Verifying a Simplification of Mutual Exclusion by Lycklama-Hadzilacos

Wim H. Hesselink (whh442, February 24, 2013)

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

Appears in Acta Informatica. The final publication will be available at link.springer.com.

Abstract. A simplification of the mutual exclusion algorithm of Lycklama and Hadzilacos (1991) is presented. It uses only four nonatomic shared bits per thread to guarantee mutual exclusion with the first-come-first-served property. The algorithm is verified by assertional methods, aided by the proof assistant PVS. A variation with five bits per thread is also given. This variation may give better performance when the number of threads is large. The use of the proof assistant made it easy to transfer the proof of the main algorithm to the variation.

Keywords: concurrent algorithm, mutual exclusion, atomicity, assertional method, verification, proof assistant.

1 Introduction

The advent of multiprocessors and multicore architectures has raised the interest in the correctness of concurrent algorithms because, as is well known, concurrent algorithms can unexpectedly misbehave due to subtle bugs or race conditions. They are almost impossible to test, and difficult to verify.

Testing and model checking are important methods to find errors, but, for concurrent algorithms, they are in most cases not able to ensure correctness. In verifications on the other hand, the concurrency usually induces many case distinctions, even when attractive conceptualizations are available. The verification, therefore, sometimes seems to hinder rather than help understanding of the algorithm.

The verification problem for concurrent algorithms has been with us for more than forty years. When Dijkstra [13] introduced mutual exclusion in 1965, he presented Dekker's algorithm for it with an extensive correctness argument. Yet, in 1974, he published his self-stabilizing systems [15] without any hint for correctness.

In recent years, the advance of mechanical theorem provers like ACL2, Coq, HOL, Isabelle, PVS has made it easier to prove concurrent algorithms exhaustively, primarily because one can safely rely on the prover for exhaustive case distinctions. Effective use of the prover for these purposes, however, requires a good understanding of the methods of concurrency verification.

The present paper illustrates the new possibilities by presenting a computer assisted verification of a simplification of the algorithm of Lycklama and Hadzilacos [46]. This simplification establishes mutual exclusion with the first-come-first-served property by means of only four nonatomic shared bits per thread.

1.1 Overview

We first give an overview of the verification methods for concurrent and distributed algorithms in Section 1.2. In Section 1.3, we indicate the relationship between these verification methods and the practices in the design of concurrent algorithms. The correctness aspects of concurrent algorithms are briefly discussed in Section 1.4. Section 1.5 presents the mutual exclusion problem, while Section 1.6 introduces the solution of Lycklama and Hadzilacos.

Section 2 presents the formal model for concurrent and distributed algorithms, and then specializes to shared memory concurrency, with possibly nonatomic variables. It concludes with the definitions of mutual exclusion (MX) and the first-come-first-served property (FCFS).

Section 3 presents our simplified algorithm with 4 shared bits. It first explains how Lycklama and Hadzilacos [46] have separated the concerns for mutual exclusion and FCFS. It then presents the new algorithm along these lines, and proves that it satisfies mutual exclusion and FCFS. Section 4 proves absence of immediate deadlock, equally important but more difficult. In Section 5, we prove lockout freedom, i.e., absence of livelock and individual starvation. More precisely, under the assumption of weak fairness [44], we prove that every thread always eventually comes back to the noncritical section.

In Section 6, we discuss some variations of the algorithm, in particular a variation with 5 shared bits that may have a better performance if the number of threads is large. Here, we also sketch the relationship with the algorithm of Lycklama and Hadzilacos [46]. In Section 7, we indicate how the correctness proof is verified with the proof assistant PVS. We conclude in Section 8.

1.2 Assertional and behavioural methods in concurrency

Methods to verify concurrent algorithms were first developed in the 1970s. This happened in parallel with and under inspiration of the methods for sequential algorithms in [29,16]. The main papers were by Ashcroft [9], Owicki and Gries [52], and Lamport [39]. The method of the *proof outlines* of [52] was the most influential one. Its main idea is that the code of the threads is annotated with assertions that determine Hoare triples specifying the actions of the threads, while possible interference by other threads is dealt with separately.

The three papers are exponents of what came to be known as the *assertional method*, in which the correctness argument primarily relies on assertions about the state and the next-state relation.

In 1979, Lamport [40] introduced the *behavioural approach*. In this approach, one can argue about the behaviours of a concurrent algorithm by introducing an ordering on the steps that are executed. In 1990, however, Lamport [43, p. 427] came back to the assertional method and abandoned his behavioural method because "despite an appearance of extreme rigour, the method ultimately reduces to the unstructured, informal reasoning of ordinary mathematics".

The main point of [43] was the decision to concentrate on a single global invariant: the conjunction of all properties that hold continuously during any program execution. In the spirit of the days around 1990, the paper introduced predicate transformers *win* and *sin* that yield the weakest invariant implying a given predicate and the strongest invariant implied by it.

A second point of [43] is the insistence that in assertional methods one should be able and willing to reckon explicitly with the control state. In the proof outlines method of [52,6], the control state is not directly available, but follows from the locations of the assertions in the proof outlines and can be made explicit by means of auxiliary variables.

In 1991, Abadi and Lamport [1] extended the assertional method with prophecy variables and refinement mappings. A refinement mapping is roughly speaking a function between the state spaces of two specifications S_1 and S_2, such that every step of S_1 maps to a step of S_2. It follows that every execution of S_1 maps to an execution of S_2. This means that S_1 "implements" S_2. Prophecy variables are auxiliary variables added to a specification to "guess" future developments of the execution. The auxiliary variables to record the past of an execution introduced by [52] (also called ghost variables) were renamed to *history variables*. In this paper, we only need history variables, no prophesies or refinement mappings.

Under some technical conditions, the paper [1] proved soundness and semantic completeness of the assertional method with history variables, prophecy variables and refinement mappings. Here semantic completeness means that every "simulation relation" can be factorized over an extension with history variables, followed by an extension with prophecy variables and a refinement mapping. In [27], we replaced the prophecy variables by eternity variables and in this way eliminated the technical conditions of [1].

1.3 Methods and practice

Methods for concurrency verification can only be judged by their relevance for good example problems. The examples should not be too small because small examples can be verified by all sound methods. They should not be too large because that makes it difficult to get an overview and special issues of the example may become dominant.

In practice, new concurrent algorithms are often presented in conference papers and the proofs are omitted due to space constraints, possibly with references to proofs in a technical report, e.g., [33,57,17,50,51].

Yet, several sizable concurrent algorithms have been proved (almost) exhaustively in the past 25 years. We list here an, obviously biased, selection of them, grouped according to the verification method applied.

- B1. Behavioural methods based on execution histories (without an explicit partial order as in [40]): an implementation of LL/SC variables [34].
- B2. Behavioural methods based on [40]: mutual exclusion with 5 safe bits [46], bounded concurrent timestamp systems [21].
- A1. Assertional methods with proof outlines: we know of no sizable case study.
- A2. Assertional methods for message passing algorithms: the GHS minimum weight spanning tree algorithm [63,25].
- A3. Assertional methods for shared memory, based on global invariants with control information: Ben Ari's garbage collector [55], wait-free linearization [23], lazy caching [37], lock-free dynamic hashtables [18], nonatomic mutual exclusion [36], lock-free concurrent garbage collection [19].
- A4. Methods like A3 extended with essential use of refinement: a serializable database interface [26], a lock-free implementation of atomicity [20], a mutual exclusion algorithm [7].

Of course, the verification method is closely linked to the specification method, as it is the specification that needs to be verified. This is especially notable for the algorithms mentioned under B2 and A4.

While we have tried to compile a fair selection above, we can leave no doubt about our preferences. With [43], we are convinced that the assertional methods A2, A3, and A4 are to be preferred because behavioural proofs are seldom reliable. The verification of any concurrent algorithm should be based on invariants, possibly augmented with history, prophecy or eternity variables, and refinement methods. This is not a matter of dogma, but of reliability and ease of comparison. Assertional methods have been successful for more than twenty years now, and complicated algorithms have been verified with these methods.

The proof of correctness of a concurrent algorithm always requires many case distinctions. Handling case distinctions can be done much more reliably by a mechanical proof assistant than by a human verifier. We use the proof assistant PVS [53] for this purpose.

Another method of concurrency verification is model checking, e.g. [11,30]. Model checking can decide the correctness of an algorithm if the state space is finite and not too large for the available machine. We prefer theorem proving over model checking because theorem proving improves the understanding of the algorithm. For a doubtful algorithm, however, model checking can often quickly reveal incorrectness.

There are several good books on distributed algorithms and concurrent programming that do give correctness proofs [5,22,47,61]. These books mostly use behavioural arguments based on [40], with occasional application of invariants. In our view, the proofs suffer from what Lamport called "the unstructured, informal reasoning of ordinary mathematics". In the vast universe of mathematics, the problem of correctness of concurrent algorithms is a very special case. The single approach by assertional methods with its three versions A2, A3, A4 gives sufficient flexibility to treat all of them. There is of course more than correctness. The books mentioned also give mathematical treatments of other theoretical issues like time complexity and consensus numbers. There are also good books on concurrency verification, e.g., [6,12,48,49], all of these using assertional methods. We are not aware of verification books that use behavioural methods.

Verification of concurrent algorithms does not really require an in-depth study of the theory of verification. One can learn much from a good case study. It is therefore important that assertional verifications of representative concurrent algorithms are available. The aim of the present paper is to provide this. For this purpose, we have chosen the mutual exclusion algorithm of Lycklama and Hadzilacos [46], because an assertional proof was lacking, and because it is relatively simple and yet illustrates the problems of larger algorithms. This is also a good opportunity to simplify the algorithm. The verification presented here belongs to the above class A3.

1.4 Safety and liveness

The verification of a concurrent algorithm has two concerns [39]: safety and liveness. *Safety* means that nothing bad happens. *Liveness* means that eventually something good happens. Safety is typically captured in invariants. The treatment of liveness is much more open, and dependent on the algorithm.

In this paper, we consider systems that offer some service to a number of threads. The threads can be idle or competing. *Competing* means interested in the service. The competing threads usually need to communicate for this. *Idle* means not interested in the service, but possibly involved in other activities.

In this context, the two main liveness properties are *deadlock freedom* and *lockout freedom* [46,22]. Deadlock freedom means that, when there are competing threads, eventually some thread gets the service. Lockout freedom means that every competing thread eventually gets the service. Of course, lockout freedom implies deadlock freedom.

In order to prove that our algorithm satisfies (deadlock freedom and) lockout freedom, we have to assume *weak fairness* for most of the steps of the algorithm. Roughly speaking, weak fairness for some step is the assumption that, if some thread from some moment onward is always able to do the step, it will eventually take it. On uniprocessor systems, a reasonable scheduler will take care of this. On multiprocessor systems, it means that all processors are fast enough to read and modify shared variables. The assumption of weak fairness is formalized by excluding the "unfair" executions from consideration. The precise treatment is postponed to Section 5.2.

1.5 Mutual exclusion

The problem that concurrent processes may need exclusive access to some shared resource was first proposed and solved in [13]. The problem came to be known as mutual exclusion in [14]. Numerous solutions to this problem have been proposed. Surveys can be found in [4,54,60].

An early and elegant solution is Lamport's bakery algorithm [38], which has two additional properties that need to be discussed. It has the first-come-first-served property (FCFS), and the shared variables used need not be atomic. The second point means that the algorithm does not assume mutual exclusion on read and write operations on shared variables. The downside is that these shared variables hold integer values that can become arbitrarily large. Several modifications to the bakery algorithm have been proposed to bound these numbers [64,2,32,35,58,62,59]. All these modifications compromise the nonatomicity property of the algorithm.

Although devising busy-waiting solutions to the mutual exclusion problem was early on mostly an academic exercise because busy waiting is inefficient on a single processor, the advent of multiprocessors and multicore architectures has spurred renewed interest in such solutions [5, p. 135].

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 [31,56,65].

Mutual exclusion without FCFS does not require much shared memory. Indeed, the algorithm of Burns-Lamport [10,41] establishes mutual exclusion for N threads with only one nonatomic shared Boolean variable per thread.

Lamport's bakery algorithm gives mutual exclusion with FCFS, using only N nonatomic shared integer variables, but these variables cannot be bounded. The algorithm of [59] uses N atomic variables with values in $[0 \dots N]$, and $N+1$ atomic shared bits. The algorithm of [8] uses N nonatomic variables with values in $[0 \dots N]$, 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 [46].

1.6 Mutual exclusion by Lycklama-Hadzilacos

The mutual exclusion algorithm of Lycklama and Hadzilacos [46] 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 paper [46] contains a behavioural correctness proof that we are unable to follow.

The presentation of the algorithm in [46] 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 [10] and Lamport [41], 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 comparison with [46], we simplify the outer algorithm, such that it has simpler waiting conditions and simpler invariants, and that it may have a somewhat better performance.

The conclusion of [46] asserts that the number of communication bits required can be reduced from five to four per thread. A proof of correctness of this modification is not given. The reduction is achieved by reusing the bit of the inner algorithm in the outer algorithm. At first sight, this appears to be an ugly trick.

We claim, however, that, if this inner bit is used in the outer algorithm, the inner algorithm makes one of the waiting conditions in the outer algorithm superfluous. This enables us to simplify the combined algorithm. The present paper substantiates this claim. It is a representative assertional verification of a concurrent algorithm with modest complexity, supported by the mechanical proof assistant PVS.

1.7 Our contributions

We simplify the algorithm of [46] by using fewer shared bits and straightening the associated await conditions. We present an assertional proof of its correctness, both safety and liveness, verified with the proof assistant PVS. Our contributions, however, are not so much in the results, but in the methods by which these results have been obtained.

2 The Model and the Repertoire

The basic model is described in Section 2.1. In Section 2.2, we extend the model with threads and their programs. Atomicity is discussed in Section 2.3. Nonatomic shared variables are introduced in Section 2.4. Mutual exclusion is introduced in Section 2.5.

2.1 The basic model

Our basic model for concurrent and distributed algorithms is a transition system which consists of a set, say X, called the *state space*, a relation $\mathit{step} \subseteq X^2$, called the *step relation*, and a subset $\mathit{init} \subseteq X$, the set of *initial* states. Following [1], we assume that the step relation is reflexive: it contains the identity relation 1_X. An *execution* is an infinite sequence of states $(x_n \mid n \in \mathbb{N})$ with $x_0 \in \mathit{init}$, and $(x_n, x_{n+1}) \in \mathit{step}$ for all $n \in \mathbb{N}$.

Usually (e.g., [1,27]), there is also a so-called *supplementary property* $P \subseteq X^\omega$ to capture fairness or liveness properties. If so, an execution that belongs to P is called a *behaviour*. In this paper, we use the supplementary property of weak fairness, but we postpone the treatment of this to Section 5.2.

A state $x \in X$ is called *reachable* if it occurs in an execution. A function on the set X is called a *state function*. A Boolean state function is called a *predicate*. A predicate is called an *invariant* if it holds in all reachable states. A predicate f is called *inductive* if it holds in all initial states and is preserved under every step, i.e.,

$$(\forall\, x \in \mathit{init} : f(x)) \;\wedge\; (\forall\, x, y : f(x) \wedge (x,y) \in \mathit{step} \;\Rightarrow\; f(y)) \;.$$

It is easy to see that every inductive predicate is an invariant. Conversely, in order to show that some predicate is an invariant, it suffices to show that it is implied by an inductive one. In this way, one can largely avoid the executions from consideration, and concentrate on the states and the step relation. This is the central aim of the assertional method.

2.2 Threads and programs

In concurrent algorithms, there is usually a set of threads that can do steps. Each thread p has its own step relation step_p to determine the steps it can make. The actual step relation step is a union $\mathit{step} = 1_X \cup \bigcup_p \mathit{step}_p$ where p ranges over the threads. The state space X is a Cartesian product of a shared state space and the private state spaces of the threads. The private state spaces of the threads are spanned by private variables. For simplicity, we assume that all threads have the same private variables. If v is a private variable, we write $v.p$ to indicate the value of v for thread p.

For algorithms with shared memory, we assume that the shared state space is spanned by the shared variables. We use the convention that shared variables are in typewriter font, while private variables are slanted.

Remark. The shared state space need not be spanned by shared variables. For example, for algorithms with message passing, the shared state space is the multiset of all messages that are in transit, i.e., that have been sent and have not yet been received (e.g., see [24]). In this paper, however, we stick to shared memory.

2.3 Atomicity

In concurrency, the atomic commands of the threads are interleaved in an arbitrary way. It is therefore important to specify the grain of atomicity of the commands. According to the *principle of single critical reference*, e.g., [52, (3.1)], [6, 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. In the design of algorithms and in the specification of operating system calls, however, one often needs to deviate from the principle. The present paper, however, can follow 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 $pc.p$, which takes values in the set of labels and determines which command is to be executed next. If $pc.p = \ell$, we say that thread p is at label ℓ (notation p **at** ℓ). If L is a set of labels, we write p **in** L for $pc.p \in L$. 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 unlabelled semicolon.

We use a pseudocode with a standard syntax for simple commands, with := for the assignment, **if then** (**else**) **endif** for the conditional command, **while do endwhile** for the conditional repetition, and **for all do end** for the bounded 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 labelled command can be a complete repetition, but then only the steps of the repetition are atomic. For example, the labelled command

```
$\ell$ : while $B$ do $S$ endwhile ; $T$
```

is modelled as

```
$\ell$ : if $B$ then $S$ ; goto $\ell$ else $T$ 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

```
$\ell$ : await $B$ then $S$ .
```

This command can only be executed when the guard B holds, in which case it means execution of S followed by a jump to the next label.

2.4 Nonatomic shared variables

The algorithm of [46] truly solves mutual exclusion, in the sense that it does 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 [42] 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 any 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 [3]. 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

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

We model this by a nondeterministic choice

(1) $\quad \ell : \ (\mathtt{x} := \mathit{arbitrary}\ ;\ \textbf{goto}\ \ell\ [\!]\ \mathtt{x} := E)\ .$

In other words, command (0) is modelled in (1) as a repetition of arbitrary assignments to $\mathtt{x}$ that ends with the actual assignment of E to $\mathtt{x}$. The value of $\mathtt{x}$ during the repetition is indeterminate. We assume the liveness condition that the repetition terminates.

The paper [43] allows arbitrary nonatomic operations as primitive building blocks. This complicates the semantics of nonatomic operations considerably. In the special case of writing into a safe shared variable, however, the definitions agree.

2.5 Mutual Exclusion

Traditionally, mutual exclusion 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. The threads are thus of the form:

```
thread $(p : 0 \le p < N) =$
    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 modelled 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*).

The first-come-first-served property (FCFS) is defined in [38] to mean that when 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*. This is deadlock freedom in the sense of 1.4. Together with FCFS, it implies lockout freedom: every thread in *Waiting* eventually enters *CS*.

3 The New Algorithm

In the mutual exclusion algorithm of [46], mutual exclusion (MX) and FCFS are accomplished by two separate algorithms that are nested as shown in Figure 1. The line (†) holds the inner algorithm for mutual exclusion, which is sketched in Section 3.1 below. The outer algorithm to guarantee FCFS consists of *Doorway*, *WaitingFcfs*, and *ExitFcfs*. Its design is sketched in Section 3.2. The new algorithm is presented in Figure 2 in Section 3.3. In Section 3.4, we specify its grain of atomicity.

The verification of the inner algorithm is done in Section 3.5. In Section 3.6, we specify the FCFS property by means of a history variable that, for an entering thread p, holds the set of threads that thread p must give priority to, and then prove by means of invariants that the algorithm of Figure 2 satisfies this specification.

```
    thread $(p : 0 \le p < N) =$
        while $true$ do
            $NCS$ ; $Doorway$ ; $WaitingFcfs$ ;
(†)         $WaitingMx$ ; $CS$ ; $ExitMx$ ;
            $ExitFcfs$
        endwhile .
```

Fig. 1. Splitting mutual exclusion and FCFS

3.1 The mutual exclusion algorithm

The inner algorithm is the mutual exclusion algorithm of Burns [10] and Lamport [41]. It uses a single safe Boolean shared variable per thread, called $\texttt{cc}[p]$, which is initially false. The idea is that, when it starts competing, thread p makes $\texttt{cc}[p]$ true, and then inspects $\texttt{cc}[q]$ for all threads $q \neq p$. Thread p traverses the list of threads starting from $thr = 0$. When it encounters a thread $thr < p$ with $\texttt{cc}[thr]$, it gives precedence to thr by making $\texttt{cc}[p]$ false, waits until thr has made $\texttt{cc}[thr]$ false, and then sets $\texttt{cc}[p]$ true and restarts its inspection loop, again starting from $thr = 0$. When it has passed itself and finds a thread $thr > p$ with $\texttt{cc}[thr]$, it just waits until thr has made $\texttt{cc}[thr]$ false. When its inspection loop terminates, thread p can enter *CS*. After *CS*, it makes $\texttt{cc}[p]$ false.

This description applies to the lines 28. . . 34 of Figure 2 below.

3.2 Design for the FCFS property

Recall from Section 2.5 that FCFS means that when thread p enters *Doorway* while thread q is in *Waiting*, thread p will not enter *CS* before q does.

The idea to establish the FCFS property is that a thread in *Doorway* announces that it is competing by modifying a shared Boolean array `turn`, but before this announcement it forms a private list of the threads that have announced and have not yet completed *CS*. After *Doorway*, the thread waits until all threads in this private list have completed *CS*. Then it can be served, i.e., participate in the inner mutual exclusion algorithm. After that, it removes its announcement.

This idea guarantees FCFS because the thread does not enter *CS* before all threads in its private list have completed *CS*. Unfortunately, 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 version numbers modulo a constant $R > 1$. This, however, is not enough, because while thread p_1 is in the doorway, thread p_0 may cycle R times through competing periods and then use the version number observed by p_1 again.

In order to preclude this unrestricted progress, we let every thread (p_1 in this case) set the flag `cc` of the inner algorithm before it traverses the doorway, and reset this flag before it starts waiting. This hampers progress because, while p_1 is in the doorway, thread p_0 cannot complete the inner algorithm. It turns out that this idea ensures absence of deadlock, provided we take $R \geq 3$. This is proved in Section 4 (the requirement $R \geq 3$ is needed there to imply the invariant *Mq2A*).

3.3 The new combined algorithm

The algorithm is given in Figure 2. We use labels beginning at 21 to ease proof refactoring as described in Section 7. The fragment of the lines 28 up to 34 is the mutual exclusion algorithm of Burns-Lamport as sketched in Section 3.1, i.e., the line (†) of Figure 1. *Doorway* of Figure 1 consists of the lines 22 up to 25, *WaitingFcfs* consists of the lines 26 and 27, *ExitFcfs* is line 35.

The algorithm uses the index domains

$$\begin{aligned} &\mathit{Thread} = \{q \in \mathbb{N} \mid q < N\} \ , \\ &\mathit{Range} = \{k \in \mathbb{N} \mid k < N \cdot R\} \ , \\ &\mathit{Version} = \{e \in \mathbb{N} \mid e < R\} \ . \end{aligned}$$

Every $k \in \mathit{Range}$ has a unique expression $k = q \cdot R + e$ with $q \in \mathit{Thread}$ and $e \in \mathit{Version}$.

The algorithm uses the following shared variables:

$$\begin{aligned} &\texttt{cc} : \ \textbf{array} \ [\mathit{Thread}] \ \textbf{of} \ \mathit{Boolean} \ ; \\ &\texttt{turn} : \ \textbf{array} \ [\mathit{Range}] \ \textbf{of} \ \mathit{Boolean} \ . \end{aligned}$$

The variables $\texttt{cc}[p]$ and $\texttt{turn}[p \cdot R + e]$ with $e \in \mathit{Version}$ are safe Boolean output variables for thread p. In this way, the algorithm uses $R+1$ shared Boolean variables per thread. Initially, all these shared Boolean variables are false.

A thread p sets its flag $\texttt{cc}[p]$ at 22 when it enters the doorway, and resets it at 25 when leaving the doorway. It sets $\texttt{turn}[p \cdot R + \mathit{nx}.p]$ at 24 in the doorway, and resets it after being served at 35. It thus uses a (persistent) private variable

$$\mathit{nx} : \mathit{Version} \ ,$$

as a version number. This means that $\mathit{nx}.p$ determines the index $p \cdot R + \mathit{nx}.p$ in array `turn`, where thread p indicates its interest in the critical section. After having been served, the thread increments its version number $\mathit{nx}.p$ with 1 modulo R for the next time. We use the operator $\oplus$ to denote addition modulo R.

The threads also have the private variables

$$\begin{aligned} &\mathit{copy} : \ \textbf{array} \ [\mathit{Range}] \ \textbf{of} \ \mathit{Boolean} \ , \\ &\mathit{kk} : \mathit{Range} \ , \\ &\mathit{thr} : \{q \in \mathbb{N} \mid q \leq N\} \ . \end{aligned}$$

```
    thread $(p : 0 \le p < N) =$
21      $NCS$ ;
22      cc[$p$] := (flickering) $true$ ;
23      for all $k \in Range$ do $copy[k]$ := turn[$k$] end ;
24      turn[$p \cdot R + nx$] := (flickering) $true$ ;
25      cc[$p$] := (flickering) $false$ ;
26      while exists $kk \in Range$ with $copy[kk]$ do
27          await $\neg$ turn[$kk$] then $copy[kk] := false$
        endwhile ;
28      cc[$p$] := (flickering) $true$ ; $thr := 0$ ;
29      while $thr < p$ do
            if $\neg$ cc[$thr$] then $thr := thr + 1$
            else
30              cc[$p$] := (flickering) $false$ ;
31              await $\neg$ cc[$thr$] then goto 28
            endif
        endwhile ;
        $thr := p + 1$ ;
32      while $thr < N$ do
            await $\neg$ cc[$thr$] then $thr := thr + 1$
        endwhile ;
33      $CS$ ;
34      cc[$p$] := (flickering) $false$ ;
35      turn[$p \cdot R + nx$] := (flickering) $false$ ; $nx := nx \oplus 1$ ; goto 21 .
```

Fig. 2. The new algorithm for 4 shared bits

The array *copy* serves as a weak private copy of the shared array `turn`, made in line 23. It is a copy in the sense that line 23 looks like copying, but it is weak because we cannot claim any equality between *copy.p* and `turn` because of possible interferences. The variable *kk* serves as an index in this array in lines 26, 27. The variable *thr* is the loop variable of the inner algorithm 29. . . 32.

The elements of array *copy* are initially all false. The private variables *nx*, *thr* and *kk* may have arbitrary initial values of their type. We allow $thr = N$ in order to have a stopping value at line 32.

3.4 The transition system of the algorithm

The transition system of the algorithm is drawn in Figure 3. The initial state has label 21. The main repetition is the outer loop. It has the subloop 26–27, the self-loops at lines 23, 29, 32, and the try-again loop via 30 and 31. The flickering self-loops are indicated by circles around the locations 22, 24, 28, 30, 34, 35.

The individual transitions are most easily described in the syntax of our proof assistant PVS. For PVS, we represent the state space as a datatype. The program variables are fields in this datatype. We use `x` and `y` as variables of this datatype. The value of *pc.p* in state `x` is represented by `` x`pc(p) ``. We define `step(p, x, y)` to mean that $(x, y) \in step_p$; so `p` is the acting thread, and the old state `x` is replaced by the new state `y`. The new state `y` is usually constructed by simultaneous substitution using the `WITH` construction of PVS.

The relation `step(p, x, y)` is constructed as a union (disjunction) of step relations at the different labels. The flickering assignments to `cc` and `turn` are represented in the transition system as indicated in Section 2.4. For instance, for `cc` at line 22, we use two step relations, one for the actual assignment:

```
    step22(p, x, y): bool =
```

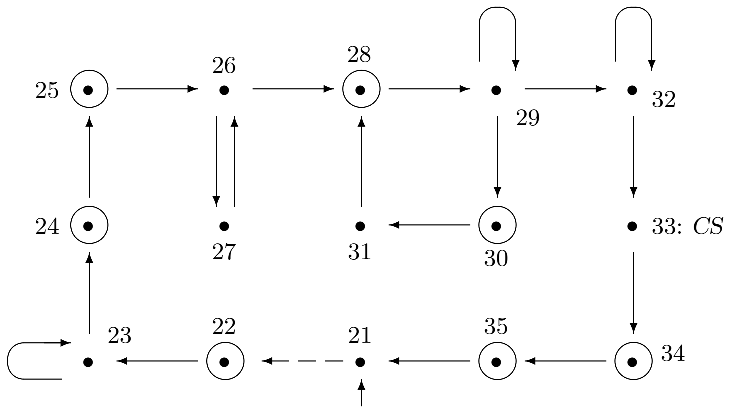

Fig. 3. The transition system

```
    x‘pc(p) = 22 AND
    y = x WITH [ ‘cc(p) := true, ‘pc(p) := 23 ]
```

and a second one for the flickerings of cc at the lines 22, 25, 28, 30, 34:

```
stepCF(p, x, y): bool =
  (x‘pc(p) = 22 OR x‘pc(p) = 25 OR x‘pc(p) = 28
    OR x‘pc(p) = 30 OR x‘pc(p) = 34)
  AND y = x WITH [ ‘cc(p) := NOT x‘cc(p) ]
```

Line 32 is also modelled by two atomic steps, one for the body of the loop:

```
step32B(p, x, y): bool =
  x‘pc(p) = 32 AND x‘thr(p) < N
  AND NOT x‘cc(x‘thr(p))
  AND y = x WITH [ ‘thr(p) := x‘thr(p) + 1 ]
```

and a second one for the termination of the loop:

```
step32E(p, x, y): bool =
  x‘pc(p) = 32 AND x‘thr(p) = N
  AND y = x WITH [ ‘pc(p) := 33 ]
```

The loop of line 23 is modelled in essentially the same way, but we come back to this loop in Section 3.6.

Loops that have inner labels are treated in the natural way. For example, at line 29, entering the loop body is modelled by

```
step29B(p, x, y): bool =
  x‘pc(p) = 29 AND x‘thr(p) < p
  AND y = x WITH [
    ‘thr(p) := IF x‘cc(x‘thr(p)) THEN x‘thr(p)
               ELSE x‘thr(p)+1 ENDIF,
    ‘pc(p) := IF x‘cc(x‘thr(p)) THEN 30 ELSE 29 ENDIF
  ]
```

while termination of the loop is modelled by

```
step29E(p, x, y): bool =
  x‘pc(p) = 29 AND p <= x‘thr(p)
  AND y = x WITH [ ‘thr(p) := p+1, ‘pc(p) := 32 ]
```

The numbered lines of Figure 2 thus correspond to the atomic steps of the transition system. Every thread therefore is always at one label. The reader may verify that every numbered line contains at most one inspection or modification of a shared variable. We write p **at** ℓ for $pc.p = \ell$. In that case, thread p has yet to execute line ℓ. If L is a set of line numbers, we write p **in** L to indicate that p **at** ℓ holds for some $\ell \in L$. We use line numbers starting at 21 to ease proof refactoring as described in Section 7.

3.5 The inner algorithm to guarantee mutual exclusion

Mutual exclusion (MX) at the critical section *CS* is expressed by the invariant:

MX: $\quad q \textbf{ at } 33 \ \wedge\ r \textbf{ at } 33 \ \Rightarrow\ q = r$.

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

We define the state function

$$hot(q) \quad = \quad q \textbf{ in } \{29, 32, 33\} \ ,$$

which expresses that thread q in the region 29—33, and is not at 30 or 31 from which it will go back to line 28. We next claim the invariant

Iq0: $\quad hot(q) \ \Rightarrow\ \texttt{cc}[q]$.

Indeed, it is straightforward to prove that predicate *Iq0* is inductive: it holds initially (because then $hot(q)$ is false), and is preserved in every step. We use "q"s in the names of invariants to ease proof refactoring as described in Section 7.

Now predicate *MX* is implied by the new invariants:

Iq1: $\quad q \textbf{ in } \{33 \ldots\} \ \Rightarrow\ thr.q \geq N$,
Iq2: $\quad hot(q) \ \wedge\ r < thr.q \ \wedge\ hot(r) \ \wedge\ q < thr.r \ \Rightarrow\ q = r$.

Indeed, by *Iq1*, we have that q **at** 33 implies $hot(q) \wedge r < thr.q$. Therefore, *Iq2* implies *MX*.

Predicate *Iq1* is inductive, because the loop of q at line 32 terminates with $thr.q \geq N$. Preservation of *Iq2* is proved as follows. Assume that some thread p does a step that falsifies *Iq2*. Then $q \neq r$, and the step of thread p makes the antecedent of *Iq2* true. Then $p = q$ or $p = r$, because only q and r can change the value of *Iq2*. By symmetry, we may assume $p = q$. In *Iq2*, the step of p ($= q$) can then only influence $hot(q)$ or $thr.q$. To falsify the implication of *Iq2*, its consequent being false ($q \neq r$), the antecedent should become true. When $hot(q)$ becomes true in step 28, $thr.q$ becomes 0, contradicting $r < thr.q$. Therefore, $hot(q)$ holds in the precondition, and the step of q is the incrementation of $thr.q$ in line 29 or 32. In the precondition of the step, we therefore have $r = thr.q$, and $hot(r)$ because the other conjuncts of the antecedent of *Iq2* must evaluate to true. By *Iq0* for thread r, this implies $\texttt{cc}[thr.q]$, contradicting the guard of the incrementation in line 29 or 32. This proves:

Theorem 1. *The algorithm satisfies mutual exclusion.*

To formally prove that *MX* is an invariant, we prove that the conjunction of the universal quantifications of *Iq0*, *Iq1*, *Iq2* (and some other predicates) is inductive and therefore an invariant, and that it implies *MX*.

The inner algorithm also has the obvious invariant:

Iq3: $\quad q \textbf{ at } 32 \ \Rightarrow\ q < thr.q$,

which is needed in the proof of absence of immediate deadlock in Section 4.

3.6 The outer algorithm guarantees FCFS

As announced, the FCFS property means that, when thread p enters *Doorway* while thread q is in *Waiting*, thread p will not be served before q. We specify this formally by introducing a variable `central` to hold the set of the central threads, i.e., those that are in *Waiting* or *CS*, and by giving each thread p a set of predecessors, `predec`$[p]$, equal to `central` when it enters the *Doorway*. We thus introduce two shared history variables:

$$\texttt{central} : \textbf{set}[\mathit{Thread}] \ ,$$
$$\texttt{predec} : \textbf{array}[\mathit{Thread}] \ \textbf{of} \ \textbf{set}[\mathit{Thread}] \ ,$$

with initially `central` $= \emptyset$ and `predec`$[p] = \emptyset$ for all threads p.

The FCFS property is specified by Figure 4. Just before a thread p enters *Doorway*, the set `predec`$[p]$ becomes `central`. Before starting *Waiting*, thread p enters `central`. After *CS*, thread p leaves `central` and `predec`$[q]$ for all threads q. FCFS is expressed by the requirement that thread p is not served before `predec`$[p]$ is empty. The modifications of `central` and `predec` are supposed to be atomic. This is possible because they are history variables: they only serve to specify and prove, not to compute.

```
      thread $(p : 0 \le p < N)$ =
$\ell_0$ :      $NCS$ ; predec$[p]$ := central ; $Doorway$ ;
$\ell_1$ :      central$[p]$ := $true$ ; $Waiting$ ;
$\ell_2$ :      await predec$[p] = \emptyset$ ;
           $being\ served$ ;
$\ell_3$ :      central$[p]$ := $false$ ; for all $q$ do predec$[q][p]$ := $false$ end ;
           $Exit$ ; goto $\ell_0$ .
```

Fig. 4. The specification of FCFS

In order to show that the algorithm of Figure 2 satisfies this specification, we extend it with actions on the history variables `central` and `predec` in Figure 5, and prove that `predec`$[p]$ is empty when p is being served. We let the labels ℓ_0, ℓ_1, ℓ_3 of Figure 4 correspond to the line numbers 21, 24, and 34 of the Figures 2 and 5. As the *Doorway* must not contain wait statements, it is important to verify that all waiting in Figure 5 occurs after label ℓ_1, i.e., line 24. The modifications of `central` and `predec` can be combined atomically with the same line of Figure 2. This is allowed because they are history variables [1] (called auxiliary variables in [52]): they do not influence the computation and only serve in the proof of correctness. It is clear from Figure 5 that `central` satisfies the invariant:

Kq0: $\quad r \in \texttt{central} \equiv r \ \textbf{in} \ \{25 \ldots 34\}$.

We need to place label ℓ_2 of Figure 4 somewhere before *CS*. For convenience, we place it at line 28, in the sense that we prove the invariant that `predec`$[q]$ is empty unless thread q is in the region from 22 to 27:

FCFS: $\quad \texttt{predec}[q] = \emptyset \ \vee \ q \ \textbf{in} \ \{22 \ldots 27\}$.

Indeed, it follows from *Kq0* that, while thread p is in $\{22 \ldots\}$, the variable `predec`$[p]$ holds all its *predecessors*, the threads that have announced before p executed line 21 and have not yet reached line 35. The invariant *FCFS* thus expresses that all its predecessors have reached line 35 when p is in $\{28 \ldots\}$, and in particular when p is at *CS*. This implies the first-come-first-served property as defined in Section 2.5.

```
     thread ($p : 0 \le p < N$) =
21       predec[$p$] := central ;
         for all $k$ do $turnset[k] := true$ end ;
22       cc[$p$] := (flickering) $true$ ;
23       while exists $k$ with $turnset[k]$ do
             $copy[k]$ := turn[$k$] ; $turnset[k] := false$
         endwhile ;
         $cnt$ := shacnt ; shacnt := shacnt + 1 ;
24       turn[$p \cdot R + nx$] := (flickering) $true$ ; central[$p$] := $true$ ;
25       cc[$p$] := (flickering) $false$ ;
26       while exists $kk$ with $copy[kk]$ do
27           await $\neg$ turn[$kk$] then $copy[kk] := false$
         endwhile ;
28       cc[$p$] := (flickering) $true$ ; $thr := 0$ ;
29       while $thr < p$ do
             if $\neg$ cc[$thr$] then $thr := thr + 1$
             else
30               cc[$p$] := (flickering) $false$ ;
31               await $\neg$ cc[$thr$] then goto 28
             endif
         endwhile ;
         $thr := p + 1$ ;
32       while $thr < N$ do
             await $\neg$ cc[$thr$] then $thr := thr + 1$
         endwhile ;
33       $CS$ ;
34       cc[$p$] := (flickering) $false$ ; central[$p$] := $false$ ;
         for all $q$ do predec[$q$][$p$] := $false$ end ;
35       turn[$p \cdot R + nx$] := (flickering) $false$ ;
         $nx := nx \oplus 1$ ; $inc := inc + 1$ ; goto 21 .
```

Fig. 5. History extension of the 4 bit algorithm

For convenience, we give the complete extension of the algorithm with history variables in Figure 5. The reader may verify that Figure 5 only differs from Figure 2 in the lines 21, 23, 24, 34, and 35. Moreover, `central` and `predec` only occur in the lines 21, 24, and 34, and do not influence the computation. For simplicity, in Figure 5, we omit *NCS* from line 21 as irrelevant.

The order of inspection of the elements of array `turn` in line 23 is irrelevant. In order to verify this, we give each thread a private array *turnset* of the same type as *copy* to indicate the indices k for which it still has to inspect `turn`[k]. Variable *turnset* only occurs in the lines 21 and 23 of Figure 5. It is initially empty. It is set in line 21 for the ease of invariant *Kq1* below.

Anticipating the proof of immediate deadlock, we introduce a shared integer history variable `shacnt` and private integer history variables *cnt* to determine the order in which the threads arrive at line 24. These variables only occur at the end of line 23 of Figure 5. We come back to this in Section 4. The private history variable *inc* incremented in line 35 will be used in the proof of progress (Section 5).

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

$$(2)\qquad \begin{aligned} (q, e) \in \mathtt{turn} &\equiv \mathtt{turn}[q \cdot R + e] \ , \\ (q, e) \in \mathit{turnset.p} &\equiv \mathit{turnset.p}[q \cdot R + e] \ , \end{aligned}$$

$$(q, e) \in \mathit{copy}.p \quad \equiv \quad \mathit{copy}.p[q \cdot R + e] \ .$$

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

We postulate the invariants:

Kq1: $r \in \texttt{predec}[q] \ \Rightarrow \ (r, \mathit{nx}.r) \in \mathit{turnset}.q \cup \mathit{copy}.q$,
Kq2: $\mathit{turnset}.q = \emptyset \ \vee \ q \ \textbf{in} \ \{22, 23\}$,
Kq3: $\mathit{copy}.q = \emptyset \ \vee \ q \ \textbf{in} \ \{23 \ldots 27\}$.

It is clear that these three predicates together imply *FCFS* as defined above. Note that we would have to formulate *Kq1* in a more complicated way if we had initialized *turnset* in line 22 instead of line 21.

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 *Kq1* holds initially because $\texttt{predec}[q]$ is empty. It is threatened only by the modifications of $\texttt{predec}[q]$, $\mathit{turnset}.q$, $\mathit{copy}.q$, and $\mathit{nx}.r$ in 21, 23, 27, 35. It is preserved by thread q in line 21 because $(r, \mathit{nx}.r) \in \mathit{turnset}.q$ becomes true. It is preserved by q under the modifications of *turnset* and *copy* in lines 23 and 27 because of the additional invariants *Kq4* and *Kq5* below. At line 27, the guard $\neg\,\texttt{turn}[r]$ is used to infer that thread r is no longer in `central`. Note that the history variable $\texttt{predec}[q]$ is modified when thread r executes 34. Predicate *Kq1* is preserved under the modification of $\mathit{nx}.r$ in line 35 because *Kq0* and *Kq4* imply that thread r is not in $\texttt{predec}[q][r]$ when it is at line 35.

Kq4: $\texttt{predec}[q] \subseteq \texttt{central}$,
Kq5: $r \in \texttt{central} \ \Rightarrow \ (r, \mathit{nx}.r) \in \texttt{turn}$.

The predicates *Kq4* and *Kq5* are easily seen to be invariant (use *Kq0* to prove preservation of *Kq5* at lines 24 and 35).

4 Absence of immediate deadlock

As a first step in the proof of deadlock freedom, we prove absence of immediate deadlock as defined below. This is weaker, because immediate deadlock is a special form of deadlock.

A step of thread p is a pair of states in the step relation step_p. We partition the steps of p in two types: forward steps and environment steps. We define a step of p to be an *environment* step if it is the step from line 21 to 22, or a flickering step in which $\mathit{pc}.p$ remains unchanged. All other steps of p are *forward* steps of p. The reader may verify that the forward steps are the solid arrows of Figure 3, and also that they are the steps that begin with p not at line 21 and that modify $\mathit{pc}.p$ or the loop variables $\mathit{thr}.p$ or $\mathit{turnset}.p$. The step of 21 is called an environment step because it is the request for access to the critical section. The flickering steps 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.

One can argue that the critical section *CS* is also executed by the environment, but for the proof of progress we need the assumption that *CS* terminates. We therefore treat *CS* as 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, while absence of deadlock as defined in section 1.4 is a liveness property.

We define a thread to be *forward disabled* iff it cannot do a forward step. It is easy to verify that thread p is forward disabled if and only if it satisfies the condition:

$$\begin{array}{l} \mathit{Dis}(p) \quad \equiv \\ p\ \mathbf{at}\ 21\ \vee\ (p\ \mathbf{at}\ 27 \wedge \mathtt{turn}[\mathit{kk}.p])\ \vee\ (p\ \mathbf{in}\ \{31, 32\} \wedge \mathtt{cc}[\mathit{thr}.p])\ . \end{array}$$

When it is at a flickering location, 22, 24, etc., thread p is not disabled because it can do the actual assignment, which is a forward step.

We turn to the proof of absence of immediate deadlock. For the first step of the proof, we claim the easy invariant:

Lq0: $\quad \mathtt{cc}[q]\ \Rightarrow\ q\ \mathbf{in}\ \{22 \ldots 25\} \cup \{28 \ldots 30\} \cup \{32 \ldots 34\}\ .$

Indeed, it is inductive: it holds initially and is preserved in every step. Note that *Lq0* is an implication, not an equivalence: due to flickering, $\mathtt{cc}[q]$ 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 forward disabled. Then all threads are at lines 21 or 27.*

Proof. By condition *Dis*, all threads are idle (at line 21) or forward disabled at one of the lines 27, 31, or 32. Therefore, if there are threads q for which $\mathtt{cc}[q]$ holds, all of these are at 32 because of the invariant *Lq0*. Let q_1 be the highest number q with $\mathtt{cc}[q]$. Then q_1 is a thread, it is at line 32, and is forward disabled. It therefore satisfies $\mathit{thr}.q_1 < N$ and $\mathtt{cc}[\mathit{thr}.q_1]$. By invariant *Iq3* of Section 3.5, we have $q_1 < \mathit{thr}.q_1 < N$. As $\mathtt{cc}[\mathit{thr}.q_1]$ 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 forward disabled at 31 or 32. This proves that all threads are at 21 or 27. □

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

To argue about this order, we record it in private integer history variables $\mathit{cnt}.p$ for all threads p. For this purpose, we also introduce a shared history variable $\mathtt{shacnt}$, and let cnt and $\mathtt{shacnt}$ be updated in the final step of loop 23 as given in Figure 5. We initialize $\mathtt{shacnt} := 1$ and $\mathit{cnt}.p := 0$ 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 three invariants:

Lq1: $\quad q\ \mathbf{at}\ 27\ \Rightarrow\ \mathit{copy}.q[\mathit{kk}.q]\ ,$
Lq2: $\quad (q, e) \in \mathtt{turn}\ \Rightarrow\ q\ \mathbf{in}\ \{24 \ldots\}\ \wedge\ \mathit{nx}.q = e\ ,$
Lq3: $\quad q\ \mathbf{in}\ \{24 \ldots\}\ \wedge\ (r, \mathit{nx}.r) \in \mathit{copy}.q\ \Rightarrow\ \mathit{cnt}.r < \mathit{cnt}.q\ .$

Indeed, the predicates *Lq1*, and *Lq2* are easily seen to be inductive. For *Lq2*, note that, due to flickering of $\mathtt{turn}$, $(q, e) \in \mathtt{turn}$ can be true when q is at 24, and false when q is at 35.

Since the private variable *cnt* is set to the shared variable $\mathtt{shacnt}$ in the step towards 24, the consequent of *Lq3* 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 *Lq3* 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. Then, by Lemma 1, all threads are at 21 or 27. Therefore every competing thread q is forward disabled at 27, and has $\mathtt{turn}[kk.q]$ because of $Dis(q)$, and $copy.q[kk.q]$ because of *Lq1*. If there is a thread at line 27, let q_0 be the thread at 27 with the smallest value for $cnt.q$. Write $kk.q_0 = r \cdot R + e$ with $r \in Thread$ and $e \in Version$. Then we have $(r, e) \in \mathtt{turn}$ and $(r, e) \in copy.q_0$. By *Lq2*, we have r **in** $\{24 \ldots\}$ and $e = nx.r$. By *Lq3*, we have $cnt.r < cnt.q_0$. This contradicts minimality of $cnt.q_0$, because thread r is competing and hence at 27. This proves that there are no competing threads at all. □

It remains to prove invariance of *Lq3*. Here the intricacies of the algorithm appear. As suggested by a referee, we present the invariants in a bottom-up fashion.

We first claim the easy inductive invariants:

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

It is in *Mq1* that the double role of array `cc` in the outer and the inner algorithm is exploited.

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: $\quad q \ \mathbf{in}\ \{23, 24\} \wedge (r, e) \in copy.q \wedge nx.r \neq e$
$\quad\quad \Rightarrow nx.r = e \oplus 1 \wedge r \ \mathbf{in}\ \{\ldots 32\} \wedge (r \ \mathbf{in}\ \{\ldots 28\} \vee thr.r \leq q)$.

The second and third conjunct of the consequent of *Mq2* are included because they help 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, 29 and 32. 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 $copy.q[k] := \mathtt{turn}[k]$ because of *Lq2*. It is preserved at lines 29 and 32 because when thread r falsifies $thr.r \leq q$, it does so under the precondition $thr.r = q$ and hence $\neg\, \mathtt{cc}[q]$ because of the code; therefore q is not in $\{23, 24\}$ because of *Mq1*.

The assumption $R \geq 3$ implies that $v \oplus 1 \neq e$ for each $v \in \{e, e \oplus 1\}$. Taking $v = nx.r$, we see that *Mq2* implies:

Mq2A: $\quad q \ \mathbf{in}\ \{23, 24\} \Rightarrow (r, nx.r \oplus 1) \notin copy.q$.

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

Mq3: $\quad (r, nx.r \oplus 1) \in copy.q \Rightarrow r \ \mathbf{at}\ 21 \vee q \in \mathtt{predec}[r]$.

Predicate *Mq3* holds initially because $copy.q$ is empty. It is threatened at the lines 21, 23, and 34. It is preserved by the update $copy.q := \mathtt{turn}[k]$ in line 23 because of *Lq2* and $R > 1$ (to guarantee $nx.r \oplus 1 \neq nx.r$). It is preserved by the modification of `predec` in line 34 because *Kq3* implies that q is not at 34.

Preservation of *Mq3* when thread r leaves line 21 is more complicated. At this point, $\mathtt{predec}[r]$ gets a new value. The antecedent of *Mq3* implies by *Kq3* and *Mq2A* that thread q is in $\{25 \ldots 27\}$. Therefore, by *Kq0*, we have $q \in \mathtt{predec}[r]$.

The invariants *Mq3* and *FCFS* together imply the derived invariant:

Mq3A: $\quad (r, nx.r \oplus 1) \in copy.q \Rightarrow r \ \mathbf{in}\ \{\ldots 27\}$.

We next claim the invariant:

Mq4: $\quad (r, nx.r) \in copy.q \Rightarrow r \ \mathbf{in}\ \{24 \ldots\}$.

Predicate *Mq4* holds initially because then *copy.q* is empty. It is threatened at lines 23 and 35. It is preserved by the update $copy.q[k] := \texttt{turn}[k]$ in line 23 because of *Lq2*. It is preserved by the modification of *nx.r* in line 35 because of *Mq3A*.

We can finally prove predicate *Lq3*. It holds initially because then q is at line 21. It is threatened at lines 23 and 35. 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 35 because of *Mq3A*. This concludes the proof of the invariance of *Lq3* and hence of Theorem 2.

Note that the nonatomicity of the elements of the arrays `turn` and `cc` comes up in the invariants *Iq0*, *Kq5*, *Lq0*, *Lq2*, and *Mq1*. Atomicity of these elements would allow the stronger invariants:

Kq5a: $\quad (q, e) \in \texttt{turn} \quad \equiv \quad q \textbf{ in } \{25 \ldots 35\} \ \wedge \ nx.q = e \ ,$
Lq0a: $\quad \texttt{cc}[q] \quad \equiv \quad q \textbf{ in } \{23 \ldots 25\} \cup \{29, 30\} \cup \{32 \ldots 34\} \ .$

For us, these predicates are no invariants because of the flickering in the lines 22, 24, 25, 28, 30, 34, and 35. For instance, *Kq5a* is not preserved when $\texttt{turn}[p \cdot R + nx]$ flickers at line 35. Then $(p, nx.p)$ leaves the set `turn`, while p remains at 35 and does not modify *nx.p*.

5 Liveness: lockout freedom

In the algorithm of Figure 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 $\texttt{cc}[q]$. Therefore, even though we have proved absence of immediate deadlock, we still need a careful proof of liveness.

This is proved in this section: under the assumption of weak fairness, we prove that every thread is eventually always idle, i.e., at line 21. This implies lockout freedom: every competing thread eventually passes *CS* to come back to line 21.

Roughly speaking, the argument is as follows. If there is a thread that remains competing, FCFS implies that from some point onward no thread can enter *CS* anymore. Weak fairness then implies that every thread eventually either gets stuck at one of the lines 21, 27, 31 or 32, or remains forever cycling over 28–31. We eliminate the latter possibility by considering the cycling thread with the lowest number. We eliminate the locations 27, 31, 32 by invoking Theorem 2, the absence of immediate deadlock.

5.1 A variant function for progress

To formally prove the limit behaviour suggested, we construct an integer valued state function $avf(p)$ that expresses progress for thread p, and that grows proportionally to the number of times the thread executes the critical section.

This is done in the following way. As shown in Figure 5, each thread p has a private history variable *inc.p* which is incremented with 1 whenever thread i executes line 35 and goes back to *NCS* (*inc* abbreviates “incarnation”). We now introduce a numerical constant A and the variant function

$$\begin{aligned} avf(p) = {} & A \cdot inc.p + (p \textbf{ in } \{28 \ldots 31\} \ ?\ 7 : pc.p - 21) \\ & + (p \textbf{ in } \{22 \ldots\} \ ?\ 3 \cdot N \cdot R - 3 \cdot \#turnset.p - 2 \cdot \#copy.p : 0) \\ & + (p \textbf{ in } \{32 \ldots\} \ ?\ thr.p : 0) \ . \end{aligned}$$

Here, $(b \ ?\ x : y)$ is the conditional expression **if** b **then** x **else** y **endif**, as in the programming language C. The notation $\#S$ denotes the number of elements of S regarded as a set. It is clear that $avf(p)$ changes only when thread p itself does a step. We choose $A \geq 3 \cdot N \cdot R + N + 15$. We infer from this that the jump from line 35 to line 21 increases $\mathit{avf}(p)$ and that $\mathit{avf}(p)$ never decreases. Indeed, most forward steps of thread p increase $\mathit{avf}(p)$. For example, the step from 23 to 23 removes one element of *turnset.p* and possibly adds one elements to *copy.p*. The only exceptions are the steps within the loop 28–31, which keep $\mathit{avf}(p)$ constant. We have chosen to keep $\mathit{avf}(p)$ constant in this loop in order to keep it at least nondecreasing.

The growth of $\mathit{avf}(p)$ is proportional to the growth of *inc.p* because of

$$A \cdot \mathit{inc.p} \leq \mathit{avf}(p) < A \cdot (\mathit{inc.p} + 1) \ .$$

5.2 Weak fairness

We prove that under the assumption of weak fairness every thread that enters at line 22 eventually comes back to the noncritical section. We need weak fairness for this, because otherwise the thread can remain in the doorway, or another thread might block it by infinitely many flickerings.

The assumption of *weak fairness* means that we restrict the attention to weakly fair executions. Roughly speaking, we assume that every occurring execution of the system is weakly fair. Recall that an execution is an infinite sequence of states that starts in an initial state and has a step between every pair of subsequent states. An execution is called *weakly fair* iff, whenever from some state onward some thread q can always do a forward step, thread q will eventually do the step. Recall that a *forward* step is a nonflickering step from a location $\neq 21$ (see Section 4).

We formalize these concepts in a set-theoretic version of temporal logic. Let X be the state space. We identify the set X^ω of the infinite sequences of states with the set of functions $\mathbb{N} \to X$. For a sequence $\mathit{xs} \in X^\omega$ and $n \in \mathbb{N}$, we occasionally refer to $\mathit{xs}(n)$ as the state at time n. For a subset $U \subseteq X$, we define $[\![U]\!] \subseteq X^\omega$ as the set of sequences xs with $\mathit{xs}(0) \in U$. For a relation $A \subseteq X^2$, we define $[\![A]\!]_2 \subseteq X^\omega$ as the set of sequences xs with $(\mathit{xs}(0), \mathit{xs}(1)) \in A$.

For $\mathit{xs} \in X^\omega$ and $m \in \mathbb{N}$, we define the shifted sequence $\mathit{drop}(m, \mathit{xs})$ by $\mathit{drop}(m, \mathit{xs})(n) = \mathit{xs}(m + n)$. For a subset $P \subseteq X^\omega$ we define $\Box P$ and $\Diamond P$ as the subsets of X^ω given by

$$\begin{aligned} \mathit{xs} \in \Box P &\equiv (\forall\, m \in \mathbb{N} : \mathit{drop}(m, \mathit{xs}) \in P) \ , \\ \mathit{xs} \in \Diamond P &\equiv (\exists\, m \in \mathbb{N} : \mathit{drop}(m, \mathit{xs}) \in P) \ . \end{aligned}$$

We read $\Box P$ as *always* P, and $\Diamond P$ as *eventually* P.

We now apply this to the algorithm. We write $\mathit{init} \subseteq X$ for the set of initial states and $\mathit{step} \subseteq X^2$ for the step relation on X. Following [1], we use the convention that relation *step* is reflexive (contains the identity relation). An *execution* is an infinite sequence of states that starts in an initial state and in which each subsequent pair of states is connected by a step. The set of executions of the algorithm is therefore

$$\mathit{Ex} = [\![\mathit{init}]\!] \cap \Box [\![\mathit{step}]\!]_2 \ .$$

By induction, every state in an execution satisfies all invariants proved for the algorithm. For any thread q, the numbers *inc.q* and $\mathit{avf}(q)$ form nondecreasing sequences along any execution.

We define (q **at** 21) to be the subset of X of the states in which thread q is at line 21. An execution in which thread q is always eventually at *NCS*, is therefore an element of $\Box\Diamond[\![q \textbf{ at } 21]\!]$.

Weak fairness is formalized as follows. Let $\mathit{fw}(q) \subseteq X^2$ be the set of forward steps of thread q. Using forward-disabledness as defined in Section 4, an execution is defined to be *weakly fair* [44] for thread q iff thread q is always eventually forward-disabled or it always eventually does a forward step. In other words, iff xs belongs to the set

$$WF(q) = \Box\Diamond[\![\, Dis(q)\,]\!] \ \cup\ \Box\Diamond[\![\, fw(q)\,]\!]_2 \ .$$

The set of weakly fair executions is defined as the intersection (conjunction):

$$WFEx = Ex \cap \textstyle\bigcap_q WF(q) \ .$$

Remark. The strength of weak fairness depends in an essential way on the grain of atomicity. For example, if thread p performs a sequence of flickering steps at line 22, in our transition system it remains at line 22, and is therefore continuously enabled to do the forward step in which it goes to line 23 and makes $\mathtt{cc}[p]$ true. Weak fairness therefore implies that the flickering at line 22 terminates.

If we would split the flickering step into a part that does the arbitrary assignment to $\mathtt{cc}[p]$ and goes to some other label, and a second part in which p jumps back to 22, the forward step of p would not be continuously enabled and, under weak fairness, flickering would not necessarily terminate.

5.3 Liveness

In this section, we claim and prove the main liveness result:

Theorem 3. *In every weakly fair execution, every thread q is always eventually at NCS.*

This is expressed in the temporal formula:

(3) $$WFEx \subseteq \Box\Diamond[\![\, q \ \textbf{at}\ 21\,]\!] \ .$$

This is lockout freedom under assumption of weak fairness: every competing thread eventually returns to *NCS*; therefore it must have been served.

We have proved the theorem formally with the proof assistant PVS, but we present the proof here in a less formal way. In particular, we use the words *always* and *eventually* for the operators $\Box$ and $\Diamond$.

The proof of the theorem is based on *inc* and *avf* as introduced in Section 5.1. Let us define an execution to be *always busy* for q if it is not always eventually idle (at 21). We shall prove that every weakly fair execution is not always busy.

Thread q increments $inc.q$ only when it goes back to 21. Therefore, if the execution is always busy for q, $inc.q$ is bounded.

Lemma 2. *If an execution is weakly fair for q and has $inc.q$ bounded, then $pc.q$ is eventually constant with one of the values 21, 27, 31, or 32, or $pc.q$ remains cycling through 28–31.*

Proof. Because of $avf(q) < A \cdot (inc.q + 1)$, we have that $avf(q)$ is also bounded. Because $avf(q)$ never decreases, it follows that $avf(q)$ is eventually constant.

Assume that $pc.q$ is eventually constant, say $pc.q = \ell$ for all states from time n onward. If $\ell \in \{21, 27, 31, 32\}$, we are done. Otherwise, from time n onward, $Dis(q)$ is false and thread q is continuously forward enabled. By weak fairness, it will therefore eventually take infinitely many forward steps. As these do not increase $avf(q)$ or modify $pc.q$, we have that $\ell = 29$ and that every forward step of q increases $thr.q$. Therefore, the guard of line 29 will force q to jump to line 32, a contradiction.

If $pc.q$ is not eventually constant, the fact that $avf(q)$ is eventually constant implies that thread q is eventually cycling in the loop 28–31. $\Box$

Lemma 3. *Consider an execution that is weakly fair for all threads and that is always busy for some thread r. Then, for every thread q (including r), $pc.q$ is eventually constant with one of the values 21, 27, 31, or 32.*

Proof. By Lemma 2, from some time n_0 onward, thread r is always in $\{27 \ldots 32\}$, and therefore in `central` by *Kq0*. It follows that *inc.r* is bounded. Any thread $q \neq r$ that enters after time n_0, gets $r \in \texttt{predec}[q]$. By *FCFS*, such a thread q cannot go beyond line 27, and therefore cannot increment *inc.q* again. This implies that *inc.q* is bounded for every thread q, including r. By Lemma 2, every thread q therefore has *pc.q* eventually constant with one of the values 21, 27, 31, or 32, or it remains cycling through 28–31.

To prove that the latter alternative does not occur, we consider the set S of the threads q that remain cycling through 28–31. All threads $q \notin S$ have *pc.q* eventually constant with one of the values 21, 27, 31, or 32. As the set of threads is finite, there is a time $n_1 \geq n_0$ such that, from time n_1 onward, for every thread $q \notin S$, we have *pc.q* constant with one of the values 21, 27, 31, 32. By *Iq0* and *Lq0*, it follows that, from time n_0 onward, $\texttt{cc}[q]$ is constant for every $q \notin S$.

It suffices to prove that the set S is empty. Assume that S is nonempty. Let $p \in S$ be the smallest number in S. Thread p remains cycling through 28–31. Therefore, at line 29, at some time after n_1, thread p observes $\texttt{cc}[thr.p]$ so that it can go to 30. At line 31 it then eventually observes $\neg\texttt{cc}[thr.p]$ to go back to line 28. This concerns the same value of *thr.p*. Therefore, the value of $\texttt{cc}[thr.p]$ changes after time n_1. This implies that $thr.p \in S$. The guard of line 29 is $thr.p < p$. This contradicts minimality of p in S. Therefore S is empty. □

We turn to the proof of Theorem 3. Consider a weakly fair execution. Assume that it is always busy for some thread. By Lemma 3, for every thread q, *pc.q* is eventually constant with one of the values 21, 27, 31, or 32. As above, there is a time n_0 such that from time n_0 onward, for every thread q, we have *pc.q* constant with one of the values 21, 27, 31, 32. Moreover, we may assume that, from time n_0 onward, $avf(q)$ is constant for every thread q. Theorem 2 implies that some thread, say p, is forward enabled at time n_0.

The only steps that can be taken at time n_0 by any thread are forward steps at 27, 31, 32, and the environment step from 21 to 22. The assumptions imply that none of these steps is taken. Therefore, thread p remains forward enabled indefinitely. By weak fairness it will eventually take a step, thus contradicting the assumptions. This proves Theorem 3. □

6 Variations of the Algorithm

The Boolean array `cc` serves two roles in the algorithm. In the lines 22 and 25, it guards *Doorway*. In the lines 28. . . 32, it guarantees mutual exclusion. This has the effect that a thread (say p) at line 29 can be forced to repeatedly restart its loop at 28 because another thread ($q < p$) is in *Doorway*. This restarting can happen at most p times. When it happens, all threads r in the lines 22–27 with $p \in \texttt{predec}[r]$ also have to wait.

If there are many threads (large N), this may be bad for performance. Moreover, in principle, it is superfluous: we can split the two roles of `cc`.

6.1 A separate guardian for the doorway

For large N, therefore, it may be better for performance to split the roles of `cc` by introducing a second Boolean array, say `dw` for doorway, and replacing `cc` by `dw` in the lines 22 and 25. Array `dw` has length N, and its elements are initially all false. In order to preclude the unrestricted progress mentioned in Section 3.2, we then need to introduce another waiting loop:

```
(*)        for all $q$ do await $\neg\,\texttt{dw}[q]$ end ,
```

which can be placed before line 22, or 26, or 28, or 33.

This can be optimized even further by letting every thread make a private copy *dwcopy* of `dw`. In this way, we arrive at a variation of Figure 2 where the lines 21. . . 25 are replaced by Figure 6. Note that the loop (*) is replaced by lines 24 and 25. Obviously, in line 35, command **goto** 21 must be replaced by **goto** 18.

```
18        $NCS$ ;
19        for all threads $q$ do $dwcopy[q] := \mathtt{dw}[q]$ end ;
20        $\mathtt{dw}[p] :=$ (flickering) $true$ ;
21        for all $k \in Range$ do $copy[k] := \mathtt{turn}[k]$ end ;
22        $\mathtt{turn}[p \cdot R + nx] :=$ (flickering) $true$ ;
23        $\mathtt{dw}[p] :=$ (flickering) $false$ ;
24        while exists $q$ with $dwcopy[q]$ do
25            await $\neg\, \mathtt{dw}[q]$ then $dwcopy[q] := false$
          endwhile .
```

Fig. 6. The variation for 5 shared bits. These lines replace lines 21. . . 25 of Fig. 2.

With respect to average performance, one may hope that, when thread p at line 19 observes that q is in *Doorway*, thread q usually has left *Doorway* when thread p is at line 25.

The correctness proof of the main algorithm can be easily adapted to the variation. The line numbers have to be shifted carefully. The main difference is in the invariants *Mq1* and *Mq2*, which are replaced by

Mq1B: q **in** $\{21, 22\} \Rightarrow \mathtt{dw}[q]$,
Mq2B: q **in** $\{21, 22\} \ \wedge\ (r, e) \in copy.q \ \wedge\ nx.r \neq e$
$\Rightarrow nx.r = e \oplus 1 \ \wedge\ (r$ **at** $18 \ \vee\ threadset.r[q] \ \vee\ dwcopy.r[q])$,

where $threadset.r[q]$ means that thread r is at line 19 and has yet to treat thread q in its loop. As with *Mq2*, the second conjunct of the consequent of *Mq2B* is needed to prove preservation of the first conjunct. In the present version the unrestricted progress mentioned in Section 3.2 is precluded by waiting at line 25 by means of `dw`, while our main algorithm does so at the lines 29 and 32 by means of `cc`.

In *Lq3*, *Mq2A*, *Mq3*, and *Mq4*, only line numbers have to be changed. We also need some new but obvious invariants about *threadset*, `dw`, and *dwcopy*.

In the proof of liveness, the variant function *avf* needs to be adapted. It must increase whenever thread p executes the loop body of line 19 (removes a thread from *threadset.p*) or executes line 25 (puts $dwcopy.p[q] := false$ and goes to 24):

$$\begin{aligned} avf(p) = {} & A \cdot inc.p + (p \ \mathbf{in}\ \{28 \ldots 31\}\ ?\ 10 : pc.p - 18) \\ & + (p \ \mathbf{in}\ \{19 \ldots\}\ ?\ 3 \cdot N \cdot R - 3 \cdot \#turnset.p - 2 \cdot \#copy.p \\ & \qquad + 3 \cdot N - 3 \cdot \#threadset.p - 2 \cdot \#dwcopy.p :\ 0) \\ & + (p \ \mathbf{in}\ \{32 \ldots\}\ ?\ thr.p : 0)\ . \end{aligned}$$

Because of the two extra loops (the second one has two lines), the condition on A has to be replaced by $A \geq 3 \cdot N \cdot R + 4 \cdot N + 18$. In the proof of Theorem 3, we need to reckon with the additional waiting at line 25. We have completely verified this alternative by adapting the PVS proof of the shorter version, see [28].

As an alternative to Figure 6, the loop with `dw` tests at lines 24, 25 can be postponed to the location just before the inner algorithm (line 28) or the critical section (line 33). This would make the probability of waiting smaller, but in the case of waiting other threads may be hindered more.

6.2 The version of Lycklama-Hadzilacos

On the other hand, the algorithm *LH* of [46] can be obtained from Figure 2, by introducing the array `dw` (not using Figure 6), and replacing the assignements to `cc` in the lines 22 and 25 by assignments to `dw`, by taking $R = 2$, adding a third Boolean array $\texttt{v}[N]$ (also initially false), and modifying the lines 24, 26, 27, and 35 in the following way.

```
24a     $nx := nx \oplus 1$ ;
24b     $\texttt{turn}[2p + nx] :=$ (flickering) $\neg\,\texttt{turn}[2p + nx]$ ;
24c     $\texttt{v}[p] :=$ (flickering) $true$ ;
26      for all $q < N$ do
27          await $\neg\,\texttt{dw}[q]$
              $\wedge\ \neg\,(\texttt{v}[q] \wedge copy[2q] = \texttt{turn}[2q] \wedge copy[2q+1] = \texttt{turn}[2q+1])$ ;
35      $\texttt{v}[p] :=$ (flickering) $false$ ; goto 21 .
```

Algorithm *LH* uses $\texttt{v}[p]$ to indicate that thread p is competing and two `turn` bits of p to encode the version number in Gray code, whereas our algorithm uses three `turn` bits to indicate that thread p is competing together with its version number.

6.3 Efficiency

The await condition at line 27 of *LH* is much more complicated than the await conditions in our algorithm at line 27. The paper [46] is not completely clear whether the four shared variables inspected in line 27 need to be inspected in one atomic inspection or in some specific order. In any case, the algorithm *LH* inspects $4N$ shared bits at this point.

In the algorithm of Figure 2 with $R = 3$, the number of shared bits inspected at line 27 is the number of elements of the set $copy.p$. This number is always smaller than $3N$. It is almost equal to the number of currently competing threads, and therefore often much smaller than N.

The version of Section 6.1 additionally performs N inspections of `dw` at line 19, and fewer than N inspections at line 25. Therefore, in either version, the number of bits inspected is smaller than in algorithm *LH*, even if, for simplicity, we count an **await** statement as a single inspection.

For large N, in both our algorithms, it pays to treat the private Boolean arrays *copy* and *dwcopy* as sets, and to implement them (e.g.) as stacks. In this way, the loop(s) at the lines (24, 25 and) 26, 27 can be much shorter than N iterations. A similar optimization is not useful for the algorithm *LH* because, in *LH*, on the long run, the average number of true bits in `turn` is N, because it is modified only by toggles to `turn` in line 24b.

7 Applying a Proof Assistant

The verification that some predicate is preserved under all possible steps always requires a case distinction. Usually, there are many innocent cases, and some or several cases that requires special consideration or additional arguments. Mechanical proof assistants are much better in separating the innocent cases from the interesting ones than human verifiers.

Throughout the project we use the proof assistant PVS [53]. We have verified both the algorithm of Figure 5 as presented in the Sections 3, 4, and 5, and the variation presented in Section 6.1. The proof scripts are available at [28].

The first theory `mx4bitS` concerns the algorithm of Figure 5 with the proofs of safety, i.e., mutual exclusion, FCFS, and absence of deadlock.

The first thing to do is to define the state space, and then the step relations for the individual steps. A sample of the step relations is given above in Section 3.4. The step relation itself is defined as a disjunction (union) of all steps at all labels. We then define each invariant separately, and prove its invariance. For instance, predicate *Kq0* is defined by

```
kq0(q, x): bool =
  x`central(q) IFF 25 <= x`pc(q) AND x`pc(q) <= 34
```

Here `x`central` is the Boolean function corresponding to the set `central` in state `x`. Preservation of this predicate is proved in

```
kq0_step: LEMMA
  kq0(q, x) AND step(p, x, y) IMPLIES kq0(q, y)
```

Such a lemma means the implicit universal quantification over the free variables `p`, `q` of type *Thread* and `x`, `y` of type *State*. In this case, PVS is able to prove the lemma by the single command `(grind)`, which performs all case distinctions required.

We mentioned in Section 3.6 that every index $k \in \mathit{Range}$ has a unique expression $k = q \cdot R + e$ with $q \in \mathit{Thread}$ and $e \in \mathit{Version}$. In order to avoid many PVS applications of this fact, we model the arrays `turn`, *turnset.q* and *copy.q* for threads q, as Boolean functions on the product type $\mathit{Thread} \times \mathit{Version}$. Therefore, predicate *Kq1* is defined by

```
kq1(q, r, x): bool =
  x`predec(q)(r) IMPLIES
  x`turnset(q)(r, x`nx(r)) OR x`copy(q)(r, x`nx(r))
```

Preservation of *Kq1* provided the precondition satisfies *Kq0*, *Kq4*, and *Kq5*, is expressed by

```
kq1_step: LEMMA
  kq1(q, r, x) AND step(p, x, y)
  AND kq0(r, x) AND kq4(q, r, x) AND kq5(r, x)
  IMPLIES kq1(q, r, y)
```

In order to prove this result, we first prove the much simpler result that *Kq1* is preserved without additional preconditions except at the lines 23, 27, and 35, as is shown by

```
kq1_step_rest: LEMMA
  kq1(q, r, x) AND step(p, x, y)
  IMPLIES kq1(q, r, y) OR step23B(p, x, y)
  OR step27(p, x, y) OR step35(p, x, y)
```

The lines 23, 27, 35 are treated one by one. For instance, line 23 (more precisely: the loop body, not the step to line 24) is treated in

```
kq1_step_23: LEMMA
  kq1(q, r, x) AND step23B(p, x, y)
  AND kq4(q, r, x) AND kq5(r, x)
  IMPLIES kq1(q, r, y)
```

When this has been done, lemma `kq1_step` can be proved. This way of attacking the proof obligations makes it much easier to restore the proof when the algorithm or invariants are modified.

When we have defined and proved all invariants, we construct the global invariant and prove its inductivity as a verification of consistency. Here, we also construct the initial state and prove that it satisfies the global invariant. Finally, Theorem 2 (no immediate deadlock) is proved. All this is contained in one PVS theory of 82 lemmas. For the proof of absence of immediate deadlock we need the obvious invariant that every thread is always in $\{21 \ldots 35\}$. We did not mention this invariant before because human readers tend to take this for granted.

The proof of liveness under weak fairness given in Section 5 is verified in a second PVS theory, `mx4bitL`, which is based on the first one. This theory consists of 33 lemmas. It is based on one general theory about functions from integers to natural numbers with in total 8 lemmas. The combination of the theories is contained in a single PVS dump file available at [28]. This site also contains a dumpfile that verifies the variation presented in Section 6.1.

It is a lot of work to create the proof in this way, but when it is done, it gives confidence, and the results can often be reused in modified situations. Indeed, initially we found a different list of invariants for a different algorithm. We then refactored the proof with cleaner invariants to accommodate a top-down presentation. Then we simplified the algorithm and refactored again. Finally, we designed the variation of Section 6.1, submitted a corresponding version to PVS, and adapted the verification such that it was proved as well. The line numbers starting with 21 and the invariants with a `q` in the name are used for the ease of query-replacing in this refactoring process.

Note that in the mechanical proof of safety, we exclusively argue in terms of the states, the step relation, the initial state, the invariants, and the variant functions. It is only for the proof of liveness that we define executions and the temporal operators $\Box$ and $\Diamond$.

8 Conclusions

Our algorithm is simpler than the one of [46], especially in the waiting conditions. It requires four nonatomic shared bits per thread. In the case of many threads, we have a possibly more efficient variation with five bits per thread.

The global invariant used for the proof of the algorithm is the conjunction of the universal quantifications of the invariants of the five families *Iq**, *Kq**, *Lq**, and *Mq**. Of these so-called constituant invariants, only seven are not inductive, namely *Iq2*, *Kq1*, *Kq5*, *Lq3*, *Mq2*, *Mq3*, and *Mq4*. The invariants *MX* and *FCFS* are implied by the global invariant.

The number of invariants is a measure of the complexity of the proof, but not a very precise one: it can be made smaller by combining all invariants in the global invariant, or by combining some invariants in a single predicate. On the other hand, it can be made bigger, e.g., by distributing *Mq2* over the three conjuncts of its consequent. Yet, it gives some idea of the difficulty of the algorithm.

As shown, we organize the verification around the invariants rather than around the program text. In the method of the proof outlines, we would have to mention most of the invariants almost everywhere. The invariants *Iq3*, *Lq1*, *Mq1*, *Mq2* are the only ones that are only applicable at one or two labels. On the other hand, control information is essential: *Iq0*, *Kq1*, *Kq4*, *Kq5*, and *Mq0* are the only invariants that do not mention the control state. In our view, this indicates that the method of the proof outlines is inadequate for anything bigger than toy problems.

We need only eight simple invariants for the proofs of mutual exclusion and FCFS, even though the proof of FCFS requires the introduction of a history variable. More invariants are needed for the proof of absence of immediate deadlock. These are more complicated, and mirror the subtlety of the algorithm.

It is possible to formalize the splitting of the algorithms of [46] and of Section 6.1 into a mutual exclusion algorithm and an FCFS algorithm by performing separate verifications and then composing these in some way. Here, however, we have to agree with Lamport [45]: it would be a way to make the proof harder.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, 1991.
2. U. Abraham. Bakery Algorithms. In *Proceedings of the Concurrency, Specification, and Programming Workshop*, pages 7–40, 1993.
3. J.H. Anderson and M.G. Gouda. Atomic semantics of nonatomic programs. *Inf. Process. Lett.*, 28:99–103, 1988.
4. J.H. Anderson, Y.J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distr. Comput.*, 16:75–110, 2003.
5. G.R. Andrews. *Foundations of multithreaded, parallel, and distributed Programming.* Addison Wesley, Reading, etc., 2000.
6. K.R. Apt, F.S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, New York, 2009.
7. A.A. Aravind and W.H. Hesselink. A queue based mutual exclusion algorithm. *Acta Inf.*, 46:73–86, 2009.
8. A.A. Aravind and W.H. Hesselink. Nonatomic dual bakery algorithm with bounded tokens. *Acta Inf.*, 48:67–96, 2011.
9. E. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, 1975.
10. J.E. Burns. Complexity of communication among asynchronous parallel processes, 1981. Ph.D. thesis, School of Information and Computer Science, Georgia Institute of Technology.
11. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.
12. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification, Introduction to Compositional and Noncompositional Methods.* Cambridge University Press, Cambridge, 2001.
13. E.W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8:569, 1965.
14. E.W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112, London, etc., 1968. NATO Advanced Study Institute, Academic Press.
15. E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, 1974.
16. E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
17. S. Doherty, M. Herlihy, V. Luchangco, and M. Moir. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 31–39, New York, NY, USA, 2004. ACM Press.
18. H. Gao, J.F. Groote, and W.H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distr. Comput.*, 17:21–42, 2005.
19. H. Gao, J.F. Groote, and W.H. Hesselink. Lock-free parallel and concurrent garbage collection by mark&sweep. *Sci. Comput. Program.*, 64:341–374, 2007.
20. H. Gao and W.H. Hesselink. A general lock-free algorithm using compare-and-swap. *Information and Computation*, 205:225–241, 2007.
21. S. Haldar and P. Vitanyi. Bounded concurrent timestamp systems using vector clocks. *Journal ACM*, 49:101–126, 2002.
22. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, 2008.
23. W.H. Hesselink. Wait–free linearization with a mechanical proof. *Distr. Comput.*, 9:21–36, 1995.
24. W.H. Hesselink. A mechanical proof of Segall's PIF algorithm. *Formal Aspects of Computing*, 9:208–226, 1997.
25. W.H. Hesselink. The verified incremental design of a distributed spanning tree algorithm: extended abstract. *Formal Aspects of Computing*, 11:45–55, 1999.

26. W.H. Hesselink. Using eternity variables to specify and prove a serializable database interface. *Sci. Comput. Program.*, 51:47–85, 2004.
27. W.H. Hesselink. Universal extensions to simulate specifications. *Information and Computation*, 206:108–128, 2008.
28. W.H. Hesselink. Revisiting mutual exclusion by Lycklama-Hadzilacos, PVS scripts. `http://wimhesselink.nl/mechver/mx4bits`, 2010.
29. C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–583, 1969.
30. G.J. Holzmann. *The SPIN Model Checker, primer and reference manual.* Addison-Wesley, 2004.
31. T. Inoue, T. Hironaka, T. Sasaki, S. Fukae, T. Koide, and H.J. Mattausch. Evaluation of bank-based multiport memory architecture with blocking network. *Electronics and Communications in Japan*, 89:498–510, 2006.
32. A. Israeli and M. Li. Bounded time-stamps. *Distr. Comput.*, 6:205–209, 1993.
33. P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of Distributed Computing*, pages 285–294, New York, NY, USA, 2003. ACM Press.
34. P. Jayanti and S. Petrovic. Efficient wait-free implementation of multiword LL/SC variables. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS), June 2005*, pages 59–68. IEEE, 2005.
35. P. Jayanti, K. Tan, G. Friedland, and A. Katz. Bounding Lamport's Bakery algorithm. In *Proceedings of the SOFSEM*, volume 2234 of *LNCS*, pages 261–270, 2001.
36. Y.J. Kim and J.H. Anderson. Nonatomic mutual exclusion with local spinning. *Distr. Comput.*, 19:19–61, 2006.
37. P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in TLA. *Distr. Comput.*, 12:151–174, 1999.
38. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17:453–455, 1974.
39. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2:125–143, 1977.
40. L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1:84–97, 1979.
41. L. Lamport. The mutual exclusion problem – part I: a theory of interprocess communication, part II: Statement and solutions. *Journal ACM*, 33:313–348, 1986.
42. L. Lamport. On interprocess communication. Parts I and II. *Distr. Comput.*, 1:77–101, 1986.
43. L. Lamport. win and sin: predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst.*, 12:396–428, 1990.
44. L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, 1994.
45. L. Lamport. Composition: a way to make proofs harder. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: the significant difference*, volume 1536 of *LNCS*, pages 402–423. Springer, 1997.
46. E.A. Lycklama and V. Hadzilacos. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Trans. Program. Lang. Syst.*, 13:558–576, 1991.
47. N.A. Lynch. *Distributed Algorithms.* Morgan Kaufman, San Francisco, 1996.
48. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, New York, 1992.
49. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety.* Springer, New York, 1995.
50. M.M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In R. Guerraoui, editor, *18th International Symposium on Distributed Computing*, volume 3274 of *LNCS*, pages 144–158, 2004.
51. M.M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, 2004.
52. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Inf.*, 6:319–340, 1976.

53. S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference*, 2001. `http://pvs.csl.sri.com`
54. M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, 1986.
55. D.M. Rusinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Comput.*, 6:359–390, 1994.
56. W.-T. Shiue and C. Chakrabarti. Multi-module multi-port memory design for low power embedded systems. *Design Automation for Embedded Systems*, 9:235–261, 2004.
57. H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1438–1445. ACM Press, 2004.
58. M. Takamura and Y. Igarashi. Simple mutual exclusion algorithms based on bounded tickets on the asynchronous shared memory model. In *Proceedings of the COCOON*, volume 2387 of *LNCS*, pages 259–268, 2002.
59. G. Taubenfeld. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. In *Proceedings of the DISC*, volume 3274 of *LNCS*, pages 56–70, 2004.
60. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Pearson Education/Prentice Hall, 2006.
61. G. Tel. *Distributed Algorithms.* Cambridge University Press, 1994.
62. S. Vijayaraghavan. A variant of the bakery algorithm with bounded values as a solution to Abraham's concurrent programming problem. In *Proc. of Design, Analysis and Simulation of Distributed Systems*, 2003.
63. J. Welch, L. Lamport, and N.A. Lynch. A lattice-structured proof technique applied to a minimum-weight spanning tree algorithm. In *Proceedings 7th ACM Symposium on Principles of Distributed Computing*, 1988.
64. T-K. Woo. A note on Lamport's mutual exclusion algorithm. *SIGOPS Operating Systems Review*, 24 (4):78–81, 1990.
65. W. Zuo, Z. Qi, and L. Jiaxing. An intelligent multi-port memory. In *Proc. of the IEEE Intl. Symp. on Information Technology Application Workshops*, pages 251–254, 2008.