DOI 10.1007/s00165-016-0364-4

Formal Aspects of Computing (2016) 28: 325–341

Formal Aspects of Computing

Correctness and concurrent complexity of the Black-White Bakery Algorithm

Wim H. Hesselink

Johann Bernoulli Institute, University of Groningen, Groningen, The Netherlands

Abstract. Lamport's Bakery Algorithm (Commun ACM 17:453–455, 1974) implements mutual exclusion for a fixed number of threads with the first-come first-served property. It has the disadvantage, however, that it uses integer communication variables that can become arbitrarily large. Taubenfeld's Black-White Bakery Algorithm (Proceedings of the DISC. LNCS, vol 3274, pp 56–70, 2004) keeps the integers bounded, and is adaptive in the sense that the time complexity only depends on the number of competing threads, say N. The present paper offers an assertional proof of correctness and shows that the concurrent complexity for throughput is linear in N, and for individual progress is quadratic in N. This is proved with a bounded version of UNITY, i.e., by assertional means.

Keywords: Mutual exclusion, FCFS, Concurrent complexity, UNITY

1. Introduction

The advent of multiprocessors and multicore architectures has revived the interest in concurrent algorithms. Concurrent algorithms are difficult to design, however, because they can unexpectedly misbehave due to subtle bugs or race conditions. They are almost impossible to test. Verification is not easy either, but if one has a good proof assistant, it can be done.

A typical concurrency problem is mutual exclusion. Over the years, many mutual exclusion algorithms have been proposed. Recently, we performed an investigation [BDH15] of 20 of these algorithms: the algorithms were implemented and their performance compared, under both 0 and high contention. It turned out that some algorithms for mutual exclusion based on reading and writing of atomic variables perform almost as good as algorithms based on stronger hardware primitives. This justifies a renewed interest in the theoretical performance analysis of these algorithms.

One of the most elegant mutual exclusion algorithms ever proposed is Lamport's Bakery Algorithm [Lam74]. This algorithm has the so-called first-come first-served property (FCFS). In particular, it has no starvation. A disadvantage is that it requires unbounded integers. In 2004, Taubenfeld [Tau04] proposed the so-called Black-White Bakery Algorithm, which shares some of the good properties of the Bakery Algorithm, in particular FCFS, but does not need unbounded integers.

Correspondence and offprint requests to: W. H. Hesselink, E-mail: w.h.hesselink@rug.nl

There are several other mutual exclusion algorithms with FCFS. The most prominent one is the algorithm of Lycklama and Hadzilacos [LH91], with improvements in [Ara13, Hes13b, Hes15a]. These algorithms are special in that they use only five or four shared variables per thread, and that these variables are Boolean and need not be atomic (such variables are called safe [Lam86b], see also [LH91, Ara13, Hes13b, Hes15a]). The price paid is a higher concurrent complexity.

The idea of a concurrent complexity was proposed in [Hes98], in 1998. Recently, we saw that the concurrent complexity of an algorithm can be obtained as a side product of a progress proof in UNITY logic [CM88, Mis01]. This idea was applied for the first time in our verification [Hes15a] of the mutual exclusion algorithm of Lycklama–Hadzilacos–Aravind.

The present paper is devoted to the verification of Taubenfeld's Black-White Bakery Algorithm for mutual exclusion. It first transforms the algorithm in such a way that the atomic steps can be isolated. It subsequently verifies the safety properties of the algorithm by means of invariants. Finally, the paper proves progress and quantifies it by estimating the concurrent complexity. This proof of progress heavily relies on the safety properties obtained first. If N is an upper bound of the number of concurrently competing threads, the result is that the throughput, general progress, is proportional N, whereas the individual progress is proportional to N^2 (see Sect. 1.5 for definitions of these notions of progress). For the algorithm of Lycklama–Hadzilacos–Aravind these numbers are N times as large [Hes15a]. In other words, the Black-White Bakery Algorithm is significantly less sensitive to congestion than the LHA algorithm.

Overview

Mutual exclusion is introduced in Sect. 1.1. Section 1.2 deals with the atomicity of the commands. In Sect. 1.3, correctness is discussed in general terms. Concurrent complexity is introduced in Sect. 1.4. The problem setting and the resulting estimates for progress are presented in Sect. 1.5.

The algorithm is presented and discussed in Sect. 2. In Sect. 3, the algorithm is extended with an environment and translated to a transition system amenable to formal verification. This section further contains the proofs of the safety properties. The progress properties and the concurrent complexity are treated in Sect. 4. Conclusions are drawn in Sect. 5.

All results in this paper about the transition system have been verified with the proof assistant PVS [OSRSC01]. The proof script is available at [Hes15b], but this is not the focus here.

Novelty, generality, applicability

Proving safety properties by means of invariants for a transition system has been the accepted way to do it for 20 years. In our view, it is the only reliable way, but not all workers in the field seem to agree on this. The application of bounded UNITY to prove the progress properties of the Black-White Bakery Algorithm is new. The method was earlier applied to the LHA algorithm in [Hes15a], but not to other algorithms.

Both methods are applicable to all concurrent algorithms for shared memory systems, and also, after minor adaptation, to algorithms for message passing systems. The bottleneck lies in how difficult the algorithm and its verification are. A proof assistant helps, but the human user needs to understand the algorithm completely, the proof assistant primarily helps with exhaustive case distinctions and to keep track of proof obligations.

Finally, the paper makes Taubenfeld's algorithm better applicable, by giving a version of the algorithm (in Fig. 1) that is closer to application code than the code of [Tau04].

1.1. Mutual exclusion

The problem of mutual exclusion was proposed in 1965 by Dijkstra [Dij65]. It can be formulated as follows. Consider a system of concurrent threads or processes that can communicate by shared variables. From time to time these threads need exclusive access to some shared resource. Such exclusive access is called the *critical section CS*. When a thread is in the *CS*, other threads that need the resource must wait. *Mutual exclusion* is the design of an entry and exit protocol that protects the *CS* so that there is never more than one thread in the *CS*.

1.2. Atomicity

In a concurrent system, the atomic commands of the threads are interleaved in arbitrary ways. It is therefore important to specify the grain of atomicity of the commands. This must be done in such a way that it can be

respected by the implementation. According to the *principle of single critical reference* [OG76, (3.1)] and [AdBO09, p. 273], an atomic command shall read or write at most one shared variable (not both), unless it is specifically provided by the operating system (e.g. a CAS or a semaphore action). The principle serves to forbid (e.g.) atomic commands of the form `x := x + 1` when they are not explicitly provided by the operating system. Actions on private variables can be added to atomic commands because they never give interference.

When one has to implement an algorithm with fine-grain concurrency on hardware with a weak memory model, one may have to insert memory fences in the code to ensure that the intended atomicity is respected by the hardware. In Sect. 2, we describe how this can be done for the algorithm at hand.

1.3. Correctness

The correctness requirements of concurrent algorithms are distinguished in safety (no bad things happen) and progress (eventually something good happens). In general, the safety properties are (and must be) the first concern. For a mutual exclusion algorithm, this primarily means that there is never more than one thread in the critical section, and that the system cannot reach a deadlock state. There are two progress requirements: general progress, i.e. when there are threads that need to enter the critical section, eventually, some do (this is called deadlock freedom); and individual progress: any thread that needs to enter the critical section, eventually does so (this is called lockout freedom). Usually, the proof of progress needs several of the properties established in the proof of safety. The more reason to treat safety first and carefully.

1.4. Concurrent complexity

In principle, progress is unquantified, but for practical purposes it is useful to know that the progress to some well-defined goal does not take too much time. This leads to the questions of time-complexity.

Due to the many possible interleavings, it is not easy to come up with faithful time-complexity measure for concurrent algorithms. In [Hes98, Hes15a], we proposed a concept of concurrent complexity based on "rounds". This concept is closely tied to UNITY of [CM88, Mis01], in the sense that, in most cases, a progress proof with UNITY can easily be adapted to also give an upper bound of the concurrent complexity.

In the analysis of a concurrent algorithm, a transition system is constructed that models the algorithm, but also its environment, which contains the clients of the system. One therefore needs to distinguish two kinds of steps: the *forward* steps that are executed by the threads for the sake of the algorithm, and *environment* steps that model uncontrolable actions of the environment. In general, the steps of an algorithm are forward steps. See Sect. 3.3 for a more detailed discussion. Progress can be hampered by the disabling of forward steps, e.g., when a thread needs to wait for a semaphore. In general, disabling of environment steps improves performance. The distinction between forward steps and environment steps corresponds to Guarantee and Rely in Rely/Guarantee approaches.

An *execution fragment* is a nonempty finite sequence of states such that every pair of subsequent states is connected by a step of the transition system. Two execution fragments can be concatenated if the last state of the first fragment equals the first state of the second fragment. An execution fragment is called a *round* if, for every thread p, it either contains at least one forward step of p, or at least one state in which the forward steps of p are disabled. Informally speaking, in a round, every thread is scheduled at least once.

Finally, the concurrent complexity n of reaching a postcondition Q from a precondition P is expressed as the assertion "P leads to Q within n rounds", notation

$$P \;\; \mathbf{Lt}\,\langle n\rangle \;\; Q.$$

This is defined to mean that every execution fragment that starts in a state where P holds, and that contains a concatenation of n rounds, contains a state where Q holds. This concept of *leads-to-within* specializes the *leads-to* concept of UNITY and temporal logic.

Example. A mutual exclusion algorithm may satisfy the property

$$(p \;\mathbf{in}\; \mathit{Entry}) \;\; \mathbf{Lt}\,\langle 9\rangle \;\; (p \;\mathbf{in}\; \mathit{CS}).$$

This would mean that, when some thread p is in the entry protocol, it will enter the critical section within nine rounds. Note that the predicates p **in** *Entry* and p **in** *CS* need not be stable.

Remark. The approach implicitly requires a weak kind of scheduling fairness. There need not be a fair scheduler. Yet, if the next forward command of some thread is never done, we cannot expect progress of this thread, and the absence of the command may even block progress for all other threads. Therefore, in order to prove progress, we need some assumption that enforces so-called weak fairness. This assumption is built in by the idea of rounds.

1.5. Problem setting and progress estimates

The mutual exclusion problem is traditionally modelled as follows. The threads are in an infinite loop of the form:

Thread $(p) =$
  loop
    NCS ; $Entry$; CS ; $Exit$
  end.

Here, *NCS* and *CS* are given program fragments that stand for the noncritical section and the critical section, respectively. *NCS* need not terminate, *CS* is guaranteed to terminate. The problem is to implement *Entry* and *Exit* in such a way that the number of threads in *CS* is guaranteed to remain ≤ 1 (mutual exclusion). Lamport [Lam86a] also required that *Exit* be waitfree in the sense that every thread can pass *Exit* without waiting, in a bounded number of its own steps.

The progress requirement is that, when some thread has entered *Entry*, eventually some thread will enter *CS*. Individual progress (lockout-freedom) is the condition that, if some thread has entered *Entry*, eventually it will enter *CS*, go through *Exit*, and return to *NCS*.

The first-come-first-served property FCFS is defined as follows [Lam74]. It is required that the program fragment *Entry* is a sequential composition of two fragments *Doorway* and *Waiting*, such that *Doorway* is waitfree and that, when a thread has passed *Doorway*, it will enter *CS* before any other thread that is currently not in *Entry*. See Sect. 3.4 for the formalization we have used.

The Black-White Bakery (BWB) Algorithm is adaptive, in the sense that, if the number of competing threads is bounded by a number N, the concurrent complexity is bounded by a function of N. Two kinds of concurrent complexity are distinguished: throughput and individual progress.

Throughput is measured by a shared history variable `rc` (return counter) which is incremented with 1 whenever any thread returns to *NCS*. Let *AI* be the condition that all threads are idle, i.e., at *NCS*. Of course, there is no throughput when *AI* holds. A linear estimate of throughput is therefore a pair of constants A, B such that, for all i, m,

(0) $$m \leq \texttt{rc} \;\; \mathbf{Lt} \, \langle A \cdot i + B \rangle \;\; (m + i \leq \texttt{rc} \;\vee\; AI).$$

In words, given a number i, if the number of rounds is large enough ($A \cdot i + B$) and the threads do not become all idle, at least i times a thread returns to *NCS*. The number A is the *throughput factor*. The smaller it is the better the performance of the algorithm. The number B is a kind of initial delay. According to Theorem 3 below, for the BWB algorithm, throughput (0) holds with a throughput factor $A = \mathcal{O}(N)$.

Individual progress of thread p is expressed and quantified by

(1) $$\mathit{true} \;\; \mathbf{Lt} \, \langle n \rangle \;\; p \;\mathbf{at}\; NCS.$$

This says that, from every location, thread p returns to the noncritical section within n rounds. The number n is (an upper bound for) the individual delay. According to Theorem 4 below, for the BWB algorithm, individual progress (1) holds with individual delay $n = \mathcal{O}(N^2)$.

Remark. In the Formulas (0) and (1), passage of the critical section is assumed to take only one round.

2. The Black-White Bakery Algorithm

Our version of the Black-White Bakery Algorithm [Tau04, Fig. 3] is given in Fig. 1, where *thread* is the set of all threads. *Entry* consists of the lines 22–34; *Exit* consists of the lines 36–38. Shared variables are written in typewriter font, private variables are written slanted. Below, if v is a private variable, the value of v for thread p

outside the code is denoted by $v.p$. We use line numbers starting at 22 as a matter of convenience, see Sect. 3.1.

The object `partic` is a so-called *active set*, as introduced by Afek et al. [AST99]. It has three methods: *join*, *leave*, and *getset*. The method *getset* returns a set that contains all threads that have completed their last call of *join* and have not yet started *leave*, and that does not contain any threads that have completed *leave* and have not yet started a next call of *join*. Formally, the object can be regarded as a large boolean array. We come back to this in Sect. 3.1.

Entry to the critical section is guarded by two queues, distinguished by the shared variable $\texttt{color} : bit$. In line 25, an entering thread lines up in the queue of `color`. The threads in the queue of $1 - \texttt{color}$ have priority. Thread p computes its priority $lev.p$ in this queue in line 26. It announces the queue chosen and its priority by the assignment in line 27: the integer $\texttt{pair}\ (p)$ encodes the queue thread p has chosen, $mcol.p = \texttt{pair}(p)$ **mod** 2, as well as its priority $lev.p = \texttt{pair}(p)$ **div** 2 in this queue.

In order to prevent interference between the writing of `pair` in line 27 and the reading of `pair` in lines 33, 34, the doorway 22–29 of thread p is guarded by the boolean $\texttt{cho}(p)$, just as in Lamport's Bakery Algorithm.

The algorithm thus uses the shared variables:

$$\begin{array}{l}\texttt{color} : bit\ ;\\ \texttt{pair}: thread \to nat\ ;\\ \texttt{cho}, \texttt{partic} : thread \to bool.\end{array}$$

The initial condition is

$$\forall\, q \in thread : \texttt{cho}(q) = \texttt{partic}(q) = false\ \wedge\ \texttt{pair}(q) \in \{0, 1\}.$$

Initially, `color` can be arbitrary. Thread p only writes the array elements $\texttt{pair}(p)$, $\texttt{cho}(p)$, $\texttt{partic}(p)$.

The main communication variable is array `pair`. Thread p writes $\texttt{pair}(p)$ in the lines 27 and 37. It reads $\texttt{pair}(thr.p)$ in the lines 26, 32, 33, and 34. In the lines 26, 33, 34, thread p processes the value of $\texttt{pair}(thr)$ by means of private functions *fn*, *guardA*, *guardB*, which also use its private variables *mcol*, *lev*, and *thr*. The ordering used in *guardA* is the lexical ordering:

$$(x, y) \le (x', y') \quad \equiv \quad x < x'\ \vee\ (x = x' \wedge y \le y').$$

In the lines 30–34, thread p waits for any other participating thread *thr*, first to conclude its lines 24–28, next to conclude its waiting section if *thr* has priority over p. After this waiting section, thread p can enter the critical section *CS*. Subsequently, it resets `color`, but only when its private color $mcol.p$ equals the public `color`.

The algorithm of Fig. 1 deviates at two points from Algorithm 3 of [Tau04]. The latter algorithm violates the principle of single critical reference of Sect. 1.2. At the point of our line 25, it reads the shared variable *color* and immediately writes the value read to the shared variable $\texttt{pair}(p)$. For the sake of the verification, we need to identify the atomic action in such a way that the principle of single critical reference is satisfied. We therefore introduce a private variable *mcol* to hold the value read in line 25, and postpone the assignement to $\texttt{pair}(p)$ to line 27. A more innocent deviation is that the computation of the maximum over all threads in *set1* is split in a sequence of steps in line 26.

Note that the program now almost satisfies the principle of single critical reference: in every transition (i.e., at every line number) at most one shared variable is read or written, and not both. This is the reason to separate the lines 31 and 32. Strictly speaking, line 34 violates the principle because it inspects $\texttt{pair}(thr)$ and `color`. This is allowed, however, because the thread is waiting for a *disjunction*: it can pass when either of the disjuncts holds.

If one has to execute the algorithm on hardware with a weak memory model, one may have to insert fences after every write operation that is followed by a read operation. Therefore, Fig. 1 offers optional fences after the lines 23 and 28.

Remark. For the sake of simplicity, or when the set of threads is small enough, one can remove the variable `partic` and the lines 22 and 38. In the lines 24 and 29, `partic` must then be replaced by the set *thread* of all threads. The result is more or less equivalent to Fig. 2 of [Tau04].

[t]

```
Thread(p : thread) =
        set1, set2 : set[thread] ,
        mcol : bit, lev : posnat, thr : thread ;
22      join(partic) ;
23      cho(p) := true ; (fence) ;
24      set1 := getset(partic) − {p} ;
25      mcol := color ; lev := 1 ;
26      for each thr ∈ set1 do lev := max(lev, fn(pair(thr))) endfor ;
27      pair(p) := 2 · lev + mcol ;
28      cho(p) := false ; (fence) ;
29      set2 := getset(partic) − {p} ;
30      for each thr ∈ set2 do
31          await ¬ cho(thr) ;
32          if pair(thr) mod 2 = mcol then
33              await guardA(pair(thr)) ;
            else
34              await guardB(pair(thr)) ∨ color ≠ mcol ;
            endif ;
        endfor ;
35      CS ;
36      color := 1 − mcol ;
37      pair(p) := mcol ;
38      leave(partic) ;
    where
fn(n) = 1 + (n mod 2 = mcol ? n div 2 : 0) ,
guardA(n) = (n div 2 = 0 ∨ n mod 2 ≠ mcol ∨ (lev, p) ≤ (n div 2, thr)) ,
guardB(n) = (n div 2 = 0 ∨ n mod 2 = mcol) .
```

Fig. 1. Taubenfeld's Black-White Bakery Algorithm

3. Verification of safety

In order to verify the BWB algorithm, it is modelled as a transition system with a global state that comprises the values of all shared and private variables, including program counters. In this system, the threads perform steps in arbitrary order. This transition system is then used to prove the relevant safety and liveness properties.

The transition system is developed in Sect. 3.1. Section 3.2 contains the proof of mutual exclusion. Absence of deadlock states is proved in Sect. 3.3. The FCFS property is proved in Sect. 3.4. In Sect. 3.5, it is proved that the communication variables can remain bounded.

3.1. The transition system

The program of Fig. 1 is extended and transformed into the transition system of Fig. 2. This is a formalization step, not subject to verification by PVS. Indeed, Fig. 2 is the starting point of the PVS verification.

First, at line 21, a noncritical section *NCS* has been added, where thread p resides initially. This is also the location thread p goes back to after line 38. The decision at *NCS* to aim at the critical section and to go to line 22 is an environment step because it is done by the client of the system.

During the design and verification of an algorithm, we occasionally have to change line numbers and numbered invariants. To avoid introducing mistakes in the PVS proof when modifying the files with query-replace, we use line numbers of two digits. Therefore, in Fig. 2, the transitions are numbered from 21 onward (the choice of 21 is arbitrary). Every thread has a private variable *pc* that holds the current line number. Every transition of thread

p implicitly increments $pc.p$, unless this is overridden by a branch or goto instruction.

We thus use the line numbers to refer to the steps of the algorithm. We distinguish the steps at line 26 into step 26B, the execution of the loop body (which does not change pc), and step 26E, the jump to line 27 when *set1* is empty. Similarly, step 30B goes to line 31, while step 30E jumps to line 35 when *set2* is empty. Note that, in Fig. 2, the variables *set1* and *set2* change in the loop bodies: they now serve to hold the threads for which the loop body has yet to be executed.

In order to verify the FCFS property, we let thread p register, when it becomes competing, the threads that it must give priority to in the ghost variable $\texttt{predec}(p)$. When p leaves the *CS*, it disclaims all its priorities by removing itself from the sets $\texttt{predec}(q)$. When thread p becomes idle again, in line 38, it increments a private ghost variable $cnt.p$. We come back to this below in the Sects. 3.4 and 4.3, respectively.

In lines 22 and 38, the operations *join* and *leave* are modelled as a flickering assignments of *true* and *false*, respectively. Such a flickering assignment

$$\texttt{partic}(p) := \text{(flickering) } E;$$

is modelled as a repeated nondeterministic choice

$$\begin{array}{ll} \ell: & (\quad \texttt{partic}(p) := \mathit{arbitrary};\ \ \textbf{goto}\ \ell\ ; \\ & [\!]\ \ \texttt{partic}(p) := E). \end{array}$$

Formally, fairness is used to imply that this repetition terminates. This treatment of the set `partic` as a safe variable in the sense of Lamport [Lam86b] precisely captures the properties postulated in Sect. 2. See also [Hes13a, Section 1.4].

For the ease of verification, the array `pair` is split into arrays `col` and `num` with

$$\texttt{pair}(q) = 2 \cdot \texttt{num}(q) + \texttt{col}(q)\ \ \wedge\ \ \texttt{num}(q) \in \mathbb{N}\ \ \wedge\ \ \texttt{col}(q) \in \{0, 1\}.$$

Therefore, line 27 now holds a concurrent assignment to fields of these arrays, and line 37 only resets $\texttt{num}(p)$.

3.2. Proof of mutual exclusion

Mutual exclusion is the property that there are never more than one thread in the *CS*, i.e., if thread q is in *CS*, any thread (say r) in *CS* equals q:

$$MX\text{:}\qquad q\ \textbf{in}\ CS\ \wedge\ r\ \textbf{in}\ CS\ \Rightarrow\ q = r.$$

Implicitly, by postulating such an invariant, we mean that it should hold for all values of the free variables (here q and r).

Remark. Predicate MX expresses mutual exclusion in an idealized environment. One may employ a Rely/Guarantee framework (e.g. [NLWSD14]) to express how clients of the data structure can benefit from this. This falls out of the scope of this paper, and it would be the same for almost all mutual exclusion algorithms.

In the invariants, we use q (and r) as free variables of type thread. In the discussion, we use p for the acting thread, because an invariant about q (and r) can be falsified by actions of any thread p. Of course, p, q, r always range over all threads, and equalities between them are not excluded.

In order to prove that MX is indeed invariant, we need to establish quite a number of other invariants. There are two ways of finding invariants: either bottom-up by looking at the algorithm, or top-down by weakening the required invariant (here MX). For the present algorithm, we begin with a bottom-up approach.

As thread q is the only one that writes the fields $\texttt{partic}(q)$, $\texttt{cho}(q)$, $\texttt{num}(q)$, $\texttt{col}(q)$, we clearly have the invariants

$$\begin{array}{ll} \mathit{Iq0}\text{:} & q\ \textbf{in}\ \{23 \ldots 37\}\ \Rightarrow\ \texttt{partic}(q), \\ \mathit{Iq1}\text{:} & q\ \textbf{in}\ \{24 \ldots 28\}\ \Rightarrow\ \texttt{cho}(q), \\ \mathit{Iq2}\text{:} & q\ \textbf{in}\ \{28 \ldots 37\}\ \Rightarrow\ \texttt{num}(q) = \mathit{lev}.q > 0, \\ \mathit{Iq3}\text{:} & q\ \textbf{in}\ \{28 \ldots\}\ \Rightarrow\ \texttt{col}(q) = \mathit{mcol}.q. \end{array}$$

Similarly, the variables *set1* and *set2* satisfy the invariants

$$\begin{array}{ll} \mathit{Iq4}\text{:} & q\ \textbf{in}\ \{27 \ldots\}\ \Rightarrow\ \mathit{set1}.q = \emptyset, \\ \mathit{Iq5}\text{:} & q\ \textbf{in}\ \{35 \ldots\}\ \Rightarrow\ \mathit{set2}.q = \emptyset. \end{array}$$

[t]

```
Thread(p : thread) =
21    NCS ; predec(p) := { q | q in {30 ... 35} } ;
22    partic(p) := (flickering) true ;
23    cho(p) := true ;
24    set1 := partic − {p} ;
25    mcol := color ; lev := 1 ;
26    while exists thr ∈ set1 do
          if col(thr) = mcol ∧ lev ≤ num(thr)
          then lev := num(thr) + 1 endif ;
          remove thr from set1 ;
      endwhile ;
27    num(p) := lev ; col(p) := mcol ;
28    cho(p) := false ;
29    set2 := partic − {p} ;
30    while exists thr ∈ set2 do
31        await ¬ cho(thr) ;
32        if col(thr) = mcol then
33            await num(thr) = 0 ∨ col(thr) ≠ mcol
                  ∨ (lev, p) ≤ (num(thr), thr) ;
              remove thr from set2 ;
          else
34            await num(thr) = 0 ∨ col(thr) = mcol ∨ color ≠ mcol ;
              remove thr from set2 ;
          endif ;
      endwhile ;
35    CS ; for each q do remove p from predec(q) endfor ;
36    color := 1 − mcol ;
37    num(p) := 0 ;
38    partic(p) := (flickering) false ; cnt := cnt + 1 ; goto 21 .
```

Fig. 2. The transition system

After this preparation, we take a top-down approach. As announced, the competing threads q with $mcol.q \neq$ color have priority over those with $mcol.q =$ color. This may suggest the predicate

Jq0a: q **in** $\{35 \ldots 37\} \ \wedge \ r$ **in** $\{26 \ldots 37\} \ \wedge \ mcol.q =$ color
$\Rightarrow \quad mcol.r =$ color.

This predicate easily follows from Iq5 and the postulate

Jq0: q **in** $\{30 \ldots 37\} \ \wedge \ r$ **in** $\{26 \ldots 37\} \ \wedge \ mcol.q =$ color
$\Rightarrow \quad mcol.r =$ color $\vee \ r \in set2.q$.

We turn to the proof that *Jq0* is indeed an invariant. This proof was constructed using the proof assistant PVS. It requires human creativity to invent or generalize invariants, but the proof assistant is needed to verify obvious steps, to handle the numerous case distinctions, and to list proof obligations.

Initially both threads q and r are at line 21, so that *Jq0* holds. Predicate *Jq0* is threatened only by the steps 29, 33, 34, and 36. This means that, for all other steps of the transition system, the precondition *Jq0* implies that *Jq0* also holds in the postcondition. For the steps mentioned, we need additional information about the precondition to infer *Jq0* in the postcondition. Step 29 preserves *Jq0* because of *Iq0*. Step 33 preserves *Jq0* because of the new postulate

Jq1: q **at** 33 $\wedge \ thr.q$ **in** $\{26 \ldots 37\} \ \wedge \ mcol.q =$ color
$\Rightarrow \quad mcol.(thr.q) =$ color.

Step 34 preserves *Jq0* because of *Iq2*, *Iq3*, and the new postulate

Jq2: q **in** $\{32 \dots 34\} \ \wedge \ thr.q$ **in** $\{26 \dots 28\} \ \wedge \ mcol.q = \texttt{color}$
$\Rightarrow \quad mcol.(thr.q) = \texttt{color}.$

Indeed, step 34 threatens *Jq0* only when thread q does the step and $r = thr.q$, while r is in 26–37 and $mcol.q = \texttt{color} \neq mcol.r$. Then *Iq2* implies $\texttt{num}(r) > 0$ and *Iq3* implies $\texttt{col}(r) = mcol.r$, and hence $\texttt{col}(r) \neq mcol.q$. It follows that the guard of step 34 is false, and the step cannot be taken.

Step 36 of thread p preserves *Jq0* for q and r because of *Iq5* and *Jq0*. Indeed, step 36 of thread p threatens *Jq0* for q and r only when p is at 36 and modifies color, and q is in 30–37 with $mcol.q \neq \texttt{color}$. As p modifies color, it has $mcol.p = \texttt{color}$. Therefore, *Jq0* for p and q implies that $q \in set2.p$, contradicting *Iq5*.

Predicate *Jq1* is threatened only by the steps 32 and 36. It is preserved by step 32 because of *Iq3* and *Jq2*, and by step 36 because of *Iq5* and *Jq0*. Similarly, predicate *Jq2* is threatened only by the steps 31 and 36. It is preserved by step 31 because of *Iq1*, and by step 36 because of *Iq5* and *Jq0*.

This concludes the proof of preservation of *Jq0*, and hence of *Jq0a*. Predicate *Jq0a* implies that if threads q and r are both in 35–37, then $mcolq = mcol.r$. It therefore remains to consider threads near *CS* with the same private colors. At this point, the algorithm is very similar to the Bakery Algorithm, see [Lam74] or e.g. [Hes13a].

We postulate the invariant

Jq3: q **in** $\{30 \dots 37\} \ \wedge \ r$ **in** $\{28 \dots 37\} \ \wedge \ mcol.q = mcol.r$
$\Rightarrow \quad (\texttt{num}(q), q) \leq (\texttt{num}(r), r) \ \vee \ r \in set2.q.$

Predicate *Jq3* is threatened only by the steps 27, 29, 33, and 34. It is preserved by step 27 because *Iq2*, *Iq4*, and the new postulate

Jq4: q **in** $\{30 \dots 37\} \ \wedge \ r$ **in** $\{26, 27\} \ \wedge \ mcol.q = mcol.r$
$\Rightarrow \quad \texttt{num}(q) < lev.r \ \vee \ r \in set2.q \ \vee \ q \in set1.r.$

Predicate *Jq3* is preserved by step 29 because of *Iq0*. It is preserved by step 33 because of *Iq2* and *Iq3*. It is preserved by step 34 because of the new postulate

Jq5: q **at** 34 $\wedge \ thr.q$ **in** $\{28 \dots 37\} \ \wedge \ mcol.q = mcol.(thr.q)$
$\Rightarrow \quad \texttt{num}(q) < \texttt{num}(thr.q).$

Predicate *Jq4* is threatened only by the steps 25, 26B, 29, 33, and 34. It is preserved by step 25 because of the new postulate

Jq6: q **in** $\{30 \dots 37\} \ \wedge \ r$ **at** 25 $\Rightarrow \ r \in set2.q \ \vee \ q \in set1.r.$

It is preserved by step 26B because of *Iq3*, by step 29 because of *Iq0*, and by the steps 33 and 34 because of the new postulate

Jq7: q **in** $\{32 \dots 34\} \ \wedge \ thr.q$ **in** $\{26, 27\} \ \wedge \ mcol.q = mcol.(thr.q)$
$\Rightarrow \quad \texttt{num}(q) < lev.(thr.q) \ \vee \ q \in set1.(thr.q).$

Predicate *Jq5* is threatened only by the steps 27 and 32. It is preserved by step 27 because of *Iq4* and *Jq7*, and by step 32 because of *Iq3*.

Predicate *Jq6* is threatened only by the steps 24, 29, 33, 34. It is preserved by the steps 24 and 29 because of *Iq0*. It is preserved by the steps 33 and 34 because of the new postulate

Jq8: q **in** $\{32 \dots 34\} \ \wedge \ thr.q$ **at** 25 $\Rightarrow \ q \in set1.(thr.q).$

Predicate *Jq7* is threatened only by the steps 25, 26B, 31. It is preserved by step 25 because of *Jq8*. It is preserved by step 26B because of *Iq3*. It is preserved by step 31 because of *Iq1*.

Predicate *Jq8* is threatened only by the steps 24 and 31. It is preserved by step 24 because of *Iq0*, and by step 31 because of *Iq1*.

This concludes the proof of preservation of *Jq3*. The invariants *Jq0*, *Iq5*, and *Jq3* together imply

MXX: q **in** $\{35 \dots 37\} \ \wedge \ r$ **in** $\{35 \dots 37\} \ \Rightarrow \ q = r.$

This says that mutual exclusion holds in the region 35–37, and in particular in *CS* (line 35). Therefore *MXX* implies mutual exclusion *M*X. This concludes the proof of mutual exclusion.

3.3. Absence of deadlock

A thread is said to be *idle* iff it is at line 21. A thread is said to be *competing* iff it is in 22–38. A step of the transition system is called a *forward* step if it starts in one of the lines 22–38 and either modifies *pc* or modifies the private variable *set1* (in case of line 26). A thread is said to be *enabled* if it can do a forward step.

The step from lines 21 to 22 is not a forward step but an environment step because this step is not part of the system that provides mutual exclusion, but it is done by a process using the system when it needs access to the critical section.

Note that idle threads cannot do forward steps, and that the only non-forward steps of a competing thread are flickering steps at lines 22 and 38.

It is easy to verify that thread p is enabled if and only if it satisfies the predicate

$$\begin{aligned} ena(p) \;=\; & p \textbf{ in } \{22 \ldots 38\} \\ & \wedge\ (p \textbf{ at } 31 \;\Rightarrow\; \neg\, \mathtt{cho}(r)) \\ & \wedge\ (p \textbf{ at } 33 \;\Rightarrow\; \mathtt{num}(r) = 0 \vee \mathtt{col}(r) \neq mcol.p \\ & \quad \vee\ (lev.p, p) \leq (\mathtt{num}(r), r)\ ) \\ & \wedge\ (p \textbf{ at } 34 \;\Rightarrow\; \mathtt{num}(r) = 0 \vee \mathtt{col}(r) = mcol.p \vee mcol.p \neq \mathtt{color}) \\ & \text{where } \; r = thr.p. \end{aligned}$$

The transition system is said to be in *deadlock* iff there are competing threads and no (competing) thread can do a forward step. *Absence of deadlock* means that deadlock states are not reachable.

In order to prove absence of deadlock, we observe the following obvious invariants:

Kq0: q **in** $\{21 \ldots 38\}$,
Kq1: $\mathtt{cho}(q) \;\Rightarrow\; q$ **in** $\{24 \ldots 28\}$,
Kq2: $\mathtt{num}(q) > 0 \;\Rightarrow\; q$ **in** $\{28 \ldots 37\}$.

Theorem 1 *Absence of deadlock. Assume that there are no enabled threads. Then all threads are idle.*

Proof As there are no enabled threads, it follows from *ena* and *Kq0* that all threads are at the lines 21, 31, 33, or 34. By *Kq1*, it follows that $\mathtt{cho}(q)$ is false for all threads q, so that all threads at line 31 are enabled. Therefore all threads are at the lines 21, 33, 34.

For every thread p at line 34, we have that p is not enabled, so that thread $r = thr.p$ satsfies $\mathtt{num}(r) > 0$ and $\mathtt{col}(r) \neq mcol.p = \mathtt{color}$; by *Kq2* and *Iq3*, this implies that r is at line 33 and has $\mathtt{col}(r) \neq \mathtt{color}$.

It follows that, if there is a thread at line 34, then the set

$$S0 = \{r \mid r \textbf{ at } 33 \ \wedge\ \mathtt{col}(r) \neq \mathtt{color}\}$$

is nonempty. Let $q \in S0$ be the minimal element for the lexical ordering, i.e., $(\mathtt{num}(q), q) \leq (\mathtt{num}(r), r)$ for all $r \in S0$. As thread q is disabled and at line 33, the thread $r = thr.q$ satisfies $\mathtt{num}(r) > 0$ and $\mathtt{col}(r) = mcol.q$. By *Kq2*, *Iq2*, *Iq3*, and the previous paragraph, it follows that $r \in S0$, and hence $(\mathtt{num}(q), q) \leq (\mathtt{num}(r), r)$, so that thread q is enabled (by *Iq2*). This proves there are no threads at line 34. Therefore all threads are at the lines 21 or 33.

Now consider the set $S1 = \{r \mid r \textbf{ at } 33\}$. If this set is nonempty, let q be the minimal element of this set for the lexical order. By the arguments of the previous paragraph, again, thread q is enabled. This implies that $S1$ is empty. Therefore all threads are at line 21, i.e., they are idle. □

3.4. First-come first-served

The first-come first-served property (FCFS) must be distinguished from first-in first-out (FIFO). The point is that, in almost all mutual exclusion algorithms, the moment of "first-in" cannot be communicated between the threads. The first-come first-served property is therefore defined by Lamport [Lam86b] in the following way. It is required that the entry part of the protocol is a sequential composition of two fragments Doorway and Waiting, such that Doorway is waitfree and that, when a thread has passed Doorway, it will enter *CS* before any other thread that is currently not in *Entry*.

In our case, Doorway is the frament of the lines 22–29, which is indeed waitfree, and Waiting is the loop 30–34. The ghost variable `predec` (set of predecessors) is introduced to verify FCFS. Any thread p that enters

Doorway at line 21, registers all threads in 30–35 in $\texttt{predec}(p)$. Every thread that exits *CS* removes itself from all sets $\texttt{predec}(q)$. Now FCFS is expressed by the condition that any thread q cannot exit Waiting before $\texttt{predec}(q)$ is empty, as formalized in the predicate

FCFS: q **in** $\{35\ldots\} \;\Rightarrow\; \texttt{predec}(q) = \emptyset$.

In order to prove predicate *FCFS*, we observe that it is logically implied by *Iq5* and the new postulate

Lq0: q **in** $\{30\ldots\} \;\Rightarrow\; \texttt{predec}(q) \subseteq set2.q$.

Predicate *Lq0* is threatened only by the steps 29, 33, and 34. It is preserved by step 29 because of *Iq0* and the new postulate

Lq1: $r \in \texttt{predec}(q) \;\Rightarrow\; r$ **in** $\{30\ldots35\}$.

It is preserved by step 33 because of *Iq2*, *Iq4*, *Lq1*, and the new postulates

Lq2: $r \in \texttt{predec}(q) \;\wedge\; q$ **in** $\{26\ldots\} \;\wedge\; \texttt{col}(r) = mcol.q$
$\Rightarrow\quad \texttt{num}(r) < lev.q \;\vee\; r \in set1.q$,
Lq3: $thr.q \in \texttt{predec}(q) \;\wedge\; q$ **at** 33 $\;\Rightarrow\; \texttt{col}(thr.q) = mcol.q$.

Indeed, step 33 threatens *Lq0* only when q does the step and $r = thr.q \in \texttt{predec}(q)$. Then *Lq1* implies that r is in 30-35, and *Iq2* implies $\texttt{num}(r) > 0$. *Lq3* implies $\texttt{col}(r) = mcol.q$. Therefore *Lq2* together with *Iq4* imply $\texttt{num}(r) < lev.q$. It follows that the guard of step 33 of q is false.

Predicate *Lq0* is preserved by step 34 because of *Iq2*, *Lq1*, and the new postulate

Lq4: $thr.q \in \texttt{predec}(q) \;\wedge\; q$ **at** 34
$\Rightarrow\quad \texttt{col}(thr.q) \neq mcol.q \;\wedge\; mcol.q = \texttt{color}$.

Predicate *Lq1* is inductive. It holds initially and is preserved in every step.

Predicate *Lq2* is threatened only by the steps 25 and 27. It is preserved by step 25 because of the new postulate

Lq5: $r \in \texttt{predec}(q) \;\wedge\; q$ **at** 25 $\;\Rightarrow\; r \in set1.q$.

It is preserved by step 27 because of *Lq1*.

Predicate *Lq3* is threatened only by step 27, and it is preserved because of *Lq1*.

Predicate *Lq4* is threatened only by the steps 27, 32, 36. It is preserved by step 27 because of *Lq1*. It is preserved by step 32 because of the new postulate

Lq6: $r \in \texttt{predec}(q) \;\wedge\; q$ **in** $\{26\ldots\} \;\Rightarrow\; \texttt{col}(r) = mcol.q \;\vee\; \texttt{color} = mcol.q$.

Predicate *Lq4* is preserved by step 36 because of *Iq3*, *Iq5*, *Jq0*, and *Lq1*.

Predicate *Lq5* is threatened only by step 24. It is preserved because of *Iq0* and *Lq1*.

Predicate *Lq6* is threatened only by the steps 27 and 36. It is preserved by step 27 because of *Lq1*. It is preserved by step 36 because of *Iq3*, *Iq5*, *Jq0*, and *Lq1*.

This concludes the proof of the invariants *Lq**, and thus of *FCFS*.

3.5. Bounding the tickets

The Black-White Bakery Algorithm was designed as a remedy for the unbounded integers needed in the original Bakery Algorithm [Lam74]. This is verified by the next result.

Theorem 2 *Assume that the number of competing threads is always bounded by some number N. Then the tickets $\texttt{num}(q)$ are also bounded by N.*

Proof In order to prove this, the transition system is parametrized with the number N, and step 21 is forbidden whenever there are N competing threads (i.e., threads not at line 21). This implies the invariant

Mq0: $\#competing \leq N$.

The theorem is proved by distinguishing the threads that hold the current `color` from those that do not. For the first class, we define the set

$$TCol = \{q \mid q \textbf{ in } \{28 \ldots 37\} \ \wedge \ mcol.q = \texttt{color}\},$$

and postulate the invariant

Mq1: $\quad mcol.q = \texttt{color} \ \Rightarrow \ \texttt{num}(q) \leq \#TCol.$

Predicate *Mq1* is threatened only by the steps 25, 27, 36, 37. It is preserved by step 25 because of *Kq2*. It is preserved by step 27 because of the new postulate

Mq2: $\quad mcol.q = \texttt{color} \ \wedge \ q \textbf{ in } \{26, 27\} \ \Rightarrow \ lev.q \leq \#TCol + 1.$

Note that when thread q satisfies the antecedent of *Mq2* and executes step 27, it sets $\texttt{num}(q) := lev.q$, but it also enters the set TCol and hence increments $\#TCol$.

Predicate *Mq1* is preserved by step 36 because of *Iq5*, *Jq0*, *Kq2*. It is preserved by step 37 because of *Iq3* and the new postulate

Mq3: $\quad q \textbf{ at } 37 \ \Rightarrow \ mcol.q \neq \texttt{color}.$

Predicate *Mq2* is threatened only by the steps 26B, 36, 37. It is preserved by step 26B because of *Iq3*, *Kq2*, and *Mq1*. It is preserved by step 36 because of *Iq5* and *Jq0*. It is preserved by step 37 because of *Mq3*.

Predicate *Mq3* is threatened only by step 36. It is preserved because of *MXX* (this was the reason for introducing *MXX* next to *MX*).

For the second class, we define the set

$$NCol = \{q \mid q \textbf{ in } \{26, 27\} \ \wedge \ mcol.q \neq \texttt{color}\},$$

and postulate the invariant

Mq4: $\quad mcol.q \neq \texttt{color} \ \Rightarrow \ \texttt{num}(q) + \#NCol \leq N.$

Predicate *Mq4* is threatened only by the steps 27 and 36. It is preserved by step 27 because of the new postulate

Mq5: $\quad mcol.q \neq \texttt{color} \ \wedge \ q \textbf{ in } \{26, 27\} \ \Rightarrow \ lev.q + \#NCol \leq N + 1.$

It is preserved by step 36 because of *Mq0* and *Mq1*.

Predicate *Mq5* is threatened only by the steps 26B and 36. It is preserved by step 26B because of *Iq3*, *Kq2*, and *Mq4*. It is preserved by step 36 because of *Mq0* and *Mq2*. Finally, the predicates *Mq0*, *Mq1*, and *Mq4* together imply $\texttt{num}(q) \leq N$. This proves the theorem. □

It follows that $\texttt{pair}(q) \leq 2 \cdot N + 1$ always holds.

4. Progress

Progress of the algorithm is expressed in operational semantics, presented in Sect. 4.1. The operational progress assertions, however, are not proved by operational arguments but by means of "bounded UNITY" [Hes15a], presented in Sect. 4.2.

We proceed with an investigation of the quantitative throughput in Sect. 4.3, and of individual progress in Sect. 4.4, both under the assumption of Sect. 3.5 that the number of competing threads is bounded by N.

4.1. Formal operational semantics

The *state* of the system is given by the values of all shared and private variables. Usually, we prefer to keep the state implicit, but formally all invariants are boolean functions of the state. We let X be the set of all states. If P is a predicate on the state, it is also regarded as the subset of X where predicate P holds. $P \subseteq Q$ therefore means that every state that satisfies P also satisfies Q (i.e. that P implies Q). Let *start* be the initial predicate, i.e., the set of initial states.

For thread p, relation $step(p)$ is defined as the set of the pairs (x, y) of states such that in state x thread p can do a step of the algorithm that results in state y. Relation *step* is defined as the union of the relations $step(p)$ for all threads p, together with the identity relation of the state space. An *execution* is defined to be an infinite sequence xs of states with $xs_0 \in start$, and $(xs_n, xs_{n+1}) \in step$ for all $n \in \mathbb{N}$. A predicate P is an *invariant* if and only if it contains all states of all executions. We write $X_0 \subseteq X$ for the intersection of all invariants obtained. So this is the set of the states that satisfy all invariants obtained in Sect. 3.

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

Coming back to the algorithm, recall from Sect. 3.3 that the *forward steps* are defined to be the steps 22–38 that modify *pc* or *set1*. Relation $fwd(p) \subseteq step(p)$ is defined to be the set of forward steps of thread p. Thread p is therefore *enabled* in state x if and only if there is a state y with $(x, y) \in fwd(p)$. Recall that enabledness is expressed by the predicate $ena(p)$.

An *occurrence* of thread p in an execution fragment $(xs_0 \ldots xs_n)$ is a number i with $0 \leq i < n$, and $(xs_i, xs_{i+1}) \in fwd(p)$ or $xs_i \notin ena(p)$. The execution fragment is called a *round* if it contains an occurrence of every thread. In other words, in the fragment, every thread is scheduled, and either executed or found to be disabled. This applies, e.g., when thread p is always at line 21.

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

4.2. UNITY and bounded UNITY

UNITY logic [CM88, Mis01] is a way to systematically prove assertions of the form P *leads to* Q (notation $P \mapsto Q$), meaning "if P holds at any time t during a computation, Q will hold at some time $t' \geq t$".

Example. Individual progress of the algorithm means that a thread, say p, in the entry protocol, will eventually reach the critical section at line 35. This is expressed by: p **in** $\{22 \ldots 34\} \mapsto p$ **at** 35. □

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

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

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

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

$$P \textbf{ unless } Q \equiv (P \wedge \neg Q \wedge X_0) \textbf{ co} (P \vee Q),$$
$$P \textbf{ ensures } Q \equiv (P \textbf{ unless } Q) \wedge ((P \wedge \neg Q \wedge X_0) \textbf{ co!}\ Q).$$

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

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

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

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

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

There is also the Progress-Safety-Progress Rule [CM88]:

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

The soundness of these proof rules has been proved mechanically [Hes15a]. The set of proof rules is not complete, but they are enough for the present purposes.

Some progress properties are easily expressed by means of a numerical measure. For instance, as discussed in Sect. 1.5, the throughput of a mutual exclusion algorithm can be expressed by the growth of the sum `rc`, see Formula (0). We develop a small theory to estimate the growth of such a function.

A numerical state function $vf : X \to \mathbb{Z}$ is called a *forward measure* if it satisfies the following three requirements:

(2) $$\begin{array}{l}(x, y) \in \mathit{step} \wedge x \in X_0 \;\Rightarrow\; \mathit{vf}(x) \le \mathit{vf}(y),\\ (x, y) \in \mathit{fwd}(p) \wedge x \in X_0 \;\Rightarrow\; \mathit{vf}(x) < \mathit{vf}(y),\\ (x, y) \in \mathit{step} \wedge x \in \mathit{ena}(p) \cap X_0 \;\Rightarrow\; y \in \mathit{ena}(p) \;\vee\; \mathit{vf}(x) < \mathit{vf}(y).\end{array}$$

The importance of a forward measure vf is that it is guaranteed to grow with the number of rounds, unless all threads are disabled, in the sense that

(3) $$m \le \mathit{vf} \;\; \mathbf{Lt}\, \langle n \rangle \;\; (m + n \le \mathit{vf} \;\vee\; \neg(\exists p : \mathit{ena}(p))).$$

Useful progress properties are rarely coupled directly to the number of rounds. It can happen, however, that a useful progress property is measured by an integer valued state function svf that is proportional to a forward measure vf, via

(4) $$F \cdot \mathit{svf} \le \mathit{vf} < F \cdot \mathit{svf} + D,$$

for some factor $F > 0$ and some delay $D > 0$.

If the Formulas (3) and (4) hold, they imply that

(5) $$m \le \mathit{svf} \;\; \mathbf{Lt}\, \langle F \cdot i + D - F \rangle \;\; (m + i \le \mathit{svf} \;\vee\; \neg(\exists p : \mathit{ena}(p))).$$

Roughly speaking, this says that svf grows in n rounds with at least $(n - D)/F$. In the limit where the initial delay D counts no longer, svf grows at least with a speed F^{-1}.

4.3. Throughput

The throughput of the algorithm is defined as the number of times threads come back to the noncritical section. To measure this, a private ghost variable $cnt.p$ is introduced which is incremented in line 38, see Fig. 2. The throughput during an execution fragment is the growth of the sum $\mathtt{rc} = \sum_p cnt.p$ over all threads p, see Sect. 1.5.

Before analysing the growth of `rc`, we note some more invariants. As announded, we assume the invariant *Mq0* of Sect. 3.5. It is easy to see that this implies

Mq0a: $q \;\mathbf{in}\; \{23 \ldots 37\} \;\Rightarrow\; \#(\mathtt{partic} - \{q\}) \le N - 1.$

Using this, it is easy to verify the invariants

Nq0: $q \;\mathbf{in}\; \{25 \ldots\} \;\Rightarrow\; \#\mathit{set1}.q \le N - 1,$
Nq1: $q \;\mathbf{in}\; \{30 \ldots\} \;\Rightarrow\; \#\mathit{set2}.q \le N - 1.$

We also need the obvious invariants

Nq2: $q \;\mathbf{in}\; \{31 \ldots 34\} \;\Rightarrow\; \mathit{thr}.q \in \mathit{set2}.q,$
Nq3: $q \notin \mathtt{predec}(q).$

The steps of thread q are counted approximately by the function

$$\begin{array}{rl}\mathit{lvf}(q) = & \mathit{pc}.q - 21\\ & + (\mathit{pc}.q \ge 25\,?\; N - 1 - \#\mathit{set1}.q)\\ & + 4 \cdot (\mathit{pc}.q \ge 30\,?\; N - 1 - \#\mathit{set2}.q)\\ & - (\mathit{pc}.q \ge 35\,?\; 4 : (\mathit{pc}.q = 34\,?\; 1 : 0)).\end{array}$$

It follows from *Kq0*, *Nq0*, *Nq1*, that *l*vf is bounded by

$$0 \le lvf(q) < A \text{ where } A = 5 \cdot N + 9.$$

The function $lvf(q)$ increases under most steps of thread q. More precisely, it decreases under step 38, it remains constant under the flickering steps of lines 22 and 38, and it increases in all other steps. For the steps 24 and 29, this follows from *Mq0a*. For the backward jumps from the lines 33 and 34 to line 30, it follows from *Nq2*. All steps of threads $\neq q$ leave $lvf(q)$ constant.

The function lvf is connected to the ghost variable *cnt* in the function

$$avf(q) = lvf(q) + A \cdot cnt.q.$$

The bounds on *l*vf immediately imply

$$A \cdot cnt.q \le avf(q) < A \cdot (cnt.q + 1).$$

Function $avf(q)$ remains constant under the flickering steps of the lines 22 and 38, and it increases under all other steps of thread q. This holds in particular for step 38. of thread q because of the bounds for *l*vf. All steps of threads $\neq q$ leave $avf(q)$ constant.

The sum $Savf = \sum_q avf(q)$ now satisfies the bounds

$$A \cdot \mathtt{rc} \le Savf \le A \cdot \mathtt{rc}.q + (A-1) \cdot N. \tag{6}$$

The lefthand inequality is easy. The righthand inequality follows from *Mq0* and the fact that $lvf(q) = 0$ when q is not competing.

The function *Savf* remains constant under the flickering steps, and it increases under all other steps of all threads. If thread p is enabled and it does a flickering step, it remains enabled. Therefore, function *Savf* is a forward measure, see Formula (2). By Formula (6), function `rc` is proportional to *Savf* with factor A and delay $(A-1) \cdot N + 1$. According to Theorem 1, when there are no enabled threads, all threads are idle. This means that

$$\neg(\exists p : ena(p)) \cap X_0 \subseteq AI,$$

where *AI* is the condition that all threads are idle. Therefore, Formula (5) implies:

Theorem 3 *Let* $B = (A-1) \cdot (N-1)$. *Then*

$$m \le \mathtt{rc} \ \textbf{\textit{Lt}} \langle A \cdot i + B \rangle \ (m + i \le \mathtt{rc} \ \vee \ AI).$$

In other words, the algorithm has a throughput factor $A = 5 \cdot N + 9$, linear in N; and throughput delay $B = (A-1) \cdot (N-1)$.

4.4. Individual progress

As the algorithm satisfies FCFS, individual progress follows from general progress, just as in the case of the algorithm of Lycklama–Hadzilacos–Aravind [Hes15a]. The key step is an application of the PSP rule.

The problem is to guarantee that a thread q in the region 30-35 eventually leaves this region. If it does not, all newly entering threads r will collect and keep q in $\mathtt{predec}(r)$. By *FCFS*, such threads cannot reach the critical section. This should contradict Theorem 3. To formalize this argument, consider the predicate $WF(q, m)$ given by

$$WF(q, m) = (q \textbf{ in } \{30 \ldots 35\} \ \wedge \ \mathtt{rc} + \#NP = m), \text{ where}$$
$$NP = \{r \mid r \textbf{ in } \{22 \ldots\} \ \wedge \ q \notin \mathtt{predec}(r)\}.$$

While thread q remains in 30-35, no thread r can enter *NP*, because when r enters $\{22 \ldots\}$, it puts q into $\mathtt{predec}(r)$. Thread r can only leave *NP* by executing line 38, i.e., by incrementing `rc`. Conversely, when thread r increments `rc`, it executes line 38 and therefore leaves the set *NP* because of *FCFS*. This proves that

$$WF(q, m) \ \textbf{unless} \ (q \textbf{ in } \{36 \ldots 38\}).$$

Theorem 3 with $i := N$, $m := m - N$, and $n_1 := A \cdot N + (A-1) \cdot (N-1)$ gives

$$(m - N \le \mathtt{rc}) \ \textbf{Lt} \langle n_1 \rangle \ (m \le \mathtt{rc} \ \vee \ AI).$$

Application of the PSP rule to these two formulas gives

(7) $WF(q, m)$ **Lt** $\langle n_1 \rangle$ $(q$ **in** $\{36 \ldots 38\})$.

In fact, the lefthand side of the PSP rule simplifies because $WF(q, m) \subseteq (m - N \leq \mathtt{rc})$ by *Mq0*. In the righthand side of the PSP rule, we have

$$((m \leq \mathtt{rc} \ \vee \ AI) \ \wedge \ WF(q, m)) \ = \ \emptyset$$

because, if q is in 30-35, then q is not idle and $q \in NP$ by *Nq3*.

By the Disjunction Rule, Formula (4.4) for all m gives

$$(q \textbf{ in } \{30 \ldots 35\}) \ \textbf{Lt} \ \langle n_1 \rangle \ \ (q \textbf{ in } \{36 \ldots 38\}).$$

It is more or less a matter of counting steps, to obtain the leads-to assertions

$$(q \textbf{ in } \{36 \ldots 38\}) \ \textbf{Lt} \ \langle 3 \rangle \ \ (q \textbf{ at } 21),$$
$$(q \textbf{ in } \{22 \ldots 29\}) \ \textbf{Lt} \ \langle N + 7 \rangle \ \ (q \textbf{ in } \{30 \ldots 35\}).$$

As $n_1 + 3 + (N + 7) = 10 \cdot N^2 + 13 \cdot N + 2$, the combination of the last three leads-to assertions by means of transitivity and disjunction gives the result on individual progress:

Theorem 4 *true* **Lt** $\langle 10 \cdot N^2 + 13 \cdot N + 2 \rangle$ $(q$ **at** $21)$.

So, the individual delay is bounded by $10 \cdot N^2 + 13 \cdot N + 2$ and is therefore of order $\mathcal{O}(N^2)$.

5. In conclusion

All assertions in this paper about the transition system of Sect. 3 have been proved with the proof assistant PVS [OSRSC01]. The starting point is a formal description in PVS of Fig. 2 in relational semantics. PVS helps primarily with exhaustive case distinctions and the administration of the proof obligations. How this can be done and our experiences with this proof assistant are described in [Hes13a]. The proof script for the present paper is available on [Hes15b].

The safety properties of the algorithm, mutual exclusion, absence of deadlock, FCFS, boundedness of the tickets, are all proved by means of invariants, as is usual. It is much work, but with experience and a powerful proof assistant it can be done.

The treatment of progress is more innovative. The numerical quantification in the Theorems 3 and 4 does not require much more effort than a standard UNITY proof for the corresponding progress assertions. The UNITY proof seems to be easier than a temporal logic proof such as given in [Hes13a], primarily because the UNITY concepts **e**nsures and *leads-to* are more intuitive than sets of executions can ever be.

The result is that the Black-White Bakery Algorithm has a throughput factor linear in N, and individual delay quadratic in N. This can also be proved for the ordinary Bakery Algorithm. It can be compared with the result of [Hes15a] for the algorithm of Lycklama–Hadzilacos–Aravind: there the throughput factor is quadratic in N and the individual delay is cubic in N. On the other hand, we conjecture that the tournament algorithm Peterson-Buhr of [BDH15, Section 18.6] has a throughput factor logarithmic in N and individual delay linear in N.

If one wants to implement the Black-White Bakery Algorithm for a fixed and modest number of threads, the active set `partic` can be removed from the algorithm of Fig. 1. This means removal of the lines 22 and 38, and replacing $getset(\mathtt{partic})$ by *thread* in the lines 24 and 29.

References

[AdBO09] Apt KR, de Boer FS, Olderog E-R (2009) Verification of sequential and concurrent programs. Springer, New York

[Ara13] Aravind A (2013) Simple, space-efficient, and fairness improved FCFS mutual exclusion algorithms. J Parallel Distrib Comput 73:1029–1038

[AST99] Afek Y, Stupp G, Touitou D (1999) Long-lived adaptive collect with applications. In: Proceedings 40th IEEE symp. on foundations of computer science, pp 262–272
[BDH15] Buhr PA, Dice D, Hesselink WH (2015) High-performance N-thread software solutions for mutual exclusion. Concurr Comput Pract Exp 27:651–701. doi:10.1002/cpe.3263
[CM88] Chandy KM, Misra J (1988) Parallel program design. A foundation. Addison–Wesley, Menlo Park
[Dij65] Dijkstra EW (1965) Solution of a problem in concurrent programming control. Commun ACM 8:569
[Hes98] Hesselink WH (1988) Progress under bounded fairness. Distrib Comput 12:197–207
[Hes13a] Hesselink WH (2013) Mechanical verification of Lamport's Bakery Algorithm. Sci Comput Program 78:1622–1638
[Hes13b] Hesselink WH (2013) Verifying a simplification of mutual exclusion by Lycklama–Hadzilacos. Acta Inf 50:297–329
[Hes15a] Hesselink WH (2015) Mutual exclusion by four shared bits with not more than quadratic complexity. Sci Comput Program 102:57–75. doi:10.1016/j.scico.2015.01.001
[Hes15b] Hesselink WH (2015) PVS proof scripts for four Bakery Algorithms. http://wimhesselink.nl/mechver/bakery/index.html. Accessed 17 March 2016
[Lam74] Lamport L (1974) A new solution of Dijkstra's concurrent programming problem. Commun ACM 17:453–455
[Lam86a] Lamport L (1986) The mutual exclusion problem—part I: a theory of interprocess communication, part II: statement and solutions. J ACM 33:313–348
[Lam86b] Lamport L (1986) On interprocess communication. Parts I and II. Distrib Comput 1:77–101
[LH91] Lycklama EA, Hadzilacos V. (1991) A first-come-first-served mutual-exclusion algorithm with small communication variables. ACM Trans Program Lang Syst 13:558–576
[Mis01] Misra J (2001) A discipline of multiprogramming: programming theory for distributed applications. Springer, New York
[NLWSD14] Nanevski A, Ley-Wild R, Sergey I, Delbianco GA (2014) Communicating state transition systems for fine-grained concurrent resources. In: Shao Z (ed) ESOP 2014. LNCS, vol 8410, pp 290–310
[OG76] Owicki S, Gries D (1976) An axiomatic proof technique for parallel programs. Acta Inf 6:319–340
[OSRSC01] Owre S, Shankar N, Rushby JM, Stringer-Calvert DWJ (2001) PVS version 2.4, system guide, prover guide, PVS language reference. http://pvs.csl.sri.com. Accessed 17 March 2016
[Tau04] Taubenfeld G (2004) The Black-White Bakery Algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. In: Proceedings of the DISC. LNCS, vol 3274, pp 56–70

Received 25 September 2015
Accepted in revised form 31 January 2016 by Xinyu Feng
Published online 29 March 2016