + 1 ;
24     turn[$2p$ + nx] := (flickering) true ; central[$p$] := true ;
25     dw[$p$] := (flickering) false ;
26     while exists kk $\in$ Range with copy[kk] do
27         await $\neg$ turn[kk] ; copy[kk] := false
       endwhile ;
28     cc[$p$] := (flickering) true ; lower := $\{q \mid q < p\}$ ;
29     if exists thr $\in$ lower then
           if $\neg$ cc[thr] then lower[thr] := false; goto 29
           else
30             cc[$p$] := (flickering) false ;
31             await $\neg$ cc[thr] ; goto 28
           endif
       else higher := $\{q \mid p < q\}$ endif ;
32     while exists thr $\in$ higher do
33         await $\neg$ cc[thr] ; higher[thr] := false
       endwhile ;
34     turn[$2p$ + nx] := (flickering) false ; nx := 1 − nx ; dwset := Thread ;
35     while exists thr $\in$ dwset do
36         await $\neg$ dw[thr] ; dwset[thr] := false
       endwhile ;
37     CS ; central[$p$] := false ;
       for all $q$ do predec[$q$][$p$] := false end ;
38     cc[$p$] := (flickering) false ; inc := inc + 1 ; goto 21 .
```

Fig. 4. History extension of the algorithm.

In other words, command (0) is modeled in (1) as a repetition of arbitrary assignments to x that ends with the actual assignment of E to x. The value of x during the repetition is indeterminate. We declare the assignment x := arbitrary to be an environment step, while the final assignment x := E is a forward step. This ensures that the repetition terminates within one round.

4. Transition system and verification of safety

In order to verify the algorithm, we reduce it to a transition system. In Section 4.1, we extend the algorithm of Fig. 2 with control information and history variables. We also pin down the atomic steps. The meanings of failures, fault steps, and fault recovery are treated in Section 4.2.

The verification of the safety of the inner algorithm is done in Section 4.3. In Section 4.4, we specify and verify the FCFS property. In preparation for the proof of deadlock freedom and progress, we present in Section 4.5 the dichotomy between forward steps and environment steps of the algorithm, we analyze disabledness, and define immediate deadlock. Absence of immediate deadlock is proved in Section 4.6. Section 4.7 presents and proves some invariants needed in this proof or later.

4.1. The history extension of the algorithm

In order to prove its correctness, we extend the algorithm by making implicit control information explicit, and by adding some history variables. We use the same labels as in Fig. 2. The result is the algorithm of Fig. 4. We have removed NCS here, because it is superfluous and distracting. We have added "(flickering)" in the assignments to the safe shared variables.

In Fig. 2, the **for** loops at the commands 23, 29–31, 32–33, 35–36 are not meant to be performed atomically, and the loop variables k, thr can be chosen in arbitrary order. We therefore need to extend the state with the sets of values of k and thr that have yet to be treated in the loops. We thus introduce private variables:

turnset : **set of** Range ,
lower, higher, dwset : **set of** Thread .

These sets are initially all empty.

We introduce a private variable kk for the value chosen in command 26, because this value is used in command 27 and it will occur in some invariants.

```
     failing thread($p$) :
       failure step from any location $\neq 39$:
         turnset := copy := $\emptyset$ ;
         lower := higher := $\emptyset$ ;
         central[$p$] := false ; predec[$p$] := $\emptyset$ ;
         for all $q$ do predec[$q$][$p$] := false end ;
         inc := inc + ($p$ in {40...42} ? 1 : 0) ;
         nx := 0 ; goto 39 .
39     choose dw[$p$], cc[$p$], turn[$2p$], turn[$2p+1$] in $\mathbb{B}$ ;
       goto 39 .
39     dw[$p$] := (flickering) false ;
40     cc[$p$] := (flickering) false ;
41     turn[$2p$] := (flickering) false ;
42     turn[$2p+1$] := (flickering) false ;
       dwset := Thread ; inc := inc + 1 ;
43     while exists thr ∈ dwset do
44       await ¬ dw[thr] ; dwset[thr] := false
       endwhile ; turnset := Range ;
45     while exists kk ∈ turnset do
46       await ¬ turn[kk] ; turnset[kk] := false ;
       endwhile ; goto 21 .
```

Fig. 5. Failure and recovery.

For the proof of FCFS, we introduce shared history variables:

central : **set of** Thread ,
predec : **array**[Thread] **of set of** Thread ,
shacnt : $\mathbb{N}$,

with initially central $= \emptyset$, predec[p] $= \emptyset$ for all threads p. In Section 4.6, we discuss the history variables shacnt and cnt.p that occur in command 23. The private history variable inc is introduced in Section 5.3. The history variables can be modified atomically because they are not used in the algorithm, but only in the proof.

Every line number of Fig. 4 stands for an atomic command. In this way, the algorithm satisfies the principle of single critical reference (see Section 3.4), because we can ignore the inspections and modifications of the history variables central, predec, and shacnt. For example, in line 23, if turnset.p is nonempty, turn[k] is inspected for a single value of k, and control goes back to line 23. If turnset.p is empty, the history variables cnt.p and shacnt are inspected and modified, and control goes to line 24.

4.2. Failure and fault recovery

The extension of Fig. 3 with control information and history variables is given in Fig. 5. We make failure explicit by adding a failure step for thread p from any location $\neq 39$ to the fault location 39. For the sake of the invariants, all relevant private variables of p are reset to default values. The same holds for the history variables central[p] and predec[q][p] for all threads q.

When thread p is at label 39, reading of its communication variables can give arbitrary Boolean values. This is modeled by means of a fault step of p that modifies these variables arbitrarily. The failure recovery in the lines 39–46 is in line with Fig. 3. The modifications of inc.p are discussed Section 5.3.

4.3. The inner algorithm to guarantee mutual exclusion

We turn to the verification, first of mutual exclusion. Mutual exclusion (MX) at the critical section CS (line 37) is implied by the invariant:

MX: $\quad q$ **in** $\{34\ldots37\} \wedge r$ **in** $\{34\ldots37\} \Rightarrow q = r$.

Note that, by postulating such an invariant, we implicitly mean that it holds for all values of the free variables $(q, r, \ldots)$.

In order to prove MX, we first claim the invariant

Iq0: $\quad q$ **in** $\{29, 32\ldots37\} \Rightarrow$ cc[q].

Indeed, it is straightforward to prove that predicate Iq0 is inductive: it holds initially (because then thread q is at 21) and is preserved in every step. We use "q"s in the names of invariants to ease proof refactoring as described in [15].

Now predicate MX is implied by the new invariants:

Iq1: q **in** $\{32\ldots\} \Rightarrow \mathsf{lower}.q = \emptyset$,
Iq2: q **in** $\{34\ldots\} \Rightarrow \mathsf{higher}.q = \emptyset$,
Iq3: q **in** $\{29, 32\ldots37\} \wedge r$ **in** $\{29, 32\ldots37\} \wedge q < r$
$\Rightarrow q \in \mathsf{lower}.r \vee q$ **at** $29 \vee r \in \mathsf{higher}.q$.

Indeed, if $q < r$ and both are in $\{34\ldots37\}$, then the sets $\mathsf{lower}.r$ and $\mathsf{higher}.q$ are empty by Iq1 and Iq2. Therefore, Iq3 gives a contradiction, proving MX.

The predicates Iq1 and Iq2 are inductive because of the termination conditions of the loops at 29 and 32.

Preservation of Iq3 is proved as follows. Assume that some thread p does a step that falsifies Iq3. Then $q < r$, and the step of thread p makes the antecedent of Iq3 true or the consequent false. The antecedent is only made true when p equals q or r and executes line 28, in which case it makes the consequent of Iq3 true as well. Thread p makes the consequent of Iq3 false only at the lines 29 and 33 by removing $t = \mathsf{thr}.p$ from $\mathsf{lower}.p$ or $\mathsf{higher}.p$ with $t = q$ or $t = r$. Then we have $\neg\mathtt{cc}[t]$, so that thread t is not in $\{29, 32\ldots37\}$ by Iq0, implying that the antecedent of Iq3 is false. This proves that Iq3 is preserved. We thus have:

Theorem 1. *The algorithm satisfies mutual exclusion* MX.

4.4. The outer algorithm guarantees FCFS

Recall that FCFS means that, if thread p enters Doorway while thread q is in Waiting, thread p will not enter CS before q does.

In order to verify this, we ensure, in Fig. 4, that the history variable `central` satisfies the invariant

Kq0: $r \in \mathtt{central} \equiv r$ **in** $\{25\ldots37\}$.

Therefore, in command 21, thread p receives in its history variable $\mathtt{predec}[p]$ the set of all predecessors. A predecessor q is only removed from $\mathtt{predec}[p]$, when q executes line 37, and thus leaves the critical section. Therefore FCFS is implied by

FCFS: q **in** $\{34\ldots37\} \Rightarrow \mathtt{predec}[q] = \emptyset$.

In order to prove the invariant FCFS, we need some other invariants. For these invariants, it is convenient to interpret the Boolean arrays `turn`, $\mathsf{turnset}.p$ and $\mathsf{copy}.p$ for threads p, as sets of pairs (q, e) with $q \in \mathsf{Thread}$ and $e \in \mathsf{Bit}$, according to the formulas:

$$(q, e) \in \mathtt{turn} \equiv \mathtt{turn}[2q + e],$$
$$(q, e) \in \mathsf{turnset}.p \equiv \mathsf{turnset}.p[2q + e],$$
$$(q, e) \in \mathsf{copy}.p \equiv \mathsf{copy}.p[2q + e]. \qquad (2)$$

In this way, they become well-defined sets because every index $k \in \mathsf{Range}$ has a unique expression $k = 2q + e$ with $q \in \mathsf{Thread}$ and $e \in \mathsf{Bit}$.

We postulate the invariants:

Kq1: $r \in \mathtt{predec}[q] \Rightarrow (r, \mathsf{nx}.r) \in \mathsf{turnset}.q \cup \mathsf{copy}.q \vee r$ **in** $\{34\ldots37\}$,
Kq2: $\mathsf{turnset}.q = \emptyset \vee q$ **in** $\{22, 23, 43\ldots46\}$,
Kq3: $\mathsf{copy}.q = \emptyset \vee q$ **in** $\{23\ldots27\}$,
Kq4: $r \in \mathtt{predec}[q] \Rightarrow q \neq r$.

It is clear that these four predicates together with MX imply FCFS.

It is easy to see that the predicates Kq2 and Kq3 hold initially, are preserved under every step of q, and also under every step of a thread $\neq q$. They are therefore inductive, and hence invariants. Predicate Kq4 holds initially because $\mathtt{predec}[q]$ is empty. It is threatened only at line 21. It is preserved at line 21 because of Kq0.

Predicate Kq1 holds initially because $\mathtt{predec}[q]$ is empty. It is threatened only by the steps 23, 27, 46, and the failure step, because of the modifications of $\mathsf{turnset}.q$, $\mathsf{copy}.q$, and $\mathsf{nx}.r$. Step 21 is harmless because $(r, \mathsf{nx}.r) \in \mathsf{turnset}.q$ becomes true. Kq1 is preserved by the threatening steps because of Kq0 and the additional invariants:

Kq5: $\mathtt{predec}[q] \subseteq \mathtt{central}$,
Kq6: r **in** $\{25\ldots33\} \Rightarrow (r, \mathsf{nx}.r) \in \mathtt{turn}$,
Nq4: q **in** $\{38\ldots\} \Rightarrow \mathtt{predec}[q] = \emptyset$.

The predicates Kq5 and Kq6 are easily seen to be invariant. Preservation of Nq4 at line 37 follows from FCFS.

4.5. Forward steps and environment steps

We prepare the proof of progress by first proving absence of immediate deadlock as defined below. For this purpose, we partition the steps of a thread p in forward steps and environment steps, see Section 3.1.

We define a step of p to be an *environment* step if it is the step from line 21 to 22, a flickering step in which $\mathsf{pc}.p$ remains unchanged, a failure step to label 39, or a fault step at line 39. All other steps of p are *forward* steps of p. The reader may verify that the forward steps are the steps that begin with p not at line 21 and that modify $\mathsf{pc}.p$ or the loop variable $\mathsf{turnset}.p$. The step of 21 is called an environment step because it is the request for access to the critical section. The flickering steps and the failure and fault steps do not belong to the algorithm, but are part of the environment that the algorithm has to accommodate. In other words, the environment steps define the problem while the forward steps form the solution.

The relation $\mathsf{fwd}(p)$ thus consists of the pairs of states (x, y) such that, in state x, thread p can do a forward step that establishes state y. The set Env consists of the pairs (x, y) such that some thread has an environment step that leads from x to y. The step relation of the algorithm is

$$\mathsf{step} = 1_X \cup \mathsf{Env} \cup \bigcup_p \mathsf{fwd}(p).$$

We thus use the setting of Section 3.1. The set X_0 is the conjunction (intersection) of all invariants. Applying the definition in Section 3.1 to the present algorithm, we get

$$\begin{aligned}\mathsf{ena}(p) \equiv\ & p \textbf{ in } \{22\ldots46\}\\ & \wedge \big(p \textbf{ in } \{27,46\} \Rightarrow \mathtt{turn}[\mathsf{kk}.p]\big)\\ & \wedge \big(p \textbf{ in } \{31,33\} \Rightarrow \mathtt{cc}[\mathsf{thr}.p]\big)\\ & \wedge \big(p \textbf{ in } \{36,44\} \Rightarrow \mathtt{dw}[\mathsf{thr}.p]\big).\end{aligned}$$

Note that, by the definition of fwd, thread p is disabled when it is idle (at line 21). When it is at a flickering location like 22 or 39, thread p is not disabled because it can go to the next label in the actual assignment, which is a forward step.

We define a state to be *in immediate deadlock* when there are competing threads and none of these can do a forward step. Absence of immediate deadlock is a safety property.

4.6. The proof of absence of immediate deadlock

For the first step of the proof of absence of immediate deadlock, we claim the easy invariants:

Lq7:	$q \textbf{ in } \{21\ldots46\}$,
Lq0:	$\mathtt{dw}[q] \Rightarrow q \textbf{ in } \{22\ldots25, 39\}$,
Lq1:	$\mathtt{cc}[q] \Rightarrow q \textbf{ in } \{28\ldots30\} \cup \{32\ldots40\}$,
Lq2:	$q \textbf{ at } 33 \Rightarrow \mathsf{thr}.q \in \mathsf{higher}.q$,
Lq3:	$r \in \mathsf{higher}.q \Rightarrow q < r$.

Indeed, they are inductive: they hold initially and are preserved in every step. Note that Lq0 and Lq1 are implications, not equivalences: due to flickering, $\mathtt{dw}[q]$ can be true when q is at 22, and it can be false when q is at 25 (etc.).

Lemma 1. *Assume that all threads are not enabled. Then all threads are at lines 21 or 27 or 46.*

Proof. By Lq7 and condition ena, all threads are at one of the lines 21, 27, 31, 33, 36, 44, or 46. By Lq0, it follows that $\mathtt{dw}[q]$ is false for all threads. It follows that there are no threads at lines 36 and 44. If there are threads q for which $\mathtt{cc}[q]$ holds, all of these are at 33 because of the invariant Lq1. Let q_1 be the highest number q with $\mathtt{cc}[q]$. Then q_1 is a thread at line 33, and it is not enabled. It therefore satisfies $\mathtt{cc}[t_1]$ for $t_1 = \mathsf{thr}.q_1$. By the invariants Lq2 and Lq3, we have $q_1 < t_1$. As $\mathtt{cc}[t_1]$ holds, this contradicts the maximality of q_1 with $\mathtt{cc}[q_1]$. This implies that $\neg\,\mathtt{cc}[q]$ holds for all threads q.

It follows that no thread can be at 31 or 33. This proves that all threads are at 21, 27, or 46. □

In view of Lemma 1, the main danger of immediate deadlock is at line 27, at the interplay between $\mathtt{turn}$ and $\mathsf{copy}.p$. Absence of immediate deadlock thus relies on the order in which different competing threads conclude their loop at line 23 where $\mathsf{copy}.p$ gets its value.

To argue about this order, we have introduced the private integer history variables $\mathsf{cnt}.p$ for all threads p, and the shared history variable $\mathtt{shacnt}$. These variables are updated in the final step of loop 23 as given in Fig. 4. We initialize $\mathtt{shacnt} := 1$ and $\mathsf{cnt}.p := 0$ for all threads p. These variables can be unbounded, and can be modified in one atomic command, because they are only history variables.

The remainder of the proof of absence of immediate deadlock relies on the invariants:

Lq4: q **at** 27 $\Rightarrow$ kk.$q \in$ copy.q,
Lq5: $(q, e) \in \texttt{turn} \Rightarrow (q$ **in** $\{24 \ldots 34\} \wedge$ nx.$q = e)$
$\vee\, q$ **in** $\{39 \ldots 41\} \vee (q$ **at** $42 \wedge e = 1)$,
Lq6: q **in** $\{24 \ldots\} \wedge (r,$ nx.$r) \in$ copy.$q \wedge r$ **in** $\{\ldots 38\} \Rightarrow$ cnt.$r <$ cnt.q.

Indeed, predicates Lq4 and Lq5 are easily seen to be inductive. Note that, due to flickering of turn, $(q, e) \in \texttt{turn}$ can be true when q is at 24, and false when q is at 34.

Since the private variable cnt is set to the shared variable shacnt in the step towards 24, the consequent of Lq6 expresses that the most recent step of thread q towards 24 is later than the most recent step of r towards 24. We postpone the proof of Lq6 because it is more difficult. We first prove that these invariants imply absence of deadlock:

Theorem 2. *Immediate deadlock does not occur.*

Proof. Assume that none of the threads can do a forward step, and that there are competing threads. Then, by Lemma 1, all threads are at 21, 27, or 46. Therefore every competing thread q is at 27 or 46 and has turn[kk.q] because of ena(q). Therefore Lq5 implies that there are threads at label 27.

Let q_0 be the thread at 27 with the smallest value for cnt.q. Write kk.$q_0 = 2r + e$ with $r \in$ Thread and $e \in$ Bit. Then we have $(r, e) \in \texttt{turn}$ and $(r, e) \in$ copy.q_0 by Lq4. By Lq5, thread r is in $\{24 \ldots 34\}$ and $e =$ nx.r. It follows that r is at 27. By Lq6, we have cnt.$r <$ cnt.q_0. This contradicts minimality of cnt.q_0. This proves that there are no competing threads at all. □

4.7. Remaining invariants

We still have to prove invariance of Lq6. For this purpose, we need some more invariants. We present them in a bottom-up fashion. We first claim the easy inductive invariants:

Mq0: cnt.$r <$ shacnt,
Mq1: q **in** $\{23, 24\} \Rightarrow$ dw[q].

We next claim a complicated invariant that expresses that the contents of copy.q are not "too outdated" when thread q is in $\{23, 24\}$:

Mq2: q **in** $\{23, 24\} \wedge (r, e) \in$ copy.$q \wedge e \neq$ nx.$r \wedge r \in \{\ldots 38\}$
$\Rightarrow r$ **in** $\{35, 36\} \wedge q \in$ dwset.r.

The second conjunct of the consequent of Mq2 is included because it helps in the proof of invariance of the first conjunct. Predicate Mq2 holds initially because then q is at 21. It is threatened only at the lines 22, 23, and 36. It is preserved at line 22 because Kq3 implies that copy.q is empty at line 22. It is preserved at line 23 by the update copy.$q[k] :=$ turn[k] because of Lq5. It is preserved at line 36 because of Mq1.

We further delimit the outdatedness of copy.q in the invariant:

Mq3: $(r, e) \in$ copy.$q \wedge r$ **in** $\{22 \ldots 34\} \Rightarrow e =$ nx.$r \vee q \in$ predec[r].

Predicate Mq3 holds initially because copy.q is empty. It is threatened at the lines 21, 23, and 37. It is preserved by the update copy.$q :=$ turn[k] in line 23 because of Lq5. It is preserved by the modification of predec in line 37 because Kq3 implies that q is not at 37.

Preservation of Mq3 when thread r leaves line 21 is more complicated. At this point, predec[r] gets the new value central. The antecedent of Mq3 implies by Kq3 and Mq2 that thread q is in $\{25 \ldots 26\}$. Therefore, by Kq0, we have $q \in$ central.

The invariants Mq3 and FCFS together imply the derived invariant:

Mq3A: r **at** $34 \wedge (r, e) \in$ copy.$q \Rightarrow e =$ nx.r.

We next claim the invariant:

Mq4: $(r,$ nx.$r) \in$ copy.$q \Rightarrow r$ **in** $\{24 \ldots 34, 39 \ldots\}$.

Predicate Mq4 holds initially because then copy.q is empty. It is threatened at lines 23 and 34. It is preserved by the update copy.$q[k] :=$ turn[k] in line 23 because of Lq5. It is preserved by the modification of nx.r in line 34 because of Mq3A.

We can finally prove the predicate Lq6 is an invariant. It holds initially because then q is at line 21. It is threatened at lines 23 and 34. At line 23, it is preserved under the step to line 24 by thread q because of Mq0, and by the step to line 24 of r because of Mq4. It is preserved at line 34 because of Mq3A.

In the proofs of Lq6, Mq2, Mq4, we silently ignored the step from line 45 to line 21. For this step, we need the additional invariant

Rq0: $(r, e) \in \mathsf{copy}.q \ \wedge\ r \ \mathbf{in}\ \{45, 46\} \ \Rightarrow\ (q, \mathsf{nx}.q) \in \mathsf{turnset}.r$.

Preservation of Rq0 is proved by means of the invariants

Rq1: $(r, e) \in \mathsf{copy}.q \ \wedge\ r \ \mathbf{in}\ \{43 \ldots\} \ \Rightarrow\ (q, \mathsf{nx}.q) \in \mathtt{turn} \ \vee\ q \in \mathsf{dwset}.r$,
Rq2: $q \ \mathbf{in}\ \{45, 46\} \ \Rightarrow\ \mathsf{dwset}.q = \emptyset$,
Rq3: $(r, e) \in \mathsf{copy}.q \ \wedge\ r \ \mathbf{in}\ \{43 \ldots\} \ \wedge\ q \ \mathbf{in}\ \{\ldots 24\} \ \Rightarrow\ q \in \mathsf{dwset}.r$.

This concludes the proof of the invariance of Lq6 and hence of Theorem 2. Note that Rq0 indicates the need for the loop at 45–46, while Rq3 indicates the need for the loop at 43–44.

We conclude with some invariants needed in the proof of progress in Section 5:

Nq0: $q \ \mathbf{in}\ \{30, 31\} \ \Rightarrow\ \mathsf{thr}.q \in \mathsf{lower}.q$,
Nq1: $r \in \mathsf{lower}.q \ \Rightarrow\ r < q$,
Nq2: $q \ \mathbf{at}\ 36 \ \Rightarrow\ \mathsf{thr}.q \in \mathsf{dwset}.q$,
Nq3: $\mathsf{turnset}.q \cap \mathsf{copy}.q = \emptyset$.

The first three are obvious. Preservation of Nq3 at line 21 follows from Kq3.

5. Liveness: throughput and lockout freedom

In the algorithm of Fig. 2, some thread, say p, can be forced repeatedly to jump back from 31 to 28, perhaps only because some thread $q < p$ is doing a flickering assignment to $\mathtt{cc}[q]$. Therefore, even though we have proved absence of immediate deadlock, we still need a careful proof of liveness.

5.1. Preparation

In fact, the assumption of weak fairness is not enough to prove liveness because there are two obstructions. Firstly, when some thread q infinitely often fails, its value $\mathtt{cc}[q]$ can remain flickering eternally. Then, even strong fairness is not enough to imply progress for a thread $r > q$ in the cycle 28–31 because thread r can be sent back to line 28 infinitely often.

We can evade this obstruction by assuming that the number of failures is bounded. We prefer however to give a more quantitative relationship between the number of failures, the number of successful returns to the noncritical section, and the forward steps of the algorithm.

The second obstruction is that, if a thread q is in the recovery phase 43–46 waiting at line 44 for the flag $\mathtt{dw}[r]$ or at line 46 for the flag $\mathtt{turn}[2r + e]$, the responsible thread r may make unbounded progress and repeatedly set and reset the flag thread q is waiting for. Weak fairness therefore allows thread q to remain waiting indefinitely. Strong fairness at the lines 44 and 46, however, implies that thread q eventually passes the flag. We could try and adapt the algorithm by checking the unbounded progress of thread r, but this would almost certainly require an additional shared bit, and it would complicate the algorithm unnecessarily. In view of this second scenario, we exclude the recovery phase 43–46 from our progress considerations.

In order to measure general progress, we give every thread q a private variable $\mathsf{inc}.q$ that holds the number of times thread q has completed its main loop or the first part of fault recovery, or that it fails in the fragment 40–42. This variable is therefore incremented in line 38 of Fig. 4 and line 42 of Fig. 5, and under conditions in the failure step itself. General progress is measured by the sum $\mathsf{Inc} = \sum_q \mathsf{inc}.q$.

In order to show that Inc grows, and to relate the speed of growing with the forward steps of the algorithm, we introduce a more precise measure of progress, which is, roughly speaking, proportional to the number of forward steps.

The speed of progress of a concurrent system can depend on congestion. We quantify this for the present algorithm as follows. Let us bound the congestion by assuming that the number of concurrently competing threads has a fixed upper bound K with $K \leq N$. We use the derived constants $\mathsf{KN3} = K \cdot (N + 3)$ and $A = \mathsf{KN3} + 12 \cdot N + 21$. KN3 appears in the next section, A is used in Section 5.3.

5.2. Trap and cycle

In order to analyze progress for threads in 28–31, we define the set-valued state functions trap and cycle by

$$\mathsf{trap} = \big\{q \ \big|\ q \ \mathbf{at}\ 31 \wedge \mathsf{thr}.q \ \mathbf{in}\ \{32 \ldots 37\}\big\},$$
$$\mathsf{cycle} = \big\{q \ \big|\ q \ \mathbf{in}\ \{28 \ldots 31\} \wedge q \notin \mathsf{trap}\big\}.$$

If thread q is in trap, it has $\mathtt{cc}[\mathsf{thr}.q]$ because of Iq0. Thread q therefore only exits from trap when $\mathsf{thr}.q$ moves to line 38, or when q or $\mathsf{thr}.q$ fails and goes to line 39. To eliminate these cases, we define the step relation:

$$\mathsf{threshold}(p) = \big\{(x, y) \ \big|\ x.\mathsf{pc}.p \leq 37 \wedge y.\mathsf{pc}.p \in \{38, 39\}\big\}.$$

Then any inclusion $q \in \mathsf{trap}$ is kept valid in any step which is not a threshold step.

Lemma 2. *Let* cycle *be nonempty and let r be the least element of* cycle*. Assume that the algorithm takes a step which is not a threshold step, nor a forward step from 26 to 28 or from 29 to 32. Then thread r remains the least element of* cycle *or thread r enters* trap*.*

Proof. First assume that the step is done by a thread $p \neq r$. We have that thread p does not enter cycle, because it is not a step from 26 to 28, and it is not a step from trap to cycle because it is not a threshold step. Therefore, the set cycle does not grow. It follows that thread r remains its least element unless it leaves cycle. Thread r does not leave cycle by the step from 29 to 32. The only alternative is that r enters trap. □

In the setting of the lemma, assume that thread r chooses an element $\mathsf{thr}.r$ in line 28. Then the invariants Nq0 and Nq1 imply $\mathsf{thr}.r < r$ so that $\mathsf{thr}.r \notin \mathsf{cycle}$. If moreover $\mathtt{cc}[\mathsf{thr}.r]$ holds, Lq1 implies that $\mathsf{thr}.r$ is in 32–37. It follows that the forward steps of thread r either move r into trap, or make $\mathsf{lower}.r$ empty and move r toward line 32. We quantify this by means of the function

$$\begin{aligned}\mathsf{cvf}(r) = (\ & r\ \mathbf{at}\ 28\ ?\ 2\\ & : r\ \mathbf{at}\ 29\ ?\ r+3-\#\mathsf{lower}.r\\ & : r\ \mathbf{at}\ 30 \wedge \mathsf{thr}.q\ \mathbf{in}\ \{32\ldots37\}\ ?\ r+3\\ & : \mathsf{pc}.r-30),\end{aligned}$$

where $\#S$ stands for the number of elements of the set S. Note that $0 \le \mathsf{cvf}(r) < N+3$ whenever $r \in \mathsf{cycle}$.

In the measure Cvf of the cycle, we combine the number of elements of trap (multiplied by $N+3$) and the value of cvf at the least thread in cycle. The latter value is taken to be 0 when cycle is empty. We take $\mathsf{Cvf} = 0$ if nexit does not hold, where nexit is the condition

$$\mathsf{nexit} \equiv \big(\forall q : q\ \mathbf{notin}\ \{38\ldots42\}\big).$$

All this culminates in the definition:

$$\begin{aligned}\mathsf{Cvf} = \big(&\mathsf{nexit}\ ?\\ & (N+3)\cdot\#\mathsf{trap} + \big(\mathsf{cycle} \neq \emptyset\ ?\ \mathsf{cvf}\big(\min(\mathsf{cycle})\big) : 0\big)\\ &: 0\big).\end{aligned}$$

As trap and cycle are disjoint sets with together at most K elements, we have $0 \le \mathsf{Cvf} < \mathsf{KN3}$. The value of Cvf changes only in cycle steps and in the steps from 26 to 28, or 29 to 32, or when the truth value of nexit changes.

5.3. Throughput

We first introduce a threadwise measure avf that is only modified by steps of thread q, and that ignores the steps in 28–31 and 43–46:

$$\begin{aligned}\mathsf{avf}(q) &= A\cdot\mathsf{inc}.q + \big(q\ \mathbf{in}\ \{22\ldots42\}\ ?\ \mathsf{bvf}(q) : 0\big), \quad \text{where}\\ \mathsf{bvf}(q) &= \big(\ \mathsf{pc}.q \le 27\ ?\ \mathsf{pc}.q-21\\ &\quad : \mathsf{pc}.q \le 31\ ?\ N+8 : 2\cdot N+\mathsf{pc}.q-23\big)\\ &\quad + 3\cdot(\mathsf{pc}.q > 21\ ?\ 2\cdot N - \#\mathsf{turnset}.q : 0) - 2\cdot\#\mathsf{copy}.q\\ &\quad + 2\cdot(\mathsf{pc}.q \ge 32\ ?\ N-\#\mathsf{higher}.q : 0)\\ &\quad + 2\cdot(\mathsf{pc}.q \ge 35\ ?\ N-\#\mathsf{dwset}.q : 0)\\ &\quad + (\mathsf{pc}.q \ge 38\ ?\ \mathsf{KN3} : 0).\end{aligned}$$

Roughly speaking, $\mathsf{bvf}(q)$ is the distance of the current state of q from the latest idle state, and A is an upper bound for the distance from an idle state to the next idle state. Therefore $\mathsf{avf}(q)$ is an estimate of the distance from the beginning of the process to its current state. Indeed, using Lq7 and Nq3, we prove that $0 \le \mathsf{bvf}(q) < A$. It follows that $A\cdot\mathsf{inc}.q \le \mathsf{avf}(q) < A\cdot(\mathsf{inc}.q+1)$.

The number $\mathsf{avf}(q)$ never decreases, and it increases under most forward steps of q. As $\mathsf{avf}(q)$ does not increase under all forward steps of q, we partition the class of forward steps in three parts. The forward steps from locations in 43–46 are called *reentrant steps*. The forward steps at the lines 28, 30, 31, and the steps from line 29 to lines 28 or 29 are called *cycle steps*. We define the remaining forward steps to be *noncycle steps*.

The number $\mathsf{avf}(q)$ increases in all noncycle steps of q. At the lines 27, 33, 36, this follows from the invariants Lq4, Lq2, Nq2, respectively. The number $\mathsf{avf}(q)$ increases with $N+3$ when thread q jumps from 26 to 28 or from 29 to 32. It increases with at least $\mathsf{KN3}+1$ when q executes line 37, or a failure step from any line $\neq 38$. These bigger incrementations are brought in to compensate changes in the measure for the cycle to be introduced below.

As our first target is the throughput of the algorithm, we form the sum $\mathsf{Avf} = \sum_q \mathsf{avf}(q)$. This sum never decreases because its summands never decrease. It increases in all noncycle steps of any thread. It increases with $N+3$ when some thread jumps from 26 to 28 or from 29 to 32. It increases with at least $\mathsf{KN3}+1$ when some thread executes line 37 or fails from any line $\neq 38$.

We define the global measure by $\mathsf{Gvf} = \mathsf{Avf} + \mathsf{Cvf}$. The first result is that progress of Inc is proportional to progress of Gvf:

$$A \cdot \mathsf{Inc} \leq \mathsf{Gvf} \leq A \cdot (\mathsf{Inc} + N). \tag{3}$$

This result is based on the invariants Lq7, Kq2, Kq3, and Nq3.

The next result is that Gvf never decreases. At line 37, Gvf increases because Avf increases with KN3 and $\mathsf{Cvf} < \mathsf{KN3}$. In a jump from 26 to 28 or from 29 to 32, Gvf increases, because Avf increases with $N+3$, while $\mathsf{cvf}(r) < N+3$ and #trap remains constant. It follows that Gvf increases in every noncycle step.

On the other hand, if nexit holds, cycle is nonempty, and r is the least element of cycle, then any cycle step of thread r also increases Gvf. Note that this includes the step from line 30 to 31 when thread r exits cycle and enters trap. Here we use the factor $N+3$ in the definition of Cvf, because min(cycle) changes. In the step of line 29, we use that r is the least thread of cycle together with the invariants Lq1, Nq0, and Nq1. Using all this, we prove the first UNITY proposition

$$r = \min(\mathsf{cycle}) \wedge \mathsf{nexit} \wedge k \leq \mathsf{Gvf} \quad \textbf{ensures} \quad k+1 \leq \mathsf{Gvf}.$$

Note that, in such a formula, we use universal quantification over all free variables, e.g., in this case over r and k.

Using the **ensures** rule and the disjunction rule of Section 3.3, we obtain

$$\mathsf{cycle} \neq \emptyset \wedge \mathsf{nexit} \wedge k \leq \mathsf{Gvf} \quad \mathbf{Lt}\langle 1\rangle \quad k+1 \leq \mathsf{Gvf}. \tag{4}$$

We now come back to the progress by noncycle steps. For this purpose, we introduce a predicate $\mathsf{Aen}(q)$ (avf-enabling) with three crucial properties: firstly, it implies that a noncycle step of thread q is enabled; secondly, the only steps that invalidate $\mathsf{Aen}(q)$ are steps of thread q itself which increment $\mathsf{avf}(q)$; thirdly, it is insensitive to flickering steps. Predicate $\mathsf{Aen}(q)$ is defined by

$$\begin{array}{ll} \mathsf{Aen}(q): & q \;\textbf{in}\; \{22\ldots 27\} \cup \{32\ldots 42\} \\ \wedge & (q \;\textbf{at}\; 27 \Rightarrow \neg\,\mathtt{turn}[\mathsf{kk}.q] \wedge \mathsf{kk}_1.q \;\textbf{notin}\; \{24, 34\} \wedge \mathsf{nexit}) \\ \wedge & (q \;\textbf{at}\; 33 \Rightarrow \neg\,\mathtt{cc}[\mathsf{thr}.q] \wedge \mathsf{thr}.q \notin \mathsf{cycle} \wedge \mathsf{nexit}) \\ \wedge & (q \;\textbf{at}\; 36 \Rightarrow \mathsf{thr}.q \;\textbf{notin}\; \{22\ldots 25, 39\}), \end{array}$$

where $\mathsf{kk}_1.q$ is the thread that owns the pair $\mathsf{kk}.q$.

By the above results, using Lq0 at line 36, we have

$$\mathsf{Aen}(q) \wedge k \leq \mathsf{Gvf} \quad \textbf{ensures} \quad k+1 \leq \mathsf{Gvf}. \tag{5}$$

As above, this implies

$$\big(\exists q : \mathsf{Aen}(q)\big) \wedge k \leq \mathsf{Gvf} \quad \mathbf{Lt}\langle 1\rangle \quad k+1 \leq \mathsf{Gvf}. \tag{6}$$

We do not expect progress when all threads are idle or reentrant. We thus define

$$\mathsf{AI} \equiv \big(\forall q : \mathsf{pc}.q \notin \{22\ldots 42\}\big).$$

We claim:

$$\neg\mathsf{AI} \equiv (\mathsf{cycle} \neq \emptyset \wedge \mathsf{nexit}) \vee \big(\exists q : \mathsf{Aen}(q)\big). \tag{7}$$

In fact, it is easy to see that the right-hand side implies the left-hand side. Conversely, assume that the right-hand side is false. Using the definition of Aen, this implies that there are no threads in $\{22\ldots 26\}$. Consequently, there are no threads in $\{34\ldots 38\}$, and no threads at 32. In particular nexit holds. Therefore cycle is empty. If there is a thread at 33, let q be the greatest thread at 33; then $\neg\,\mathsf{Aen}(q)$ implies that $\mathtt{cc}[\mathsf{thr}.q]$ holds while cycle is empty. Using Lq1, Lq2, and Lq3, we obtain a contradiction with the maximality of q. This shows that all threads are at 21 or in $\{27\ldots 31\}$. This implies that trap is empty. Consequently, all threads are at 21 or at 27. If there are threads at 27, let q the thread at 27 with the least value of $\mathsf{cnt}.q$. From this one derives a contradiction with the invariants Lq4, Lq5, Lq6 in the same way as in the proof of Theorem 2.

By formula (7) and the disjunction rule, the formulas (4) and (6) imply

$$\neg\mathsf{AI} \wedge k \leq \mathsf{Gvf} \quad \mathbf{Lt}\langle 1\rangle \quad k+1 \leq \mathsf{Gvf}.$$

From this one can derive

$$k \le \mathsf{Gvf} \quad \mathbf{Lt}\langle 1\rangle \quad \mathsf{AI} \vee k+1 \le \mathsf{Gvf}. \tag{8}$$

Using transitivity, we then obtain

$$k \le \mathsf{Gvf} \quad \mathbf{Lt}\langle n\rangle \quad \mathsf{AI} \vee k+n \le \mathsf{Gvf}. \tag{9}$$

Substituting $k := A \cdot k$ and $n := A \cdot (n+N)$ in (9) and using formula (3), we get general progress: while there are competing threads, the measure Inc keeps growing.

Theorem 3. $k \le \mathsf{Inc}\ \mathbf{Lt}\langle A \cdot (n+N)\rangle\ \mathsf{AI} \vee k+n \le \mathsf{Inc}$.

5.4. Lockout freedom

For the proof of progress of the individual threads, we partition the competing region in three parts: $\mathsf{doorway} = \{22 \ldots 24\}$, $\mathsf{central} = \{25 \ldots 37\}$, and line 38.

As we do not expect progress while there is a failing thread, we define

$$\mathsf{failing} \equiv (\exists q : q\ \mathbf{at}\ 39).$$

To prove progress in the doorway, we use the predicate

$$U(q,n): \quad q\ \mathbf{in}\ \mathsf{doorway} \wedge 25 - \mathsf{pc}.q + \#\mathsf{turnset}.q \le n.$$

We then have

$$U(q,n+1)\ \mathbf{ensures}\ U(q,n) \vee q\ \mathbf{in}\ \mathsf{central} \vee \mathsf{failing}.$$

As $q\ \mathbf{in}\ \mathsf{doorway}$ is equivalent to $U(2 \cdot N + 3)$, and $U(q,0)$ is false, by transitivity this implies that

$$q\ \mathbf{in}\ \mathsf{doorway} \quad \mathbf{Lt}\langle 2 \cdot N + 3\rangle \quad q\ \mathbf{in}\ \mathsf{central} \vee \mathsf{failing}. \tag{10}$$

At line 38, it suffices to note that

$$q\ \mathbf{at}\ 38 \quad \mathbf{ensures} \quad q\ \mathbf{at}\ 21 \vee \mathsf{failing}. \tag{11}$$

The proof of progress from $\mathsf{central}$ to line 38 is based on the observation that individual progress is implied by general progress in conjunction with FCFS. More precisely, we have that, while thread q is in $\mathsf{central}$, another thread r can increment $\mathsf{inc}.r$, but its progress is bounded because r gets $q \in \mathsf{predec}[r]$. We therefore define

$$\mathsf{cinc}(r,q) = \mathsf{inc}.r + \big(r\ \mathbf{in}\ \{22 \ldots 42\} \wedge q \notin \mathtt{predec}[r]\ ?\ 1 : 0\big).$$

The value of $\mathsf{cinc}(r,q)$ never decreases. While q is in $\mathsf{central}$, $\mathsf{cinc}(r,q)$ does not change unless q goes to line 38 or a failure occurs. The proof of this uses the invariant $\mathsf{Nq4}$ of Section 4.7. It follows that the sum $\mathsf{Cinc}(q) = \sum_r \mathsf{cinc}(r,q)$ satisfies, for any number k,

$$q\ \mathbf{in}\ \mathsf{central} \wedge \mathsf{Cinc}(q) = k\ \mathbf{unless}\ q\ \mathbf{at}\ 38 \vee \mathsf{failing}. \tag{12}$$

On the other hand, we clearly have $\mathsf{inc}.r \le \mathsf{cinc}(r,q) \le \mathsf{inc}.r + 1$. As the number of competing threads is bounded by K, it follows that $\mathsf{Inc} \le \mathsf{Cinc}(q) \le \mathsf{Inc} + K$. We now apply the PSP rule to Theorem 3 with $k := k - K$ and $n := K + 1$, and formula (12). As $q\ \mathbf{in}\ \mathsf{central}$ contradicts AI, this yields

$$q\ \mathbf{in}\ \mathsf{central} \wedge \mathsf{Cinc}(q) = k \quad \mathbf{Lt}\big\langle A \cdot (K + N + 1)\big\rangle \quad q\ \mathbf{at}\ 38 \vee \mathsf{failing}. \tag{13}$$

By disjunction over k, this gives

$$q\ \mathbf{in}\ \mathsf{central} \quad \mathbf{Lt}\big\langle A \cdot (K + N + 1)\big\rangle \quad q\ \mathbf{at}\ 38 \vee \mathsf{failing}. \tag{14}$$

Combining this result with the formulas (10) and (11), we finally obtain the main result for individual progress:

Theorem 4. $q\ \mathbf{in}\ \{22 \ldots 38\}\ \mathbf{Lt}\langle A \cdot (K + N + 1) + 2 \cdot N + 4\rangle\ q\ \mathbf{at}\ 21 \vee \mathsf{failing}$.

6. Conclusions

The algorithm described here is more elegant than the 5-bits algorithm of [21] and the 4-bits algorithm of [15]. In [21], the waiting conditions are more complicated, this would add complications to the proof. The differences with the algorithm of [15] are marginal and do not affect the difficulty of the verification. Indeed, the proofs of safety, i.e., of mutual exclusion, FCFS, and absence of immediate deadlock, given here are completely analogous to those given in [15].

The proof of progress with unity logic given in Section 5, is just as complicated as the corresponding behavioral proof we gave in [15]. It does give more information, however, because it enables us to give explicit progress bounds in Theorems 3 and 4. In the behavioral proof of [15], we just go to the limit, and investigate an infinite behavior in which eventually some thread makes no progress anymore. This leads to a contradiction and thus proves progress, but it gives no indication of the speed of progress.

As suggested by a referee, it should be possible to prove analogues of Theorems 3 and 4 for most other mutual exclusion algorithms. For instance, for the tournament algorithm of [7, Section 18.6], we conjecture that Theorem 3 holds with a constant A proportional to $\log N$, and that Theorem 4 holds with a progress bound proportional to N. A drawback of bounded UNITY is that the progress bounds obtained are only upper bounds. It seems to require extensive operational reasoning and difficult scenarios to obtain lower bounds for the number of rounds needed.

We could not have obtained the results in this paper without the proof assistant PVS [24]. Our proof script is available at [16]. The manually written part of it has around 1300 lines for the results of Section 4, 300 lines for the rules and soundness of bounded UNITY, and 500 lines for Section 5.

Indeed, anyone who needs to verify a concurrent algorithm is strongly advised to use a proof assistant like PVS, Isabelle, or Coq. In our experience, even the mental effort of pondering about the problem of how to verify such an algorithm with a proof assistant can uncover flaws in the algorithm. The verification of an algorithm takes a lot of work, but it leads to improved understanding and confidence. The methods and experiences of using a proof assistant for concurrency verification were discussed extensively in our paper [14] about Lamport's Bakery Algorithm. The treatment of progress with bounded UNITY in the present paper, however, is more elegant than the approach via temporal logic in [14,15].

Next to the algorithm discussed in this paper, Aravind [5] also proposes a "fair" algorithm, where fairness is meant with the technical meaning that, if threads q and r are competing concurrently, both have 50% probability of winning the tie-break. The idea is to give every thread two slots in the range of thread identifiers. We can see that this idea can be implemented in a sound way, in an algorithm that, in total, use 5 safe shared bits per thread. Aravind [5], however, suggests in his HF-FCFS algorithm a 4-bit version, which in terms of our version of the algorithm (Fig. 4) requires communication of the private variables $\mathrm{nx}.p$. We do not see how this can be done in time in a reliable way.

References

[1] J.H. Anderson, M.G. Gouda, Atomic semantics of nonatomic programs, Inf. Process. Lett. 28 (1988) 99–103.

[2] J.H. Anderson, Y.J. Kim, T. Herman, Shared-memory mutual exclusion: major research trends since 1986, Distrib. Comput. 16 (2003) 75–110.

[3] G.R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison–Wesley, Reading, etc., 2000.

[4] K.R. Apt, F.S. de Boer, E.-R. Olderog, Verification of Sequential and Concurrent Programs, Springer, New York, 2009.

[5] A. Aravind, Simple, space-efficient, and fairness improved FCFS mutual exclusion algorithms, J. Parallel Distrib. Comput. 73 (2013) 1029–1038.

[6] A.A. Aravind, W.H. Hesselink, Nonatomic dual bakery algorithm with bounded tokens, Acta Inform. 48 (2011) 67–96.

[7] P.A. Buhr, D. Dice, W.H. Hesselink, High-performance N-thread software solutions for mutual exclusion, Concurr. Comput. (2014) 1–51, http://dx.doi.org/10.1002/cpe.3263.

[8] J.E. Burns, Complexity of communication among asynchronous parallel processes, Ph.D. thesis, School of Information and Computer Science, Georgia Institute of Technology, 1981.

[9] K.M. Chandy, J. Misra, Parallel Program Design, A Foundation, Addison–Wesley, 1988.

[10] E.W. Dijkstra, Solution of a problem in concurrent programming control, Commun. ACM 8 (1965) 569.

[11] E.W. Dijkstra, Co-operating sequential processes, in: F. Genuys (Ed.), Programming Languages (NATO Advanced Study Institute), Academic Press, London, etc., 1968, pp. 43–112. Also EWD123, 1965.

[12] W.H. Hesselink, Progress under bounded fairness, Distrib. Comput. 12 (1998) 197–207.

[13] W.H. Hesselink, Complete assertional proof rules for progress under weak and strong fairness, Sci. Comput. Program. 78 (2013) 1521–1537, http://dx.doi.org/10.1016/j.scico.2012.10.013.

[14] W.H. Hesselink, Mechanical verification of Lamport's bakery algorithm, Sci. Comput. Program. 78 (2013) 1622–1638.

[15] W.H. Hesselink, Verifying a simplification of mutual exclusion by Lycklama–Hadzilacos, Acta Inform. 50 (2013) 297–329.

[16] W.H. Hesselink, Mutual exclusion by Lycklama–Hadzilacos, and Aravind, PVS scripts, http://wimhesselink.nl/mechver/mx4bits, 2014.

[17] T. Inoue, T. Hironaka, T. Sasaki, S. Fukae, T. Koide, H.J. Mattausch, Evaluation of bank-based multiport memory architecture with blocking network, Electron. Commun. Jpn. 89 (2006) 498–510.

[18] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Commun. ACM 17 (1974) 453–455.

[19] L. Lamport, The mutual exclusion problem – part I: a theory of interprocess communication, part II: statement and solutions, J. ACM 33 (1986) 313–348.

[20] L. Lamport, On interprocess communication. Parts I and II, Distrib. Comput. 1 (1986) 77–101.

[21] E.A. Lycklama, V. Hadzilacos, A first-come-first-served mutual-exclusion algorithm with small communication variables, ACM Trans. Program. Lang. Syst. 13 (1991) 558–576.

[22] J. Misra, A Discipline of Multiprogramming: Programming Theory for Distributed Applications, Springer, New York, 2001.

[23] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Inform. 6 (1976) 319–340.
[24] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert, PVS version 2.4, system guide, prover guide, PVS language reference, http://pvs.csl.sri.com, 2001.
[25] M. Raynal, Algorithms for Mutual Exclusion, MIT Press, 1986.
[26] W.-T. Shiue, C. Chakrabarti, Multi-module multi-port memory design for low power embedded systems, Des. Autom. Embed. Syst. 9 (2004) 235–261.
[27] G. Taubenfeld, Synchronization Algorithms and Concurrent Programming, Pearson Education/Prentice Hall, 2006.
[28] W. Zuo, Z. Qi, L. Jiaxing, An intelligent multi-port memory, in: Proc. of the IEEE Intl. Symp. on Information Technology Application Workshops, 2008, pp. 251–254.