A partial barrier based on Trylock

Wim H. Hesselink, whh600

September 22, 2023

Abstract

A partial barrier for size M is a synchronization mechanism for thread systems that allows threads repeatedly to pass an *entry* point only in batches of precisely M elements. The paper presents the design of a partial barrier, based on trylock. The safety of the design is proved by means of invariants and a refinement function towards a partial barrier prototype. Progress is proved with bounded Unity. In this way, the proof of progress gives at the same time an estimate of the concurrent complexity.

keywords: synchronization, barrier, trylock, concurrent complexity, Unity

1 Introduction

In a system of N threads, a *barrier* is a synchronization mechanism to let the threads repeatedly pass through a barrier point. When a thread arrives at this point, it has to wait for all threads to arrive. The threads can pass the point only when all have arrived.

A *partial barrier* is a variation of this idea. A partial barrier for size M is a synchronization mechanism for thread systems that allows threads repeatedly to pass an *entry* point only in batches of precisely M elements. The members of the batch can subsequently *release* membership. When all members have released, and enough new participants are available, a new batch can be formed and launched.

Just as with mutual exclusion, in a system with a partial barrier, every thread is in an unbounded loop of the form

loop of thread p ;
    NCS ; *entry* ; *CS* ; *release*
end of loop .

Here *CS* is the critical section that the threads enter only in batches with M members, and *NCS* is the noncritical section. *CS* and *NCS* are program fragments that do not concern us except for the question which threads are currently executing them.

In general, a partial barrier is not a barrier. It can serve as a barrier if M is the total number of threads. At the other extreme, the case of $M = 1$ is mutual exclusion, compare [2] and the references given there. Indeed, the partial barrier looks like a special case of group mutual exclusion, see [1] and its references.

What are the requirements for partial barriers? It is clearly required that the number of threads in *CS* must be $\leq M$. This, however, is not enough, because it would allow batches with fewer than M members to be launched, and it would allow new threads to enter *CS* when some members of the previous batch have been released, and others not. We therefore present in Section 2 a partial barrier prototype as a formal specification.

The aim of this paper is to construct a partial barrier for size M, based on the synchronization primitive *trylock*. The implementation proposed is developed in Section 3. The safety of the design, i.e., the functional correctness, is proved in Section 4. Progress is verified with bounded Unity, in Section 5. Conclusions are drawn in Section 6.

2 Formal Specification

The easiest way to specify a partial barrier is to require that its lifetime loop implements (i.e., simulates) the following prototype, where W is the set of waiting threads and B is the batch.

shared variables :
  W, B : **set of thread** ;
initially: $W = B = \emptyset$.

loop of thread p :
1 *NCS* ;
2 **add** p **to** W ;
3 **await** $p \in B$;
4 *CS* ;
5 **remove** p **from** B
end loop .

The lines 2 and 3 represent *entry*, and line 5 represents *release*. Repeatedly, the sets W and B are modified by an auxiliary thread, the *butler*:

loop of thread *butler* :
  $\langle$ **if** $B = \emptyset \ \wedge \ \#W \geq M$ **then**
      choose $B \subseteq W$ **with** $\#B = M$;
      $W := W \setminus B$;
    endif $\rangle$
end loop .

The brackets $\langle\ \rangle$ enclose a single atomic transition. For a finite set S, the expression $\#S$ denotes its cardinality, its number of elements. So, the *butler* can act only if B is empty and at least M threads are waiting, it then moves M threads from W to B. The aim, however, is not to dedicate a special thread to the role of butler, but to let this role be played by one of the waiting threads.

The progress requirement of the prototype is weak fairness: if one of its steps is continually enabled, then it is taken eventually. This holds in particular for the butler step and for step 3. The butler step is enabled iff $B = \emptyset \ \wedge \ \#W \geq M$. When this holds, it remains valid until (unless) a batch B is chosen with $\#B = M$. Therefore, in every implementation, the condition $B = \emptyset \ \wedge \ \#W \geq M$ must lead to $\#B = M$. Similarly, weak fairness of step 3 means that, in every implementation, the condition $p \in B$ must lead to a state where p is in line 4 or 5.

3 Implementation using *trylock*

The partial barrier to be constructed is based on trylock, a synchronization primitive that consists of two functions: *trylock* and *unlock*. A call of *trylock* returns a boolean that indicates whether the calling thread has obtained the lock. A successful thread can release the lock by calling *unlock*. The primitive can be offered by the hardware or the operating system as a compare-and-swap. It can also be implemented with atomic registers, e.g., see [7].

The partial barrier is constructed as follows. Assume that the system has N threads, numbered 0, ..., $N-1$. The partial barrier uses a shared array `aa` of N integers, initially all 0. If `aa`$[p] = 2$, thread p has access to the critical section. If `aa`$[p] = 1$, thread p is waiting to obtain access. Otherwise, thread p is not interested and `aa`$[p] = 0$. A boolean `flag`, initially *true*, is used to to avoid that *trylock* is flooded unnecessarily.

The function *entry* uses the local variable k to hold a thread, and *bar* to hold a set of threads. Addition for thread numbers i, $j < N$ is defined by $i \oplus j = (i + j)$ **mod** N. The function *entry* and *release* are given by

```
    entry(thread p) :
12      aa[p] := 1 ;
13      while ¬ trylock(p) do
14          await (aa[p] = 2 ∨ flag) ;
15          if aa[p] = 2 then return endif
        endwhile ;
16      if aa[p] = 2 then
17          unlock(p) ; return endif ;
18      flag := false ; bar := singleton(p) ; k := p ⊕ 1 ;
19      while #(bar) < M do
            if aa[k] = 1 then add k to bar endif ;
            k := k ⊕ 1
        endwhile ;
20      for each k do await(aa[k] ≠ 2) endfor ;
22      for each k ∈ bar do aa[k] := 2 endfor ;
23      flag := true ;
24      unlock(p)
    end entry .

    release(thread p) :
26      aa[p] := 0
    end release .
```

For each thread q, the value of `aa`$[q]$ cycles from 0 to 1 (line 12), from 1 to 2 (line 22), from 2 to 0 (line 26). In line 13, one of the waiting threads gets the lock to select a new batch of M waiting threads. If `mu` $= \perp$, then `flag` holds and some waiting threads can try to get the lock.

At most one thread can get the lock, let us call it the "butler". It selects the new batch in the set *bar*, in loop 19. Here thread p cycles through the threads to find enough threads k with `aa`$[k] = 1$. Note that *bar* is a set; addition of an element has no effect if it is already in the set.

The new batch can only be launched when all members of the previous batch have been released; this is verified by the await statements of loop 20. The loops at 19 and 20 can be done in arbitrary order, and can even be done in parallel. The verification below treats the order presented here. While thread p is in line 19 or 20, the lock prohibits execution of line 22, and thus all transitions of `aa`$[q]$ from 1 to 2, for any thread q. It follows that the predicates inspected by thread p in the loops 19 and 20 are stable until p itself goes to line 22.

In line 22, thread p notifies the selected threads k by setting `aa`$[k] := 2$. At the moment of its notification, a selected thread can be at various locations of the function *entry*. This is the reason for the two internal return statements, in the lines 15 and 17.

When the butler has performed its task, it leaves by unlocking the lock, and enters CS with the batch it has selected. If at this point, there are threads waiting, a new butler must be appointed. For this reason the busy-waiting loop 13-15 includes repeated calls of *trylock*. On the other hand, any thread q with `aa`$[q] = 2$ must leave the call of *entry* without waiting and therefore must not be selected as butler.

4 Verification of safety

In Section 4.1, the calls *entry* and *release* of Section 3 are placed in a lifeline loop to match the prototype of Section 2. Section 4.2 collects and proves the invariants needed for safety. Section 4.3 presents and proves the refinement function to the prototype. In Section 4.4, some additional invariants are obtained that are needed for the proofs of progress in Section 5.

4.1 The transition system

The primitive *trylock* is modeled by means of a shared variable `mu`, which is initially equal to $\perp$. If `mu` $\neq \perp$, the value of `mu` is the thread that holds the lock. The call of *trylock*(p) is equivalent to atomic

statement

$$\langle \textbf{ if } \texttt{mu} = \bot \textbf{ then } \texttt{mu} := p \,;\ \textbf{return } \textit{true} \textbf{ else return } \textit{false} \textbf{ endif } \rangle \ .$$

The complement $\textit{unlock}(p)$ is equivalent to $\texttt{mu} := \bot$, but must have precondition $\texttt{mu} = p$.

For the verification of safety, the pair *entry*, *release* is included in the transition system of the lifeline loop below.

```
loop of thread p :
11    NCS ;
12    aa[p] := 1 ;
13    if mu ≠ ⊥ then
14       await aa[p] = 2 ∨ flag ;
15       if aa[p] = 2 then goto 25 else goto 13 endif ;
      else mu := p ;
16       if aa[p] = 2 then
17          mu := ⊥ ; goto 25
         endif ;
18       flag := false ; bar_p := singleton(p) ; kk_p := p ⊕ 1 ;
19       while #(bar_p) < M do
            if aa[kk_p] = 1 then add kk_p to bar_p endif ;
            kk_p := kk_p ⊕ 1
         endwhile ;
         lis_p := allthreads ;
20       while exists kk_p ∈ lis_p do
21          await aa[kk_p] ≠ 2 ; remove kk_p from lis_p
         endwhile ;
22       while exists k ∈ bar_p do
            aa[k] := 2 ; remove k from bar_p
         endwhile ;
23       flag := true ;
24       mu := ⊥
      endif ;
25    CS ;
26    aa[p] := 0
end loop .
```

In loop 19, the local variable kk_p indicates the next thread to investigate.

For the loops 20 and 22, the order of treatment is not important. In loop 20, the local variable lis_p holds the set of the threads k for which thread p has yet to test if $\texttt{aa}[k] \neq 2$. In line 21, the variable kk_p indicates where thread p is waiting. The variable kk_p are not needed in loop 22, because the treatment of k can be included in a single atomic command.

The initial condition is

$$\texttt{mu} = \bot \ \wedge\ \texttt{flag} \ \wedge\ (\forall\, q : \texttt{aa}[q] = 0 \ \wedge\ pc_q = 11) \ .$$

Here, pc_q is the program counter, that indicates the next line to be executed by thread q.

4.2 Invariants for the transition system

The notation $[i, k]$ is used to denote the set of threads q with $i \leq pc_q \leq k$. Similarly, $[\ell]$ is the set of the threads q with $pc_q = \ell$. The first invariant to mention implies that there is at most one thread that holds the lock, expressed in

Iq1 : $\quad q \in [16, 24] \ \Rightarrow\ \texttt{mu} = q \ .$

This predicate is inductive.

The second point is that $\mathtt{aa}[q] = 2$ holds if thread q is in CS or approaching or leaving it:

$$Iq2: \quad q \in [23, 26] \;\Rightarrow\; \mathtt{aa}[q] = 2 \;.$$

This predicate is threatened only by the steps 17 and 22. It has the respective remedies

$$Iq3: \quad q \in [17] \;\Rightarrow\; \mathtt{aa}[q] = 2 \;,$$
$$Iq4: \quad q \in [19, 22] \;\Rightarrow\; q \in bar_q \;\vee\; \mathtt{aa}[q] = 2 \;.$$

The predicates *Iq3* and *Iq4* are inductive.

As a new batch must not be launched before all threads of the previous batch have been released, one needs command 20 with the invariant

$$Iq5: \quad q \in [20, 21] \;\wedge\; \mathtt{aa}[r] = 2 \;\Rightarrow\; r \in lis_q \;.$$

Predicate *Iq5* is threatened only by step 22. It has the remedy *Iq1*.

The conditional command 16-17 serves to ensure the invariant

$$Jq1: \quad q \in [18, 21] \;\Rightarrow\; \mathtt{aa}[q] = 1 \;.$$

Predicate *Jq1* is threatened only by the steps 16 and 22. At step 22, it has the remedy *Iq1*. At step 16, it has the remedy

$$Jq2: \quad q \in [13, 26] \;\Rightarrow\; \mathtt{aa}[q] = 1 \;\vee\; \mathtt{aa}[q] = 2 \;.$$

Predicate *Jq2* is inductive. Threads in $[11, 12]$ must not be included in a new batch. Therefore, one postulates

$$Jq3: \quad q \in [11, 12] \;\Rightarrow\; \mathtt{aa}[q] = 0 \;.$$

Predicate *Jq3* is threatened only by step 22. It has the remedy

$$Jq4: \quad q \in [19, 22] \;\wedge\; r \in bar_q \;\Rightarrow\; \mathtt{aa}[r] = 1 \;.$$

Predicate *Jq4* is threatened only by the steps 18, 22, and 26. It has the respective remedies *Jq1*, *Iq1*, *Iq2*.

The loop at line 19 has the obvious postcondition

$$Jq5: \quad q \in [20, 21] \;\Rightarrow\; \#(bar_q) = M.$$

Indeed, this predicate is threatened only by step 19. Its remedy is the inductive invariant

$$Jq6: \quad q \in [19, 24] \;\Rightarrow\; \#(bar_q) \leq M.$$

As a preparation of the refinement function, the disjoint sets *Waiting* and *Batch* are defined by $Waiting = A1 \setminus bar22$ and $Batch = A2 \cup bar22$ where $A1 = \{q \mid \mathtt{aa}[q] = 1\}$ and $A2 = \{q \mid \mathtt{aa}[q] = 2\}$ and

$$bar22 = \{q \mid \exists\, p : p \in [22] \;\wedge\; q \in bar_p\} \;.$$

The reason for introducing *bar22* is that step 20E from line 20 to line 22 must fill *Batch* with M elements in a single step.

It follows from *Iq5* and *Iq1* that

$$Iq5A: \quad q \in [20] \;\wedge\; lis_q = \emptyset \;\Rightarrow\; Batch = \emptyset \;.$$

It follows from *Jq4* and *Iq1* that

$$Jq4A: \quad q \in [20] \;\Rightarrow\; bar_q \subseteq Waiting \;.$$

4.3 Refinement from implementation to prototype

The state space X of the implementation is spanned by the shared variables `mu` and `aa`$[N]$, and the private variables lis_p, bar_p, kk_p, pc_p, for all threads p. Similarly, the prototype of Section 2 has a state space X_0 spanned by the shared variables `W` and `B`, and the program counters pc_p for all threads p. Note that the butler thread of the prototype does not need a program counter because it has only one atomic command, which can be executed whenever its precondition holds.

There are many ways to show simulation relations between concurrent algorithms, e.g., for an overview see [4]. The simplest way is by constructing a refinement function. This is sufficient for the present purposes. It is a function f, from the state space X of the implementation to the state space X_0 of the prototype, that maps the initial states of X to the initial states of X_0, and that maps every step of the implementation to a step of the prototype. This means that for every pair of states (x, y) in X, for which there is a thread p that can change x into y, either $f(x) = f(y)$ or there is a step from $f(x)$ to $f(y)$ in X_0 by some thread.

The refinement function proposed is defined as follows. First note that the invariants of Section 4.2 implicitly depend on the state $x \in X$. The program counters pc and the sets *Batch* and *Waiting* also depend on state x. This is made explicit by writing $x.pc_q$ and $Batch(x)$ and $Waiting(x)$. The refinement function $f : X \to X_0$ is defined by

$$f(x) = (\ \mathtt{W} := Waiting(x), \mathtt{B} := Batch(x), pc := (\lambda\, q : g(x.pc_q))\ )\ , \text{ where}$$
$$g(\ell) = (\ell = 11\ ?\ 1 :\ \ell = 12\ ?\ 2\ :\ \ell = 25\ ?\ 4\ :\ \ell = 26\ ?\ 5\ :3)\ .$$

In particular, the locations of *NCS* (11 and 1), of *CS* (25 and 4), and of *release* (26 and 5) are made to correspond.

In order to prove that this function f is a refinement function, first note that indeed the initial states of the implementation are mapped to initial states of the prototype, and that the observables *NCS* and *CS* correspond. The main point, however, is to verify that the steps of the implementation map to steps of the prototype.

The most drastic step of the implementation is step 20E, from line 20 to line 22. In line 20, the set *Batch* is empty by *Iq5A*, it becomes equal to bar_p. In the precondition, bar_p is a subset of *Waiting* because of *Jq4A*, it has M elements because of *Jq5*. Also using *Iq1*, it follows that step 20E of the implementation is mapped to the *butler* step of the prototype.

Step 11 of thread p in the implementation is mapped to step 1 of p in the prototype. Step 12 is mapped to step 2 of the prototype, here one needs the invariants *Jq3* and *Jq4*. The steps 13 and 14 of thread p are mapped to the identity step of the prototype. Step 15 of thread p is mapped to step 3 of the prototype if `aa`$[p] = 2$. Otherwise it is mapped to the identity step. Steps 16 and 17 are mapped to the identity step. Step 18 is mapped to step 3 of the prototype, because of the invariants *Iq1* and *Iq3*. The steps of line 19, and the steps from 20 to 21, and from 21 to 20 are mapped to the identity step of the prototype, as are the steps from the lines 22 and 23, because of *Iq1*. Step 24 is mapped to step 3 of the prototype because of *Iq1* and *Iq2*. Step 25 is mapped to step 4. Step 26 is mapped to step 5, because of *Iq2* and *Jq4*. This concludes the proof that the implementation simulates the prototype.

4.4 Invariants for progress

For the sake of progress, we need some additional invariants. This first one is:

Kq1 : $\quad \mathtt{mu} = \bot\ \Rightarrow\ \mathtt{flag}$.

This predicate is threatened only by the steps 17, 18, 24. At step 18, *Iq1* is a remedy. At the steps 17 and 24, it has the remedies

Kq2 : $\quad q \in [16, 17]\ \Rightarrow\ \mathtt{flag}$,
Kq3 : $\quad q \in [24]\ \Rightarrow\ \mathtt{flag}$.

Predicate *Kq2* is threatened only by the steps 13 and 18. It has the remedies *Kq1* and *Iq1*. Predicate *Kq3* is threatened only by step 18. It has the remedy *Iq1*.

We also need the inductive invariants:

$Kq4:$ $\quad \mathtt{mu} = \bot \ \vee \ \mathtt{mu} \in [16, 24]$,
$Kq5:$ $\quad q \in [21] \ \Rightarrow \ kk_q \in lis_q$.

5 Progress with UNITY

The progress properties of the algorithm are expressed and proved by means of bounded UNITY. This is a variation of UNITY of Chandy and Misra [3, 8], which was proposed in [5, 6]. The central concept of UNITY is "leads to". Predicate P leads to predicate Q iff every computation that starts in a state where P holds reaches a state where Q holds. Bounded UNITY quantifies this by introducing "rounds". Predicate P leads to Q within n rounds iff every computation that starts in P and contains at least n rounds reaches a state where Q holds.

Section 5.1 contains an operational introduction. The theory of UNITY and bounded UNITY is presented in Section 5.2. Section 5.3 gives the two leads-to relations for the partial barrier, which are then treated in the Sections 5.4 and 5.5, respectively.

5.1 Operational introduction

The state of the system is given by the values of all shared and private variables. Usually, one prefers to keep the state implicit, but formally all invariants are boolean functions of the state. Just as in Section 4.3, 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. An inclusion $P \subseteq Q$ therefore means that every state that satisfies P also satisfies Q (i.e. that P implies Q). Let *init* 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 as an infinite sequence $s = (s_0, \ldots)$ with $s_0 \in init$ and $(s_i, s_{i+1}) \in step$ for all $i \geq 0$. A predicate P is an *invariant* iff it contains all executions. Let $Inv \subseteq X$ be the intersection of all invariants obtained for the algorithm, in the sections 4.2 and 4.4.

A step of thread p is called a *forward step* iff thread p does not start in line 11. This is because steps from line 11 are triggered by the environment, and not by the algorithm. One thus defines

$$(x, y) \in fwd(p) \quad \equiv \quad x.pc_p \neq 11 \ \wedge \ (x, y) \in step(p) \ .$$

Thread p is said to be enabled in state x iff there is a state y with $(x, y) \in fwf(p)$. This is expressed by the predicate $ena(p)$. For the transition system of Section 4, one obtains

$$\begin{aligned} ena(p) \quad \equiv \quad & p \in [12, 26] \ \wedge \ (p \in 14 \ \Rightarrow \ \mathtt{aa}[p] = 2 \ \vee \ \mathtt{flag}) \\ & \wedge \ (p \in [21] \ \Rightarrow \ \mathtt{aa}[kk_p] \neq 2) \ . \end{aligned}$$

Let a *run* of length $n \geq 0$ be a nonempty finite sequence $s = (s_0 \ldots s_n)$ in Inv such that $(s_i, s_{i+1}) \in step$ whenever $0 \leq i < n$. Two runs can be concatenated when the final state of the first fragment equals the initial state of the second fragment.

An *occurrence* of thread p in a run $(s_0 \ldots s_n)$ is a number i with $0 \leq i < n$, and $(s_i, s_{i+1}) \in fwd(p)$ or $s_i \notin ena(p)$. The run $(s_0 \ldots s_n)$ is called a *round* iff it contains an occurrence of every thread. In other words, in the run, every thread is scheduled at least once, and is either executed or found to be disabled.

Progress of the algorithm will be proved under the assumption that all threads do enough forward steps unless they are disabled. More precisely, progress will be proved for any run that is a concatenation of sufficiently many rounds. This is done in bounded UNITY by assertional means, not by investigating runs.

5.2 UNITY and bounded UNITY

UNITY logic [3, 8] is a method of assertional reasoning to 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 $t' \geq t$.

The starting point consists of the relations **co** and **co!** between predicates P and Q:

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

P **co** Q says that every step that starts in P ends in Q. The operator **co!** indicates that a particular thread r is *responsible* for reaching Q. The operators **co** and **co!** are used to define the operators **unless** and **ensures** in

$$P \textbf{ unless } Q \quad \equiv \quad (P \wedge \neg Q \wedge \mathit{Inv}) \textbf{ co } (P \vee Q) \;,$$
$$P \textbf{ ensures } Q \quad \equiv \quad (P \textbf{ unless } Q) \wedge ((P \wedge \neg Q \wedge \mathit{Inv}) \textbf{ co! } Q) \;.$$

The *leads-to* relation of UNITY is defined inductively by the three rules

(a) P **ensures** Q implies $P \mapsto Q$.
(b) Relation $\mapsto$ is transitive.
(c) If a family $(P_i)_{i\in I}$ has $P_i \mapsto Q$ for all $i \in I$, then $(\exists\, i \in I : P_i) \mapsto Q$.

Bounded UNITY [5, 6] extends the leads-to relation of UNITY with the number of rounds needed. We write P $\mathbf{Lt}\langle n\rangle$ Q to denote that every run that starts in a state where P holds and that contains a concatenation of n rounds, contains a state where Q holds. The main proof rules are

(a1) If $P \wedge \mathit{Inv} \subseteq Q$ then P $\mathbf{Lt}\langle n\rangle$ Q for every $n \geq 0$.
(a2) P **ensures** Q implies P $\mathbf{Lt}\langle 1\rangle$ Q.
(b) If P $\mathbf{Lt}\langle k\rangle$ Q and Q $\mathbf{Lt}\langle m\rangle$ R, then P $\mathbf{Lt}\langle k+m\rangle$ R.
(c) If a family $(P_i)_{i\in I}$ has P_i $\mathbf{Lt}\langle n\rangle$ Q for all $i \in I$, then $(\exists\, i \in I : P_i)$ $\mathbf{Lt}\langle n\rangle$ Q.

Rule (a2) is called the **ensures** rule, rule (c) the disjunction rule. The validity of these rules in the operational semantics is verified with the proof assistant PVS [9]. The same hasen done with the so-called PSP rule [3, 8], which in bounded UNITY [5, 6] has the form

PSP-rule Assume that P $\mathbf{Lt}\langle n\rangle$ Q and R **unless** B. Then $(P \wedge R)$ $\mathbf{Lt}\langle n\rangle$ $((Q \wedge R) \vee B)$.

The main rules are used to infer the following derived rules:

(b1) If P $\mathbf{Lt}\langle k\rangle$ Q and $k \leq m$ then P $\mathbf{Lt}\langle m\rangle$ Q.
(b2) If P $\mathbf{Lt}\langle k\rangle$ Q and $P' \wedge \mathit{Inv} \subseteq P$ then P' $\mathbf{Lt}\langle k\rangle$ Q.
(b3) If P $\mathbf{Lt}\langle k\rangle$ Q and $Q \wedge \mathit{Inv} \subseteq Q'$ then P $\mathbf{Lt}\langle k\rangle$ Q'.
(b4) If P $\mathbf{Lt}\langle k\rangle$ $(Q \vee R)$ and Q $\mathbf{Lt}\langle m\rangle$ R then P $\mathbf{Lt}\langle k+m\rangle$ R.

In many cases, a proof for UNITY can easily be tranformed into a proof for bounded UNITY by adding the administration of the numbers of rounds needed. The resulting number of rounds can be regarded as an estimate of the *concurrent complexity* of the algorithm.

5.3 Bounded UNITY for the algorithm

As discussed in Section 2, the liveness condition required is that every step of the prototype that is continually enabled, will eventually be taken. The steps 1, 2, 4, 5 of the prototype correspond to single steps of the implementation (11, 12, 25, 26, respectively). It remains to treat two steps: the butler step and step 3 of the prototype. When these steps are enabled, they should eventually be taken. Transferred to the implementation, they correspond to the leads-to relations

$$L1: \quad \mathit{Batch} = \emptyset \wedge \#\mathit{Waiting} \geq M \quad \mapsto \quad \#\mathit{Batch} = M \;,$$
$$L2: \quad p \in \mathit{Batch} \quad \mapsto \quad p \in \mathit{Final} \;.$$

Here *Final* is the set of location of *CS* combined with *release*. Therefore $\mathit{Final} = [25, 26]$.

In the next sections, quantified versions of these relations are proved.

5.4 L1: eventually a batch is launched

In order to prove the leads-to relation *L1*, let its precondition and its postcondition be denoted *PreB* and *PostB*, respectively. We thus have $PreB = (Batch = \emptyset \wedge \#Waiting \geq M)$ and $PostB = (\#Batch = M)$. The first conjunct of *PreB* satisfies:

(0) $\quad Batch = \emptyset \quad \equiv \quad (A2 = \emptyset) \wedge (\mathtt{mu} = \bot \vee \mathtt{mu} \in [16, 21])$.

In fact, the implication from right to left follows from the definition of *Batch*, the implication from left to right uses *Jq2*, *Jq4*, and *Kq4*. It follows that *PreB* can only be falsified by step 20E (from line 20 to 22). Now using *Jq5* and *Iq1*, one obtains as a first step towards *L1*:

(1) $\quad PreB \quad \textbf{unless} \quad PostB$.

Step 20B can only be executed if there is a thread that holds the lock and is at line 20. The lock can only be obtained at line 13. Threads interested in the lock circle through the lines 13, 14, 15. We therefore prove that

$$
\begin{array}{l}
p \in [13] \quad \textbf{ensures} \quad \mathtt{mu} \neq \bot \ , \\
p \in [14] \quad \textbf{ensures} \quad p \in [15] \vee \mathtt{mu} \neq \bot \ , \\
p \in [15] \wedge \mathtt{aa}[p] \neq 2 \quad \textbf{ensures} \quad p \in [13] \vee \mathtt{mu} \neq \bot \ .
\end{array}
$$

Note that there is no suggestion that thread p gets the lock. The step from line 13 is easy. The step from line 14 uses the invariant *Kq1*. The step from line 15 uses *Iq1* to avoid interference with step 22.

In these three propositions the **ensures** relation is replaced by $\mathbf{Lt}\langle 1\rangle$. Subsequently, the PSP-rule is applied to them with the **unless** relation of (1). Putting $Mubb = ((\mathtt{mu} \neq \bot \wedge PreB) \vee PostB)$, this gives

$$
\begin{array}{l}
p \in [13] \wedge PreB \quad \mathbf{Lt}\langle 1\rangle \quad Mubb \ , \\
p \in [14] \wedge PreB \quad \mathbf{Lt}\langle 1\rangle \quad (p \in [15] \wedge PreB) \vee Mubb \ , \\
p \in [15] \wedge PreB \quad \mathbf{Lt}\langle 1\rangle \quad (p \in [13] \wedge PreB) \vee Mubb \ .
\end{array}
$$

The UNITY rules now enable us to infer

$$p \in [13, 15] \wedge PreB \quad \mathbf{Lt}\langle 3\rangle \quad Mubb \ .$$

Predicate *PreB* implies the existence of threads p with $\mathtt{aa}[p] = 1$. If $\mathtt{mu} = \bot$, such threads are necessarily in $[13, 15]$. In this way, one arrives at

(2) $\quad \mathtt{mu} = \bot \wedge PreB \quad \mathbf{Lt}\langle 3\rangle \quad Mubb$.

We turn to the proof that the first disjunct of *Mubb* leads to the second disjunct. For this purpose, we first prove that

$$p \in [16] \wedge \mathtt{aa}[p] \neq 2 \quad \textbf{ensures} \quad p \in [18] \ .$$

Again, the **ensures** rule is applied, followed by the PSP-rule with the **unless** relation of (1). This gives

(3) $\quad p \in [16] \wedge PreB \quad \mathbf{Lt}\langle 1\rangle \quad (p \in [18] \wedge PreB) \vee PostB$.

We then prove $p \in [18]$ **ensures** $p \in [19]$. Again the PSP-rule is applied, which gives

(4) $\quad p \in [18] \wedge PreB \quad \mathbf{Lt}\langle 1\rangle \quad (p \in [19] \wedge PreB) \vee PostB$.

In loop 19, the conjunct $\#Waiting \geq M$ of *PreB* is to be used. This conjunct implies $\#A1 \geq M$. When this condition becomes true, thread p can be anywhere in loop 19, i.e., $kk_p = q$ for some thread q. After this, thread p proceeds in its loop, and kk_p moves along the circle. If $j \leq N$, then after j steps, thread p has covered the set of threads

$$S(q, j) = \{r \mid q \leq r < q + j \ \vee \ r < q + j - N\} \ .$$

By induction, one proves for any threads p, q, any set of threads W, and any natural number $j \leq N$, the leads-to relation

$$(p \in [19] \wedge kk_p = q \wedge W \subseteq A1) \quad \mathbf{Lt}\langle j \rangle$$
$$(p \in [19] \wedge kk_p = q \oplus j \wedge W \subseteq A1 \wedge (W \cap S(q,j) \subseteq bar_p)) \ \vee \ p \in [20] \ .$$

If $j = N$ then $S(q,j)$ is the set of all threads. Therefore, if W has at least M elements, bar_p has at least M elements. In this way, one arrives at

$$(p \in [19] \wedge kk_p = q \wedge W \subseteq A1 \wedge M \leq \#W) \quad \mathbf{Lt}\langle N \rangle$$
$$(p \in [19] \wedge M \leq \#bar_p) \ \vee \ p \in [20] \ .$$

The disjunction rule is used to remove the free variables q and W, and to obtain

$$(p \in [19] \wedge M \leq \#A1) \quad \mathbf{Lt}\langle N \rangle \quad (p \in [19] \wedge M \leq \#bar_p) \ \vee \ p \in [20] \ .$$

It is easy to prove that $(p \in [19] \wedge M \leq \#bar_p)$ **ensures** $p \in [20]$. Therefore, transitivity gives

$$(p \in [19] \wedge M \leq \#A1) \quad \mathbf{Lt}\langle N+1 \rangle \quad p \in [20] \ .$$

Finally, the PSP-rule with (1) gives

(5) $$p \in [19] \wedge PreB \quad \mathbf{Lt}\langle N+1 \rangle \quad (p \in [20] \wedge PreB) \vee PostB \ .$$

For the loop of the lines 20, 21, we introduce

$$C(p, \ell, i) = (p \in [\ell] \wedge A2 = \emptyset \wedge \#lis_p = i) \ .$$

Using *Kq5*, one proves that these predicates satisfy the **ensures** relations

$$C(p, 20, i+1) \ \textbf{ensures} \ C(p, 21, i+1) \ \textbf{ensures} \ C(p, 20, i) \ ,$$
$$C(p, 20, 0) \ \textbf{ensures} \ p \in [22] \ .$$

From this, one can infer

$$p \in [20, 21] \wedge A2 = \emptyset \quad \mathbf{Lt}\langle 2 \cdot N + 1 \rangle \quad p \in [22] \ .$$

The PSP-rule with the **unless** relation of (1) together with formula (0) gives

(6) $$p \in [20, 21] \wedge PreB \quad \mathbf{Lt}\langle 2 \cdot N + 1 \rangle \quad PostB \ .$$

Now rule (b4) is applied on formula (6) with (5), (4), (3), respectively. This gives

$$p \in [19] \wedge PreB \quad \mathbf{Lt}\langle 3 \cdot N + 2 \rangle \quad PostB \ ,$$
$$p \in [18] \wedge PreB \quad \mathbf{Lt}\langle 3 \cdot N + 3 \rangle \quad PostB \ ,$$
$$p \in [16] \wedge PreB \quad \mathbf{Lt}\langle 3 \cdot N + 4 \rangle \quad PostB \ .$$

The disjunction rule with formula (0) and the invariant *Iq3* gives

$$\texttt{mu} \neq \bot \wedge PreB \quad \mathbf{Lt}\langle 3 \cdot N + 4 \rangle \quad PostB \ .$$

Combined with formula (2), this results in

$$L1: \quad PreB \quad \mathbf{Lt}\langle 3 \cdot N + 7 \rangle \quad PostB \ .$$

5.5 L2: From Batch to Final

The proof of leads-to relation *L2* begins with the treatment of loop 22. One first proves that

$$p \in [22] \wedge \#bar_p = i + 1 \ \textbf{ensures} \ p \in [22] \wedge \#bar_p = i \ ,$$
$$p \in [22] \wedge bar_p = \emptyset \ \textbf{ensures} \ p \in [23] \ .$$

Using induction and the invariant *Jq6*, one then gets

$$p \in [22] \quad \mathbf{Lt}\langle M + 1 \rangle \quad p \in [23] \ .$$

Via the disjunction rule this gives $\mathit{Mu22}\ \mathbf{Lt}\langle M+1\rangle\ \neg\mathit{Mu22}$, where $\mathit{Mu22} = (\mathtt{mu} \neq \bot \wedge \mathtt{mu} \in [22])$. On the other hand, it holds that $(p \in \mathit{Batch})$ **unless** $(p \in \mathit{Final})$. The PSP-rule therefore implies

$$\mathit{Mu22} \wedge p \in \mathit{Batch} \quad \mathbf{Lt}\langle M+1\rangle \quad (\neg\,\mathit{Mu22} \wedge p \in \mathit{Batch}) \vee p \in \mathit{Final}\ .$$

The conjunction $\neg\,\mathit{Mu22} \wedge p \in \mathit{Batch}$ implies

$$(\mathtt{aa}[p] = 2 \wedge p \in [13,17]) \vee p \in [23,26]\ .$$

Therefore, rule (b3) can be used to get

(7) $$\mathit{Mu22} \wedge p \in \mathit{Batch} \quad \mathbf{Lt}\langle M+1\rangle \quad (\mathtt{aa}[p] = 2 \wedge p \in [13,17]) \vee p \in [23,26]\ .$$

For the region $[13,17]$, one can prove the **ensures** relations

$\mathtt{aa}[p] = 2 \wedge p \in [13]$ **ensures** $\mathtt{aa}[p] = 2 \wedge (p \in [14] \vee p \in [16])$,
$\mathtt{aa}[p] = 2 \wedge p \in [\ell]$ **ensures** $\mathtt{aa}[p] = 2 \wedge p \in [\ell+1]$ for $\ell = 14$ or 16,
$\mathtt{aa}[p] = 2 \wedge p \in [\ell]$ **ensures** $p \in \mathit{Final}$ for $\ell = 15$ or 17.

This is easily combined to

$$\mathtt{aa}[p] = 2 \wedge p \in [13,17] \quad \mathbf{Lt}\langle 3\rangle \quad p \in \mathit{Final}\ .$$

On the other hand, it is easy to prove that $(p \in [23,26])\ \mathbf{Lt}\langle 2\rangle\ (p \in \mathit{Final})$. Together this gives

$$(\mathtt{aa}[p] = 2 \wedge p \in [13,17]) \vee p \in [23,26] \quad \mathbf{Lt}\langle 3\rangle \quad p \in \mathit{Final}\ .$$

Combined with formula (7), this results in

L2 : $$p \in \mathit{Batch} \quad \mathbf{Lt}\langle M+4\rangle \quad p \in \mathit{Final}\ .$$

6 In conclusion

It was an important step to specify the partial barrier by means of an abstract prototype. The simplicity of the design is illustrated by the simplicity of the invariants. The main complication for the design was in the lines 13-17 where a butler p with $\mathtt{aa}[p] = 1$ must be chosen. The proof of progress in Section 5 is a nice illustration of various UNITY techniques.

The algorithm remains correct if the assignment `flag` := *false* (in line 18) is removed. Then `flag` is always *true*, and line 14 can also be removed. The reason to choose the present version is that it seems to lower the probability that the lock is taken by a thread p with $\mathtt{aa}[p] = 2$, which seems to be bad for performance. *Is this confirmed by experiments?*

The design of the algorithm is quite flexible. It can easily be modified in various ways. For example, before launching a new batch, the butler can give the threads of the new batch specific tasks to execute during the critical section. The butler can choose the size of the next batch to be formed and launched. One can give the threads weights, and replace the requirement of precisely M threads per batch by a condition on the total weight of a new batch. One can remove the requirement that a new batch is not launched before all threads of the previous batch have released.

References

[1] A. A. Aravind and W. H. Hesselink. Group mutual exclusion by fetch-and-increment. *ACM Transaction on Parallel Computing*, 5(4), 2019. Article number 14.

[2] P. A. Buhr, D. Dice, and W. H. Hesselink. High-performance N-thread software solutions for mutual exclusion. *Concurrency Computat.: Pract. Exper.*, 27:651–701, 2015. http://dx.doi.org/10.1002/cpe.3263.

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

[4] W. H. Hesselink. Simulation refinement for concurrency verification. *Sci. Comput. Program.*, 76:739–755, 2011.

[5] W. H. Hesselink. Mutual exclusion by four shared bits with not more than quadratic complexity. *Sci. Comput. Program.*, 102:57–75, 2015. DOI:10.1016/j.scico.2015.01.001.

[6] W. H. Hesselink. Correctness and concurrent complexity of the Black-White Bakery algorithm. *Formal Aspects of Comput.*, 28:325–341, 2016. DOI: 10.1007/s00165-016-0364-4.

[7] W. H. Hesselink. Trylock, a case for temporal logic and eternity variables. *Science of Computer Programming*, 216(102767), 2022.

[8] J. Misra. *A discipline of multiprogramming: programming theory for distributed applications*. Spinger V., New York, 2001.

[9] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 7.1, System Guide, Prover Guide, PVS Language Reference*, 2020. `http://pvs.csl.sri.com`, accessed 1 Dec. 2021.