The Mailbox Problem

Marcos K. Aguilera[1] Eli Gafni[2] Leslie Lamport[1]

[1]Microsoft Research Silicon Valley
[2]UCLA

22 January 2009

Abstract

We propose and solve a synchronization problem called the *mailbox problem*, motivated by a particular type of interaction between a processor and an external device or between two threads. In this problem, a postman delivers letters to the mailbox of a home owner and uses a flag to signal a non-empty mailbox. The owner must remove all letters delivered to the mailbox and should not walk to the mailbox if it is empty. We present algorithms and an impossibility result for this problem.

Contents

1 Introduction

Computers typically use interrupts to synchronize communication between a processor and an I/O device. When a device has a new request, it raises an interrupt line to get the processor's attention. The processor periodically checks if the line has been raised and, if so, it interrupts its current task and executes an interrupt handler to process unhandled device requests. The interrupt line is then cleared so that it can be used when new requests come from the device. (In modern computers, the "device" is typically an interrupt controller that mediates between the processor and the actual I/O devices.) The processor must eventually execute the interrupt handler if there is an unhandled request. It is also desirable to avoid *spurious interrupts*, in which the processor executes the interrupt handler when there is no unhandled request. A closely related problem occurs in multi-threaded programming, in which the processor and the device are separate threads and the interrupt is some type of software signal [9, 11].

We study the following theoretical synchronization problem that arises from this setting, which we call the *mailbox problem*. From time to time, Bob the postman (the device) places letters (requests) for Eva the home owner (the processor) in a mailbox by the street. The mailbox has a flag that Eva can see from her house. She looks at the flag from time to time and, depending only on what she sees, may decide to go to the mailbox to pick up its contents, perhaps changing the position of the flag. The owner and the postman can leave notes for one another at the mailbox, but the notes cannot be read from the house. We require a protocol to ensure that (i) the owner picks up every letter placed in the mailbox and (ii) she never goes to the mailbox when it is empty (corresponding to a spurious interrupt). The protocol cannot leave the owner or the postman stuck at the mailbox, regardless of what the other does. For example, if the owner and postman are both at the mailbox when the postman decides to take a nap, the owner can return from the mailbox before the postman wakes up. We do not require the owner to receive letters that are still in the sleeping postman's bag. However, we interpret condition (i) to require that she be able to receive mail left by the postman in previous visits to the mailbox without waiting for him to wake up.

The following simple protocol was once used in computers. The postman/device raises the flag after he delivers a letter/request; the owner/processor goes to the mailbox if the flag is raised and lowers the flag after emptying the mailbox. This can cause a spurious interrupt if the postman goes to the mailbox while the flag is still raised from a previous visit, puts a letter in

the box, falls asleep for a long time, and then wakes up and raises the flag after the owner has meanwhile emptied the mailbox.

There are obviously no spurious interrupts with this protocol if the postman atomically (in a single step) delivers mail to the box and raises the flag, and the owner atomically removes mail from the box and lowers the flag. The problem is also solvable with notes if the reading or writing of a note and the raising or lowering of the flag can be performed atomically. Here is a simple algorithm that uses a single note written by the postman and read by the owner. The postman stacks letters in delivery order in the box. After delivering his letters, the postman as a single action writes the total number of letters he has delivered so far on his note and raises the flag. When she sees the flag up, the owner as a single action lowers the flag and reads the postman's note. Then, starting from the bottom of the stack, the owner removes only enough letters so the total number she has ever removed from the box equals the value she read on the note.

What if a single atomic action can only either read or write a note or read or write a flag? Then, we show that there are no non-blocking algorithms that use only 1-bit flags—neither with two 1-bit flags, one writable by the owner and one by the postman, nor with a single 1-bit flag writable by both. However, we were surprised to discover that there is a wait-free algorithm that uses two 14-valued single-reader flags. We do not know if there is one with smaller flags.

The mailbox problem is easily solved with the more general *signaling* problem, in which two processes must learn the current value of a function F of both their states by reading flags whose size depends on the number of possible values of F and not on the number of possible states. We give a solution to the signaling problem that is non-blocking but not wait-free. For the special case of the mailbox problem, we modify this solution to make it bounded wait-free. We don't know if there is a wait-free solution to the general problem.

The paper is organized as follows. We define the mailbox problem in Section 2 and the signaling problem in Section 3. Section 4 contains our algorithms and proofs of their correctness. A key building block of the algorithms is the *sussus* protocol described in Section 4.1. In Section 4.2, we give a solution to the signaling problem that uses flags with large timestamps, and we prove its correctness in Section 4.3. In Section 4.4, we show how to shrink these timestamps. For the particular instance of the mailbox problem, we explain in Section 4.5 how to change the non-blocking algorithm into a wait-free algorithm . Our impossibility results for the mailbox problem (and hence for the more general signaling problem) appear in Section 5. A con-

cluding section summarizes our results and describes related work and open problems.

The problems and our solutions are described precisely in the PlusCal algorithm language (formerly called $^+$CAL) [5]. The least error-prone way of showing the correctness of algorithms such as these is with assertional proofs that they implement their specifications under suitable refinement mappings [1, 8]. Since most computer scientists are unfamiliar with assertional refinement proofs, we instead provide more traditional behavioral proofs. In an attempt to avoid the errors endemic to such proofs, where appropriate we have used a careful, hierarchically structured proof style that we hope is self-explanatory. We use this style in the impossibility proofs too. As explained in Section 4.6, we have also checked the correctness of our algorithms with the TLC model checker.

2 The Mailbox Problem

We now state the mailbox problem more precisely. For simplicity, we let only one letter at a time be delivered to or removed from the mailbox. It is easy to turn a solution to this problem into one in which multiple letters can be delivered or removed.

2.1 Safety

We specify the problem and describe its solution using the PlusCal algorithm language [5]. We explain any PlusCal construct whose meaning may not be obvious. The *owner* and *postman* are the two processes shown in Figure 1. The *owner* process is declared to be process number 0 and the *postman* to be process number 1. The *check* procedure returns a Boolean value by setting

```
process (owner = 0)
  variable hasmail {
w1: while (TRUE) { call check() ;
w2:                if (hasmail) { (* remove letter ; *)
                                  call remove()        } } }

process (postman = 1) {
p1: while (TRUE) { (* add letter ; *)
                   call deliver()     } }
```

Figure 1: The *owner* and *postman* processes.

variable $counter = [i \in \{0, 1\} \mapsto 0]$;

procedure *deliver*() **{** *dl*: $counter[1] := counter[1] + 1$;
rdl: **return** **}**

procedure *remove*() **{** *rm*: $counter[0] := counter[0] + 1$;
rrm: **return** **}**

procedure *check*() **{** *ck*: $hasmail := (counter[0] < counter[1])$;
rck: **return** **}**

Figure 2: The abstract procedures.

the process-local variable *hasmail*. (In PlusCal, procedures return values by setting variables.) The *remove* and *deliver* procedures are used only for synchronization; the actual adding and removing of letters to the mailbox are performed by code inserted in place of the comments. Since it is only the correctness of the synchronization that concerns us, we largely ignore those comments and the code they represent.

Suppose the owner calls the *check* procedure after executing the *remove* procedure v times. We want the *check* procedure to return TRUE if it is called after the postman's *deliver* procedure has returned at least $v + 1$ times; we want the *check* procedure to return FALSE if it returns before the postman's *deliver* procedure has been called $v + 1$ times. It may return either value if the execution of the *check* procedure is concurrent with the $v+1^{\text{st}}$ execution of *deliver*.

Another way of expressing these requirements is to say that the *deliver*, *remove*, and *check* procedures must *simulate* the abstract procedures in Figure 2, where the **variable** statement declares *counter* to be an array indexed by the set $\{0, 1\}$ with $counter[0]$ and $counter[1]$ initially equal to 0. In PlusCal, the grain of atomicity is indicated by labels, an atomic action being an execution from one label to the next. Thus, the entire statement labeled *ck*, including the reading of both counters and the assignment to *hasmail*, is a single atomic step. (The code is an abstract specification of the problem, not a practical solution.) Simulating the abstract procedures means behaving as if there were an array variable *counter* that is changed the same way by the *remove* and *deliver* procedures, and is used in the same way by the *check* procedure to set the value of *hasmail*. ("As if" means that the actual procedures need not have a *counter* variable; but the value of the *check* procedure must be returned in a variable called *hasmail*.) Because the reading

of *counter*[1] by the owner in statement *ck* and its writing by the postman in statement *dl* are atomic, it is easy to see that *check*, *remove*, and *deliver* procedures that simulate the abstract procedures of Figure 2 satisfy the requirements described above. Since the simulated executions of the *dl*, *ck*, and *rm* statement of Figure 2 occur while executing the actual procedure's body, simulation of the procedures is equivalent to the usual condition of a linearizable implementation of the three operations [4].

2.2 Liveness

Simulation of the abstract procedures is a safety property. A solution should also satisfy a liveness property stating that, under some hypothesis, a process returns from each procedure call. We now state two liveness properties we might require. (Since the owner and postman have process numbers 0 and 1, for each process number i the other process is number $1-i$.)

Non-Blocking: For each i, if process i keeps taking steps when executing a procedure, then either that procedure returns or process $1-i$ performs an infinite number of procedure calls and returns.

Wait-Free: For each i, every procedure execution begun by process i returns if i keeps taking steps—even if process $1-i$ halts in the middle of executing a procedure [3]. The algorithm is said to be *bounded wait-free* [3] or *loop-free* [6] if each procedure returns before the calling process has taken N steps, for some fixed constant N.

Non-blocking is stronger than *deadlock-freedom* because it requires a process to return from a procedure call even if the other process halts while executing a procedure.

2.3 Process Communication and State

A solution requires the two processes to communicate, which they do with shared variables. We want the *check* procedure to be efficient, so it should access only a small amount of persistent state. We therefore add to the mailbox problem the following *access restriction*: The *check* procedure accesses shared memory only by reading a shared variable *Flag* that can assume only a small number of values; and the value returned by the procedure depends only on the values of *Flag* that it reads.

We consider only algorithms in which variables are atomic registers or arrays of atomic registers, and an atomic step can read or write at most one

atomic register. In all our algorithms, *Flag* is an array indexed by process. We call the atomic register $Flag[i]$ the *flag* of process i.

3 The Signaling Problem

The mailbox problem is easily solved with a solution to the more general *signaling problem*. In this problem, two processes numbered 0 and 1 can call the two procedures *read* and *write*. These procedures are required to simulate the abstract procedures in Figure 3.

Process i's execution of $write(v)$ sets $ownValue[i]$ to v, and the *read* procedure returns the current value of $F(ownValue[0], ownValue[1])$ in the process-local variable *returnVal*, where the operator F is a parameter of the problem. The array *ownValue* is initially equal to *InitValues*, which we assume to be an array indexed by $\{0, 1\}$ with values in a set *Values*, where *Values* and *InitValues* are also parameters of the problem. We assume that the argument of every call to *write* is an element of *Values*.

It's trivial to solve the mailbox problem using a solution to the signaling problem with *Values* the set of natural numbers and $F(x, y)$ defined to equal the Boolean value $(x < y)$. Each process maintains a counter. The *deliver* and *remove* procedures are implemented by the process incrementing its own counter and calling *write* with the new counter value; *check* is implemented by *read*.

The definitions of non-blocking and wait-free are the same for the signaling problem as for the mailbox problem. The access restriction applies to the *read* procedure rather than to *check*. Its requirement that the flags assume only a small number of values must be modified, since it obviously cannot be achieved if the set *FValues* of all $F(x, y)$ with x and y in *Values* is large—for example, if it is infinite. Instead, letting $|S|$ be the cardinality of a set S, we require that the number of values of each flag be a small multiple of $|FValues|$, independent of $|Values|$. In our solution, each flag has at most

variable $ownValue = InitValues$

procedure $write(newVal)$ **{** *wr*: $ownValue[self] := newVal$;
rtw: **return** **}**

procedure $read()$ **{** *rd*: $returnVal := F(ownValue[0], ownValue[1])$;
rtr: **return** **}**

Figure 3: The abstract procedures that specify the signaling problem.

$7 * |FValues|$ possible values.

4 Algorithms

We now give a non-blocking solution to the signaling problem satisfying the access restriction and use it to obtain a wait-free solution to the mailbox problem. In these solutions, each flag is a single-writer (atomic) register whose value is a record with a *Val* field that contains a value $F(x, y)$ for some x and y in *Values* and a *TS* field containing a timestamp that can assume only 7 values. (Remember that the array *Flag* has the two flags $Flag[0]$ and $Flag[1]$, and that $F(x, y)$ is the Boolean $x < y$ in the mailbox problem.) We do not know if there is a solution with smaller flags.

Our algorithms are based on the *sussus* protocol described in Section 4.1. Section 4.2 uses it to obtain a solution to the signaling problem that is non-blocking, but in which each flag has, instead of the *TS* field, a field *Timestamp* of timestamp values that can grow without bound. This solution is proved correct in Section 4.3. Section 4.4 shows how to bound the timestamps, and Section 4.5 obtains a wait-free solution to the mailbox problem by modifying the bounded-timestamp algorithm.

4.1 The *sussus* Protocol

The *sussus* protocol consists of a procedure $sussus(v)$ that can be called at most once by each of two processes, numbered 0 and 1. Intuitively, when a process i calls *sussus* with argument v_i, it tries to communicate v_i to process $1-i$ (the other process) and to learn the value (if any) communicated by that process. The procedure returns in variable *outcome* a value that is either "success" or "unknown" and in variable *outValue* a value that is either the value v_{1-i} with which the other process calls *sussus* or else the special value $\perp$. The outcome value "success" indicates that process i communicates its value successfully to the other process, provided that process also executes the *sussus* procedure. The outcome "unknown" indicates that process i does not know whether it communicates its value successfully. More precisely, the protocol is bounded wait-free and satisfies the following safety properties:

SU1. If both processes return from the procedure,[1] then at least one obtains the outcome "success".

[1] A process may never call the procedure or may stop taking steps inside it.

```
variables A = [i ∈ {0,1} ↦ ⊥] , B = [i ∈ 0..1 ↦ ⊥] ;
          (* A and B are initialized with A[i] = B[i] = ⊥ for i = 0, 1. *)

procedure sussus(v) {
s1: A[self] := v ;
s2: outValue := A[1−self];
    if (outValue = ⊥) outcome := "success" ;              (* Case A *)
    else {
s3:       B[self] := "done";
s4:       if (B[1−self] = ⊥) outcome := "unknown";      (* Case B *)
          else outcome := "success";                    (* Case C *) };
s5: return;                                                          }
```

Figure 4: The *sussus* procedure.

SU2. For each i, if process i returns from the procedure before process $1-i$ calls it, then process i obtains *outcome* equal to "success" and *outValue* equal to $\perp$.

SU3. For each i, if both processes return from the procedure execution and process i obtains the outcome "success", then process $1-i$ returns with *outValue* equal to the value with which process i called the procedure.

SU4. For each i, if process i returns from the execution and obtains the outcome "unknown", then process $1-i$ has already called the procedure with a value equal to the value of *outValue* obtained by process i.

Figure 4 shows the sussus procedure, written in PlusCal. The protocol uses two shared arrays A and B indexed by process number, initialized with $A[i] = B[i] = \perp$ for each i. The first step in process i's execution of *sussus*(v) sets element $A[i]$ to v. In the next step, process i reads $A[1-i]$ and stores the result in local variable *outValue*. If the value read is $\perp$, then process i sets *outcome* to "success". Otherwise, in a third step, process i sets $B[i]$ to "done" and, in a fourth step, it reads $B[1-i]$; if the result is $\perp$, process i sets *outcome* to "unknown", otherwise it sets *outcome* to "success". Observe that each atomic step accesses at most one array element of one shared variable.

Properties SU2 and SU4 follow easily from the code and the fact that each $A[i]$ is initially equal to $\perp$. To see why the protocol satisfies properties SU1 and SU3, observe that there are three possibilities for the values of variables *outcome* and *outValue* when a process completes its operation:

$\langle 0, A\rangle$	$\langle 1, B\rangle$	$\langle 0, C\rangle$	$\langle 1, C\rangle$	$\langle 0, B\rangle$	$\langle 1, A\rangle$
S	U	S	S	U	S

Figure 5: Possibilities when both processes complete execution of the *sussus* protocol.

Case A:	$outcome =$ "success",	$outValue = \perp$
Case B:	$outcome =$ "unknown",	$outValue \neq \perp$
Case C:	$outcome =$ "success",	$outValue \neq \perp$

These cases are indicated by comments in the code.

Figure 5 shows these cases as six pairs, where each pair $\langle i, \rho\rangle$ represents process i ending up in case ρ. Beneath each such pair, we indicate the outcome that process i obtains, with S standing for "success" and U for "unknown". Two adjacent pairs indicate the results obtained by each process in some execution. For example, we see the adjacent pairs $\langle 1, B\rangle$ and $\langle 0, C\rangle$ and the letters U and S beneath them. This indicates that, in some execution, process 1 ends up in case B with outcome "unknown", while process 0 ends up in case C with outcome "success". It turns out that every execution in which both processes return from their executions of *sussus* corresponds to some adjacent pair in the figure. It is easy to prove this by straightforward case analysis, and even easier by model checking the PlusCal code. (In fact, SU1–S4 can be verified directly by model checking.) Properties SU1 and SU3 follow easily from this fact together with the observation that the only value other than $\perp$ that process i can obtain is the one with which process $1-i$ called the procedure . (Remember that each process invokes operation *sussus* at most once.)

4.2 A Non-Blocking Algorithm with Large Flags

We now present a solution to the signaling problem that is non-blocking, but in which flags contain unbounded integer timestamps. In the *write* procedure, process i sets $Flag[i]$ to a record whose *Timestamp* field is a timestamp and whose *Val* field is what the process thinks is the current value of $F(ownValue[0], ownValue[1])$. (The algorithm does not actually use a variable *ownValue*, although the last value with which *write* was called by each process can be inferred from the values of other variables.) The *read* procedure returns the value $Flag[j].Val$ for the record $Flag[j]$ with the largest *Timestamp* field. (We will show that, if the two timestamps are equal, then the two *Val* fields are equal.)

When executing $write(v)$, a process i uses the *sussus* protocol to communicate the value v (its current value of $ownValue[i]$) to the other process. More precisely, it executes a sequence of numbered rounds, each performing a new instance of *sussus*, until a *sussus* instance succeeds. It then sets $Flag[i].Timestamp$ to the number of that successful round and sets $Flag[i].Val$ to $F(ownValue[0], ownValue[1])$, using for $ownValue[1-i]$ the most recent output value different from $\perp$ it obtained from the *sussus* protocols.

The shared variables and the procedures are shown in Figure 6. The algorithm uses the following process-local variables in addition to $returnVal$:

outcome, *outValue* The variables used to return the results of a call to a procedure named *multisussus*.

round The number of the current round, initially equal to 0.

otherVal The most recently read value of the other process. For process i, its initial value is $InitValues[1-i]$ (the initial value of $ownValue[1-i]$ in the specification).

A procedure call of $multisussus(rnd, v)$ is the call $sussus(v)$ of the round rnd instance of the *sussus* protocol, with $A[rnd, i]$ and $B[rnd, i]$ being the registers $A[i]$ and $B[i]$ of that instance. The expression $[Timestamp \mapsto t,\ Val \mapsto v]$ denotes a record with a *Timestamp* field that equals t and a *Val* field that equals v.

For simplicity, we have written the *read* procedure so that process i returns the *Val* component of its own register $Flag[i]$ only if $Flag[i].Timestamp$ is strictly greater than $Flag[1-i].Timestamp$. As we remarked above, if those two timestamps are equal, then $Flag[i].Val$ will equal $Flag[1-i].Val$ and the procedure can return either value. This is proved in Section 4.3.2 below.

4.3 Correctness

4.3.1 Proof that the Algorithm is Non-Blocking

We first show that the algorithm is non-blocking. Execution of *read* obviously completes in two steps. To show that execution of *write* is non-blocking, we assume that process i takes infinitely many steps of the *write* procedure without returning, and prove that process $1-i$ performs infinitely many complete executions of *write*.

Process i must perform infinitely many rounds. If process $1-i$ performed finitely many rounds, then eventually process i would perform a round that

```
variables Flag = [i ∈ {0,1} ↦ [Timestamp ↦ 0,
                                Val ↦ F(InitValues[0], InitValues[1])]],
          A = [r ∈ Nat, i ∈ {0,1} ↦ ⊥],
          B = [r ∈ Nat, i ∈ {0,1} ↦ ⊥]
  (* Initially Flag[i].Timestamp = 0, Flag[i].Val = F(InitValues[0], InitValues[1]),
     and A[r,i] = B[r,i] = ⊥ for all i and r. *)

procedure read()
  variable readFlag {
rd:  readFlag := Flag[1−self];
rd1: if (Flag[self].Timestamp > readFlag.Timestamp)
       returnVal := Flag[self].Val;
     else returnVal := readFlag.Val;
rtr: return                                                    }

procedure write(newVal)
  variable done = FALSE; {
wr:  while (¬done) { round := round + 1;
                     call multisussus(round, newVal);
wr1:                 if (outValue ≠ bot) otherVal := outValue;
                     if (outcome = "success") done := TRUE     };
     Flag[self] := [Timestamp ↦ round,
                    Val ↦ IF self = 0 THEN F(newVal, otherVal)
                              ELSE F(otherVal, newVal)];
rtw: return                                                    }

procedure multisussus(rnd, v) {
s1: A[rnd, self] := v;
s2: outValue := A[rnd, 1−self];
    if (outValue = ⊥) outcome := "success";
    else {
s3:     B[rnd, self] := "done";
s4:     if (B[rnd, 1−self] = ⊥) outcome := "unknown";
        else outcome := "success" };
s5: return                                                     }
```

Figure 6: The global variables and *read* and *write* procedures of the Signaling algorithm with unbounded round numbers.

process $1-i$ never began. By property SU2 of the *sussus* protocol, process i would succeed in that round and return from *write*, contrary to hypothesis. Hence both i and $1-i$ perform infinitely may rounds. Property SU1 implies that for at least one process, infinitely many of the calls of *multisussus* must succeed. That process must exit from *write* infinitely many times, completing the proof.

4.3.2 Proof of Safety

We now prove that the *read* and *write* procedures of Figure 6 simulate the abstract procedures that specify them. We do this for the more general algorithm in which the *read* can return the *Val* field of either flag if it finds the timestamps of the two flags to be equal in statement *rd*1.

We say that a process *succeeds* in round r if it obtains the outcome "success" in its round r execution of *multisussus*, and we say that a *write* by process i *completes* when it writes $Flag[i]$. To avoid having to consider initial values as a special case, we pretend that there was an initial execution of $write(InitValues[i])$ by each process i that succeeded in round 0 and preceded all calls of *read*.

We prove that the specification procedures are simulated if the value of $ownValue[i]$ is defined to be written during process i's execution of *write* when either the procedure writes $Flag[i]$ or when process $1-i$ reads the value (in statement $s2$ of *multisussus*) with which the *write* procedure was called—whichever happens first. To prove simulation, it suffices to show that, with this definition of $ownValue$, an execution of the *read* procedure returns the value that $F(ownValue[0], ownValue[1])$ had when it executed statement *rd*. Let the "current" instant be the one immediately preceding that execution of statement *rd*. Throughout the proof, we let *Flag* denote the current value of the variable *Flag* (its value in that "current" instant).

1. It suffices to let i be such that $Flag[1-i].Timestamp \geq Flag[i].Timestamp$ and prove $Flag[1-i].Val = F(ownValue[0], ownValue[1])$.
   PROOF: The *read* returns the current value of one of the flags. Since we have not assumed which process is executing the *read*, we can assume by symmetry that it returns $Flag[1-i].Val$. It can do this only if $Flag[1-i].Timestamp \geq Flag[i].Timestamp$, so it suffices to show that $Flag[1-i].Val$ has the correct value $F(ownValue[0], ownValue[1])$ under this assumption.

In the proof that $Flag[1-i].Timestamp \geq Flag[i].Timestamp$ implies $Flag[1-i].Val = F(ownValue[0], ownValue[1])$, we are assuming that some

process is about to execute the *rd* statement. We note for future reference that this proof requires only the weaker assumption that the process is not between the call of *write* and the writing of its flag.

2. Process $1-i$ set *otherVal* to the value with which process i called *multisussus* in round $Flag[i].Timestamp$.
   PROOF: Process i succeeded in round $Flag[i].Timestamp$ (since it wrote $Flag[i]$ in that round). The step 1 assumption implies that process $1-i$ completed round $Flag[i].Timestamp$, so property SU3 implies that it obtained the value $Flag[i].Val$ written by i during the execution of *multisussus* for that round.

3. Process i did not complete a *write* after executing any round later than $Flag[i].Timestamp$.
   PROOF: If it had completed a later write, it would have rewritten $Flag[i]$ with a larger *Timestamp* field.

4. The current value of $ownValue[i]$ is the value that *otherVal* had in process $1-i$ when that process wrote $Flag[1-i]$.
   PROOF: The proof is broken into two cases:

   4.1. CASE: In every execution of *multisussus* in a round after $Flag[i].Timestamp$, process $1-i$ obtained only the value $\perp$.
   PROOF: In this case, when process $1-i$ wrote $Flag[1-i]$, its value of *otherVal* was still the value it had in round $Flag[i].Timestamp$, which is the value with which process i called *write* when the procedure wrote $Flag[i]$. That value is the current value of $ownValue[i]$ by our definition of *ownValue*.

   4.2. CASE: Process $1-i$ obtained a value v when executing *multisussus* in some round after $Flag[i].Timestamp$.
   PROOF: By step 3, process i must have been executing $write(v)$ when process $1-i$ obtained the value v. The definition of *ownValue* implies v equals the current value of $ownValue[i]$, and the algorithm implies that v was the value of process $(1-i)$'s variable *otherVal* when it wrote $Flag[1-i]$.

5. The current value of $ownValue[1-i]$ is the value that *newVal* had in process $1-i$ when that process wrote $Flag[1-i]$.
   PROOF: The proof is broken into two cases:

   5.1. CASE: The current execution of *read* is being performed by process $1-i$.
   PROOF: In this case, process $1-i$ could not have called *write* after writ-

ing $Flag[1-i]$, so the definition of $ownValue[1-i]$ implies that it equals the value of $newVal$ in the call of *write* in which $1-i$ wrote the current value of $Flag[1-i]$.

5.2. CASE: The current execution of *read* is being performed by process i. PROOF: In this case, process i is not currently executing *write*, so step 3 implies it did not perform any round after round $Flag[i].Timestamp$. By the step 1 assumption, process i did not read any value with which process $1-i$ called *multisussus* after writing the current value of $Flag[1-i]$. The definition of $ownValue[1-i]$ therefore implies that its current value equals the value $newVal$ had in process $1-i$ when that process wrote the current value of $Flag[1-i]$.

6. Q.E.D.
PROOF: Steps 4 and 5 imply that process $1-i$ set $Flag[1-i].Val$ to $F(ownValue[0], ownValue[1])$, which by step 1 is what we had to prove.

We now show that if two flags have equal *Timestamp* fields, then they also have equal *Val* fields. Since this is true initially, we need only show that it is true after a flag is written. By our observation after step 1, the proof shows that

$$(Flag[1-i].Timestamp \geq Flag[i].Timestamp) \Rightarrow (Flag[1-i].Val = F(ownValue[0], ownValue[1]))$$

is true immediately after a process writes its flag. If the flags have equal *Timestamp* fields after the write, then their *Val* fields are also equal because they both equal $F(ownValue[0], ownValue[1])$.

4.4 A Non-blocking algorithm with Small Flags

Let *Sig* be the algorithm of Section 4.2. We modify *Sig* to obtain a new algorithm *BSig* in which the unbounded timestamps in *Flag* are replaced with timestamps from a set *TStamps* containing only 7 values. More precisely, we replace each field $Flag[i].Timestamp$ whose value is an unbounded round number with the field $Flag[i].TS$ whose value is an element of *TStamps*. Hence, if $Flag[i].Val$ can assume at most n possible values for each i, then *Flag* assumes at most $(7n)^2$ possible values. We assume a relation $\succ$ on the set *TStamps* that, for now, we think of as a total ordering.

To derive *BSig*, we first modify *Sig* by adding a process-local variable *ts* and adding to $Flag[i]$ a *TS* field that is set to the current value of *ts* when $Flag[i]$ is written. We then show that in statement *rd*1, instead of comparing

the *Timestamp* fields with $>$, we can compare the *TS* fields with $\succ$. In other words, the **if** test in *rd*1 becomes

$$Flag[self].TS \succ readFlag.TS$$

The *Timestamp* fields are then never used, so we can eliminate them to obtain *BSig*. Of course, our problem is figuring out to what value *ts* should be set.

As we stated above, it makes no difference whether the **if** or **else** clause in *rd*1 is executed if the two flags' *Timestamp* fields are equal. To replace the comparison of the *Timestamp* fields by the comparison of the *TS* fields, it suffices to ensure that the following assertion $BInv(i)$ is invariant (true at all times) for each i:

$$\begin{aligned} BInv(i) \;\triangleq\; & (Flag[i].Timestamp > Flag[1-i].Timestamp) \\ & \Rightarrow\; (Flag[i].TS \succ Flag[1-i].TS) \end{aligned}$$

Since both *Timestamp* fields initially equal 0, $BInv(i)$ is true initially. To make it an invariant, we must ensure it is left true by any step that changes *Flag*. That is, we must ensure that $BInv(i)$ is true immediately after a step that writes either $Flag[i]$ or $Flag[1-i]$.

We further modify algorithm *Sig* by having the *write* procedure use the *multisussus* procedure to communicate not just *newVal* but also the current value of *ts*. In other words, in each round, process i calls *multisussus* with the record value $[Val \mapsto newVal,\ TS \mapsto ts]$. If the round succeeds, the process will write *ts* into the *TS* field of $Flag[i]$. We define the *round r value* of *ts* to be the value *ts* has when the process executes *multisussus* in round r and, hence, the value to which $Flag[i].TS$ is set if that execution succeeds.

We must deduce what the value of *ts* should be in each round to ensure the invariance of $BInv(i)$. We do this by trying to prove that $BInv(i)$ is true after a step that changes *Flag* and seeing what properties the value of *ts* must have to make the proof work. We now assume $Flag[i].Timestamp > Flag[1-i].Timestamp$ and prove $Flag[i].TS \succ Flag[1-i].TS$.

1. Process $1-i$ executed *multisussus* in round $Flag[1-i].Timestamp$ with *ts* equal to $Flag[1-i].TS$, and that execution succeeded.
   PROOF: The algorithm implies that process $1-i$'s execution of *multisussus* in round $Flag[1-i].Timestamp$ succeeded, and that the value of *ts* for that round was $Flag[1-i].TS$.
2. Process i set $otherVal.TS$ to $Flag[1-i].TS$ in round $Flag[1-i].Timestamp$.

PROOF: By step 1 and property SU3, since the hypothesis $Flag[i].Timestamp > Flag[1-i].Timestamp$ implies that i executed that round.

3. CASE: Process $1-i$ has just written $Flag[1-i]$.

   3.1. Process $1-i$ has not executed any round after round $Flag[1-i].Timestamp$.

   PROOF: By the case assumption, which implies process $1-i$ has just finished round $Flag[1-i].Timestamp$.

   3.2. Process i executed rounds $Flag[1-i].Timestamp$ through $Flag[i].Timestamp$ and, after each of them, its value of $otherVal.TS$ equaled $Flag[1-i].TS$.

   PROOF: The algorithm and the hypothesis $Flag[i].Timestamp > Flag[1-i].Timestamp$ implies that i executed those rounds; step 2 asserts that process i's value of $otherVal.TS$ equaled $Flag[i].TS$ after round $Flag[1-i].Timestamp$; and 3.1, SU2, and the *write* procedure's code imply that $otherVal.TS$ also equaled $Flag[i].TS$ after rounds $Flag[1-i].Timestamp + 1$ through $Flag[i].Timestamp$.

   3.3. Q.E.D.

   PROOF: The desired conclusion $Flag[i].TS \succ Flag[1-i].TS$ follows from 3.2 if the values of ts satisfy the following condition.

   TS1. In every round r, process i's value of ts satisfies $ts \succ otherVal.TS$, where $otherVal$ is the variable's value after process i finished round $r-1$, or its initial value if $r = 1$, assuming its initial value satisfies $otherVal.TS = Flag[1-i].TS$.

4. CASE: Process i has just written $Flag[i]$.

   4.1. CASE: Process $1-i$ wrote $Flag[1-i]$ after process i finished its round $Flag[i].Timestamp - 1$ execution of *multisussus*.

   PROOF: The algorithm implies that process $1-i$ succeeded in round $Flag[1-i].Timestamp$. The hypothesis $Flag[i].Timestamp > Flag[1-i].Timestamp$ implies that process i executed that round, and SU3 and the algorithm imply that process i set $otherVal.TS$ equal to $Flag[1-i].TS$ during that round. The 4.1 case assumption implies that process $1-i$ did not begin any later rounds before process i finished round $Flag[i].Timestamp - 1$, so SU2 implies $otherVal.TS$ still equaled $Flag[1-i].TS$ after that round. Property TS1 (introduced in step 3.3 above) for $r = Flag[i].Timestamp$ then implies $Flag[i].TS \succ Flag[1-i].TS$, since $Flag[i].TS$ equals the value of ts in process i for

round $Flag[i].Timestamp$.

4.2. CASE: Process $1-i$ wrote the current value of $Flag[1-i]$ before process i finished its round $Flag[i].Timestamp - 1$ execution of *multisussus*.

PROOF: The desired conclusion $Flag[i].TS \succ Flag[1-i].TS$ follows from case assumption 4.2 if the value of ts satisfies the following condition.

TS2. In each round r, process i's value of ts satisfies $ts \succ Flag[1-i].TS$, where $Flag[1-i]$ is a value read after process i returns from its round $r-1$ execution of *multisussus*, or is the initial value of $Flag[1-i]$ if $r = 1$.

4.3. Q.E.D.

PROOF: Process $1-i$ wrote $Flag[1-i]$. The hypothesis implies $Flag[i].Timestamp > 0$, so process i executed round $Flag[i].Timestamp - 1$. (If $Flag[i].Timestamp = 1$, this round is the posited initial round 0.) Therefore, cases 4.1 and 4.2 are exhaustive.

5. Q.E.D.

PROOF: Since we have to prove that $BInv(i)$ remains true when a process writes its flag, the cases of steps 3 and 4 are exhaustive.

We have shown that we obtain a correct algorithm *BSig* if ts satisfies properties TS1 and TS2. These properties can be combined into:

TS12. For each process i, the round r value of ts must satisfy $ts \succ otherVal.TS$ and $ts \succ Flag[1-i].TS$, where $otherVal$ and $Flag[1-i]$ are the variables' values at the end of round $r-1$ or, for $r = 1$, are their initial values that must satisfy $otherVal.TS = Flag[1-i].TS$.

We must satisfy TS12 when ts assumes values in a bounded set *TStamps* with a relation $\succ$. We have been thinking of $\succ$ as a total ordering on *TStamps*, but we actually require only that the relation $\succ$ satisfy the following two properties, for all elements v and w in *TStamps*:

- Antisymmetry: if $v \succ w$ is true then $w \succ v$ is false.
- There exists an element $Dominate(v, w)$ in *TStamps* such that $Dominate(v, w) \succ v$ and $Dominate(v, w) \succ w$.

In particular, we do not require that $\succ$ be transitive. A computer search reveals that the smallest set with the requisite relation $\succ$ contains 7 elements. We take *TStamps* to be the set {"T1", ..., "T7"} and define $\succ$ and *Dominate* as follows, where the value of the expression CHOOSE $x \in S : P$ is an arbitrary element x of S satisfying P (assuming there is such an x).

$$
\begin{array}{rcl}
v \succ w & \triangleq & \quad (v = \text{“T1”} \ \land \ w \in \{\text{“T3”}, \text{“T4”}, \text{“T5”}\}) \\
& & \lor \ (v = \text{“T2”} \ \land \ w \in \{\text{“T1”}, \text{“T3”}, \text{“T7”}\}) \\
& & \lor \ (v = \text{“T3”} \ \land \ w \in \{\text{“T5”}, \text{“T6”}, \text{“T7”}\}) \\
& & \lor \ (v = \text{“T4”} \ \land \ w \in \{\text{“T2”}, \text{“T3”}, \text{“T6”}\}) \\
& & \lor \ (v = \text{“T5”} \ \land \ w \in \{\text{“T2”}, \text{“T4”}, \text{“T7”}\}) \\
& & \lor \ (v = \text{“T6”} \ \land \ w \in \{\text{“T1”}, \text{“T2”}, \text{“T5”}\}) \\
& & \lor \ (v = \text{“T7”} \ \land \ w \in \{\text{“T1”}, \text{“T4”}, \text{“T6”}\})
\end{array}
$$

$$
\mathit{Dominate}(v, w) \ \triangleq \ \text{CHOOSE } x \in \mathit{TStamps} : (x \succ v) \land (x \succ w)
$$

The TLC model checker has verified that $\succ$ and *Dominate* satisfy the required properties.

To satisfy TS12, we introduce a process-local variable *nextts* that process i sets to $\mathit{Dominate}(\mathit{otherVal}.\mathit{TS}, \ \mathit{Flag}[1-i].\mathit{TS})$ at the end of the round and have process i set *ts* to *nextts* at the beginning of the round. (To satisfy TS12, we must let the initial value of *nextts* satisfy $\mathit{nextts} \succ \mathit{Flag}[1-i].\mathit{TS}$.) The new algorithm, which we call *BSig*, is obtained from *Sig* by the following modifications:

- For each process i, the initial value of $\mathit{Flag}[i].\mathit{TS}$ is arbitrarily chosen to equal “T1” (rather than 0), and *otherVal* is initially equal to $[\mathit{Val} \mapsto \mathit{InitValues}[i], \ \mathit{TS} \mapsto \text{“T1”}]$.

- The process-local variables *ts* and *nextts* are added, with *nextts* initially equal to “T2” (which makes $\mathit{nextts} \succ \mathit{Flag}[1-i]$ hold initially for each i).

- The **if** test of statement $rd1$ is changed to:

  $rd1$: **if** $(\mathit{Flag}[\mathit{self}].\mathit{TS} \succ \mathit{readFlag}.\mathit{TS})$

- The *write* procedure is changed to the one in Figure 7.

It is easy to check that the modified algorithm satisfies TS12, which implies its correctness. More precisely, TS12 implies that algorithm *BSig* satisfies the required safety property because algorithm *Sig* does. By its construction, *BSig* obviously satisfies the same non-blocking property as *Sig*.

4.5 The Bounded Wait-Free Mailbox Algorithm

The algorithms of Sections 4.2 and 4.4 are non-blocking but not wait-free, because one process could remain forever in the *write* procedure while the other process performs an infinite sequence of calls to and returns from *write*.

```
procedure write(newVal)
  variable done = FALSE ; {
wr:  while (¬done) {
          round := round + 1 ;
          ts := nextts ;
          call multisussus(round, [Val ↦ newVal, TS ↦ ts ]) ;
wr1:      if (outValue ≠ bot) otherVal := outValue ;
          nextts := Dominate(otherVal.TS, Flag[1−self].TS) ;
          if (outcome = "success") done := TRUE                    } ;
     Flag[self] := [TS ↦ ts ,
                    Val ↦ IF self = 0 THEN F(newVal, otherVal.Val)
                                      ELSE F(otherVal.Val, newVal) ] ;
rtw: return                                                          }
```

Figure 7: The *write* procedure for the bounded-timestamp algorithm *BSig*.

We do not know of a wait-free solution to the general signaling problem. However, we now show that a simple modification of the signaling algorithm yields a bounded wait-free solution for the special case of the mailbox problem.

Figure 8 shows how the *read* and *write* procedures of the signaling problem are used to implement the three procedures of the mailbox problem, where *wcounter* and *pcounter* are local counter variables of the owner and postman, respectively, that initially equal 0, and the signaling algorithm is used with $F(u, v)$ equal to the Boolean-valued expression $u < v$. It is clear that these *deliver*, *remove*, and *check* procedures satisfy their specifications in Figure 2. A mailbox algorithm calls these procedures as shown in Figure 1, with the owner being process 0 and the postman process 1.

To make the algorithm wait-free, we have the owner return from *write* without writing her flag if she observes the postman's counter to be greater than hers. We now show that this modified signaling algorithm still satisfies the required safety property when used in a mailbox algorithm. Namely, we show that the *read* and *write* procedures still simulate their specifications if the owner calls *write* (in the *remove* procedure) only after a call to *read* (in the *check* procedure) returns the value TRUE.

Recall that the unmodified procedures simulate their specification if $ownValue[i]$ is considered to be changed by a *write* operation when either it writes $Flag[i]$ or else the value being written is seen (through a call to *multisussus*) by process $1-i$. For the modified algorithm, we consider $ownValue[0]$ to be changed at any arbitrary point during the owner's *write*

```
procedure deliver() { dl:  pcounter := pcounter + 1;
                           call write(pcounter);
                     rdl:  return                    }

procedure remove() { rm:   wcounter := wcounter + 1;
                           call write(wcounter);
                     rrm:  return                    }

procedure check() { ck:    call read();
                    ck1:   hasmail := returnVal;
                    rck:   return                  }
```

Figure 8: Implementing the mailbox with signaling.

operation. We consider *ownValue*[1] to be changed by the postman's *write* operation when either (a) it writes *Flag*[1]; (b) the value being written is seen by the owner, who exits the write "normally" (by writing *Flag*[0]); or (c) the value being written is seen by the owner, who exits abnormally, and the values of the flags indicate that a *read* (done at that instant) will return TRUE—whichever of (a), (b), or (c) occurs first.

The change of each *ownValue*[i] occurs during a *write*, as required. We need only show that a *read* (performed only by the owner) returns the correct value—that is, a value consistent with the simulated values of each *ownValue*[i] at some point during the *read*. If the owner exits *write* normally, the flag values and the simulated values of the *ownValue*[i] when the owner performs the next read are the same as in the unmodified signaling algorithm. By correctness of the unmodified algorithm, the *read* returns the correct value in this case. We therefore need only prove that a read returns the correct value if the owner's previous *write* returned without setting her flag. We assume that the owner (i) executed *write*(v), (ii) read a value w of the postman's counter with $w > v$ and exited without writing her flag, and (iii) has just finished executing *read*; and we show that the *read* returns a correct value. There are two cases:

1. CASE: The call of *read* returns TRUE.
   PROOF: The abnormal exit of the owner implies that she observed the postman's counter to be greater than her own. The TRUE result is therefore correct by definition of the current value of *ownValue*[i]. For *ownValue*[1], this follows either by case (a) or, if the postman's *write*(w) is still in progress, by case (c).

2. CASE: The call of *read* returns FALSE.

2.1. At some instant t_1 during the owner's previous execution of *read*, the flags indicated that her counter was less than the postman's.
PROOF: Because *returnVal* must equal TRUE for the owner to call *write*.

2.2. At some instant t_2 during the owner's just-completed *read* execution, $Flag[1].Val = \text{FALSE}$ and that value of $Flag[1]$ was written after t_1.
PROOF: By 2.1 and the case assumption, since the owner did not write $Flag[0]$ between t_1 and executing the just-completed *read*.

2.3. The value of $Flag[1]$ at instant t_2 was written by the postman's execution of $write(v)$.
PROOF: At t_1, the value of *wcounter* was $v-1$, so 2.1 implies that *pcounter* was at least v. Since *wcounter* was at most v between t_1 and t_2, a write of $Flag[1]$ with $Flag[1].Val = \text{FALSE}$ between t_1 and t_2 could only have been performed by the postman executing $write(v)$.

2.4. The postman's execution of $write(v+1)$ did not complete by t_2.
PROOF: By 2.3.

2.5. Q.E.D.
PROOF: The return of FALSE is correct for a *read* of the $ownValue[i]$ at t_2, since the definition of when $ownValue[i]$ changes implies that $ownValue[0] = ownValue[1] = v$ at that instant.

The modified *write* procedure is obtained from the one in Figure 7 by replacing the ninth line

if (*outcome* = "success") *done* := TRUE **}**;

with

if (*outcome* = "success") *done* := TRUE
if (*self* = 0 $\wedge$ *otherVal.Val* > *newVal*)
*wr*2: **return** **}**;

If the owner exits *write* by seeing a higher counter value, then she knows that there is a letter in the mailbox and can remove it without performing a *read*. We do not bother with this optimization.

We now prove that the modified algorithm is bounded wait-free. In the proof, we consider the *write* procedure to have completed when it has either written the process's flag or, in the case of the owner, has seen a postman's counter value greater than her own. We say that process *i succeeds* (respectively, *fails*) in round r if it executes round r of the *write* procedure and the procedure exits (respectively, does not exit) in that round.

1. The owner can set $A[r,0]$ to a value v only after the postman's execution of *write*$(v-1)$ has completed.
   PROOF: We assume that the postman has not completed his execution of *write*$(v-1)$ and the owner has set $A[r,0]$ to v, and we obtain a contradiction.

   1.1. A call of *read* by the owner when her counter equaled $v-1$ returned TRUE.
   PROOF: The owner set $A[r,0]$ to v when executing *write*(v), which she does only after her counter equals $v-1$ and she executes a *read* operation that returns TRUE.

   1.2. The *read* of step 1.1 simulates an execution of the *read* in the specification of Figure 3 during which $ownValue[0] = v-1$ and $ownValue[1] \leq v-1$.
   PROOF: By the correctness of the signaling algorithm (which implies that it simulates the abstract procedures), step 1.1 (which implies $ownValue[0] = v-1$), and the assumption that the postman has not yet completed a write of $v-1$ (which implies $ownValue[1] \leq v-1$).

   1.3. Q.E.D.
   PROOF: Step 1.2 contradicts step 1.1, since *read* must return the value $ownValue[0] < ownValue[1]$.

2. For any round r, if both processes have performed their round-r assignment to A, then $A[r,0]-1 \leq A[r,1] \leq A[r,0]+1$.
   PROOF: By induction on r. It is trivial for round 1, in which each process i sets $A[1,i]$ to 1. We now assume that the inequality is true for r and prove that it holds for $r+1$.

   2.1. If $A[r,1] = A[r,0]+1$, then it is impossible for the postman to succeed and the owner to fail in round r.
   PROOF: If the postman's *write* succeeds in round r, then *sussus* property SU3 implies that the owner obtains the value $A[r,1]$ in that round, which by hypothesis is greater than her counter's value. Hence, she must succeed in that round.

   2.2. If $A[r,0]-1 = A[r,1]$, then it is impossible for the owner to succeed and the postman to fail in round r.
   PROOF: By step 1, the owner's write of $A[r,0]$ occurred after the postman's execution of *write*$(A[r,1])$, which is when the postman performed round r. Hence, the postman finished executing round r before the owner began executing round r. By *sussus* property SU2, this implies that the postman must have succeeded in round r.

2.3. Q.E.D.

Because counter values are incremented by 1, if a *write* by process i succeeds in round r, then in round $r{+}1$ the process sets $A[r{+}1, i]$ to $A[r, i] + 1$. Therefore, the step 2 inequality can hold for r and not for $r{+}1$ only in the two cases that steps 2.1 and 2.2 show to be impossible.

3. The algorithm is bounded wait-free.

PROOF: We must show that if a process i calls *write*, then it succeeds within a bounded number of rounds. Process i can fail in a round r only if it obtains the other process's value in that round (by SU4), which means that process $1{-}i$ must have written $A[r, 1{-}i]$. By *sussus* property SU1, at least one process succeeds in each round. Since a process that succeeds starts the next round with a larger value, and step 2 implies that the values written in the same round can differ by at most 1, it is obvious that process i must succeed within a bounded number of rounds. A little thought reveals that it can take up to three rounds.

4.6 Model Checking

We have presented three algorithms: *Sig*, the signaling algorithm with unbounded timestamps (Section 4.2); *BSig*, the signaling algorithm with bounded timestamps (Section 4.4); and the wait-free mailbox algorithm based on *BSig* (Section 4.5) that we call here *MB*. We have provided correctness proofs for these algorithms that we believe most researchers in concurrent algorithms will find convincing. However, the traditional behavioral style of proof that we used is inherently unreliable, in part because it provides no rigorous connection between the actions that the proof claims the algorithm performs and the actual code. We therefore directly checked the PlusCal code using the PlusCal to TLA$^+$ translator [5] and TLC, the TLA$^+$ model checker [12]. To check a signaling algorithm, we must specify F and write driver code that determines the parameters *Values* and *InitValues* and calls the *read* and *write* procedures. We checked the signaling algorithms with $F(u, v)$ equal to the pair $\langle u, v \rangle$ and with driver code in which, as in *MB*, the n^{th} call of *write* by each process has n as its argument. Algorithms *Sig* and *BSig* are independent of F and the values being written, in the sense that they do not make any decision based on F or those values. We therefore expected any error in the algorithm to manifest itself with this choice of F and driver code.

We checked that each algorithm satisfies its PlusCal specification under a suitable refinement mapping [1]. We also checked that *BSig* satisfies the invariants $BInv(i)$ of Section 4.4 and that *MB* satisfies the invariant of the

key second step in the proof of wait-freedom (Section 4.5). We checked *Sig* and *BSig* for all executions in which each process writes at most 4 times, and *MB* for all executions in which the postman delivers at most 5 letters.

Our experience provides an indication of the adequacy of these checks. After finding the basic algorithms, we made numerous errors in writing the PlusCal code, the invariants, and the refinement mappings. TLC found all but two of those errors in executions with at most 2 writes or letter deliveries. One exception was an error in *MB* that TLC found only by checking executions with 3 letters. (Having TLC find these errors before we tried writing proofs saved us a lot of time.) The other was an error in *Sig* and *BSig* in which process i's variable *otherVal* was initialized to $InitValues[i]$ rather than $InitValues[1-i]$. This error was missed by model checking because all our tests had $InitValues[0] = InitValues[1]$; it was discovered only when writing the proofs. This experience and our intuition suggest that checking executions with more writes or letter deliveries would not find any further errors. However, we could have missed some basic gap in our checking other than that caused by the symmetry of *InitValues*. Moreover, the PlusCal code provided here was formatted by hand, so it could differ from the code that was checked. Still, the model checking combined with our proofs and our checking of the manuscript gives us a high degree of confidence in the correctness of these algorithms—a degree of confidence it would have been much harder to achieve without model checking.

5 Impossibility Results

We now show that it is impossible to solve the mailbox problem in two cases when *Flag* holds small values:

- *Flag* consists of two bits, each writable by a single process; and
- *Flag* consists of one bit writable by both processes.

These results hold even if the postman and owner can access unbounded shared registers during the execution of *deliver* and *remove*.

5.1 Preliminary definitions

To prove these results, we first give some definitions and simple properties of the mailbox problem in terms of executions, where an execution is a sequence of steps and a *step* is taken to be a primitive concept. We let $\circ$ be sequence concatenation and define $\langle s_1, \ldots, s_n \rangle$ to be an n-element sequence, with

$\langle\,\rangle$ the empty sequence. We define a *mailbox execution* to be an execution in which each step is either a call of one of the three procedures or else a step executed by one those procedures, and which satisfies the following conditions:

E1. Calls and returns must be properly matched. (That is, a return must be preceded by a corresponding call, and two calls by the same process cannot occur without an intervening return.)

E2. A call of *remove* can occur only if there is a preceding step of process 0 and the last such step is a return of *check* with value TRUE.

E3. A call of *check* can occur only if there is no preceding step of process 0, or if the last such step is a return from *remove* or a return from *check* with value FALSE.

Observe that the empty sequence $\langle\,\rangle$ is a mailbox execution. We make the following definitions, for any finite mailbox execution E.

- $M(E)$ equals the number of calls of *deliver* minus the number of returns of *remove* in E.
- $N(E)$ equals the number of returns of *deliver* minus the number of returns of *remove* in E.

We identify an algorithm $\mathcal{A}$ with its set of all possible executions, which is prefix-closed, meaning that every prefix of an execution in $\mathcal{A}$ is in $\mathcal{A}$. If $\mathcal{A}$ is an algorithm, then this set of mailbox executions satisfies the following four properties:[2]

Process-Enabled: For every finite execution E in $\mathcal{A}$ and every process i, there exists a step s of i such that $E \circ \langle s \rangle$ is in $\mathcal{A}$.

Call-Enabled: For every finite execution E in $\mathcal{A}$ and every call step s, if $E \circ \langle s \rangle$ is a mailbox execution then $E \circ \langle s \rangle$ is in $\mathcal{A}$.

Check-Correct: For every finite execution E in $\mathcal{A}$ and every finite sequence S of steps beginning with a call of *check* and ending with the subsequent return from *check* such that $E \circ S$ is in $\mathcal{A}$:

- If $N(E) > 0$ then the return from *check* returns TRUE

[2]These properties follow from but do not imply the mailbox problem specification, because they do not require *deliver* to increment the postman's counter in a single atomic step.

- If $M(E \circ S) = 0$ then the return from *check* returns FALSE

Non-blocking: Every infinite execution E in $\mathcal{A}$ has infinitely many return steps.

Define an *execution of an operation* to be a finite sequence of steps with which some process performs the operation. More precisely, it is any sequence of steps of a single process that could describe an execution of the operation if they were interleaved with suitable steps of the other process. We identify a procedure P with the operation that begins with the call of P and ends with the return. A *solo execution in $\mathcal{A}$ of operation op from an execution S* is an execution T of operation *op* such that $S \circ T \in \mathcal{A}$. We say that a mailbox execution E is *complete* if E is finite and the number of call steps in E equals the number of return steps in E. If P is a procedure, we say that a mailbox execution E is *P-enabled* if $E \circ \langle \text{call } P \rangle$ is a mailbox execution. The following properties are immediate from these definitions and the four properties above.

Solo Execution Existence: For every mailbox procedure P and every mailbox execution E in $\mathcal{A}$, if E is P-enabled then there is a solo execution in $\mathcal{A}$ of P from E.

Solo Check Result: For every mailbox execution E in $\mathcal{A}$, every solo execution in $\mathcal{A}$ of *check* from E returns the Boolean value $N(E) > 0$.

Our proofs are based on reduction to the consensus problem, which is defined in terms of a procedure *propose* that takes a value as parameter and returns a value. The following properties must hold for every execution of an algorithm that solves the consensus problem:

Uniform-Agreement: No two processes return different values.

Validity: If a process returns v, then some process calls *propose*(v).

Termination: If a process takes infinitely many steps, then it returns.

5.2 Impossibility with Two Single-Writer One-Bit Flags

Theorem 1 *There is no non-blocking algorithm that solves the mailbox problem with the access restriction when Flag is an array with two one-bit single-writer atomic registers.*

PROOF: The proof is by contradiction. Let $\mathcal{A}$ be a non-blocking algorithm that solves the mailbox problem with the access restriction using two one-bit single-writer atomic registers. We will use $\mathcal{A}$ to construct a wait-free consensus algorithm for two processes, contradicting the impossibility result of Fischer, Lynch, and Paterson [10, Chapter 12].

Throughout this proof, let $Flag[0]$ and $Flag[1]$ denote the two one-bit single-writer atomic registers used by algorithm $\mathcal{A}$. Through step 11 below, "execution" means a finite execution in $\mathcal{A}$. For an execution E, we let $Flag[i](E)$ be the value of $Flag[i]$ after the last step of E.

1. We may assume that each process of $\mathcal{A}$ is deterministic, meaning that for each execution E of $\mathcal{A}$ and each process i, there is at most one step s of process i such that $E \circ \langle s \rangle$ is an execution of $\mathcal{A}$.
   PROOF: We can remove any nondeterministic choice by arbitrarily specifying which possible action a process performs. Since the algorithm must be correct regardless of which choice is made, correctness of the original algorithm implies correctness of the deterministic one.

2. (a) Each process must be allowed to write to $Flag[0]$ or $Flag[1]$.
   (b) We can assume that process i is allowed to write to $Flag[i]$.
   PROOF: (a) If a process is not allowed to write to $Flag[0]$ or $Flag[1]$, it is easy to construct a scenario in which it executes *remove* or *deliver* and the flags do not change, causing a subsequent *check* to return an incorrect result.
   (b) An algorithm in which process i can write to $Flag[1-i]$ is easily transformed into one in which process i can write to $Flag[i]$.

DEFINITION: An execution E is c-checkable if E is complete and *check*-enabled.

3. For any 1-bit values F_0 and F_1, there is a Boolean $C(F_0, F_1)$ such that for every c-checkable execution E with $Flag[i](E) = F_i$ for each i, every solo execution in $\mathcal{A}$ of *check* from E returns the value $C(F_0, F_1)$.
   PROOF: By the access restriction, an execution of *check* cannot write to shared variables and it returns a value that depends only on the values read from each $Flag[i]$. By step 1, for each F_0 and F_1 there is a single value that can be returned.

4. $C(Flag[0](E), Flag[1](E)) = (N(E) > 0)$, for every c-checkable execution E.
   PROOF: By the Solo Check Result property and definition of C.

5. For every c-checkable execution E with $N(E) = 0$ and every solo execu-

tion S in $\mathcal{A}$ of *deliver* from E,
$Flag[0](E \circ S) = Flag[0](E)$
$Flag[1](E \circ S) = 1 - Flag[1](E)$
PROOF: Since E is a c-checkable execution, $E \circ S$ is a c-checkable execution. By step 4, $C(Flag[0](E \circ S), Flag[1](E \circ S)) \neq C(Flag[0](E), Flag[1](E))$. By step 2, $Flag[0](E \circ S) = Flag[0](E)$, since S contains only steps of process 1. Hence, $Flag[1](E \circ S) \neq Flag[1](E)$, and thus $Flag[1](E \circ S) = 1 - Flag[1](E)$ since $Flag[1]$ is either 0 or 1.

6. For every c-checkable execution E with $N(E) = 1$, every solo execution S in $\mathcal{A}$ of *check* from E (that by step 4 returns TRUE), and every solo execution T in $\mathcal{A}$ of *remove* from $E \circ S$,
$Flag[0](E \circ S \circ T) = 1 - Flag[0](E)$
$Flag[1](E \circ S \circ T) = Flag[1](E)$
PROOF: Analogous to the proof of step 5.

DEFINITION: $I_i \triangleq Flag[i](\langle\,\rangle)$, the initial value of $Flag[i]$, for $i = 0, 1$

7. $C(I_0, I_1) = C(1 - I_0, 1 - I_1) =$ FALSE
$C(I_0, 1 - I_1) = C(1 - I_0, I_1) =$ TRUE

7.1. Choose step sequences S_1, S_2, S_3, and S_4 such that
- S_1 is a solo execution in $\mathcal{A}$ of *deliver* from $\langle\,\rangle$;
- S_2 is a solo execution in $\mathcal{A}$ of *check* from S_1 that returns TRUE;
- S_3 is a solo execution in $\mathcal{A}$ of *remove* from $S_1 \circ S_2$; and
- S_4 is a solo execution in $\mathcal{A}$ of *deliver* from $S_1 \circ S_2 \circ S_3$.

Then S_1, $S_1 \circ S_2 \circ S_3$ and $S_1 \circ S_2 \circ S_3 \circ S_4$ are c-checkable executions.
PROOF: The Solo Execution Existence and Solo Check Result properties imply that S_1, S_2, S_3, and S_4 exist. Clearly, S_1, $S_1 \circ S_2 \circ S_3$ and $S_1 \circ S_2 \circ S_3 \circ S_4$ are c-checkable executions.

7.2. (a) $N(\langle\,\rangle) = 0$
(b) $N(S_1) = 1$
(c) $N(S_1 \circ S_2 \circ S_3) = 0$
(d) $N(S_1 \circ S_2 \circ S_3 \circ S_4) = 1$
PROOF: Immediate from step 7.1 and definition of N.

7.3. (a) $Flag[0](\langle\,\rangle) = I_0$ and $Flag[1](\langle\,\rangle) = I_1$
(b) $Flag[0](S_1) = I_0$ and $Flag[1](S_1) = 1 - I_1$
(c) $Flag[0](S_1 \circ S_2 \circ S_3) = 1 - I_0$ and $Flag[1](S_1 \circ S_2 \circ S_3) = 1 - I_1$
(d) $Flag[0](S_1 \circ S_2 \circ S_3 \circ S_4) = 1 - I_0$ and $Flag[1](S_1 \circ S_2 \circ S_3 \circ S_4) = I_1$
PROOF:
(a) By definition.

(b) By steps 5 and 7.2(a).
(c) By steps 6 and 7.2(b).
(d) By steps 5 and 7.2(c).

7.4. Q.E.D.
PROOF: Step 7 follows from steps 7.2, 7.3, and 4.

8. Choose a solo execution S_0 in $\mathcal{A}$ of *deliver* from $\langle\rangle$. Then S_0 is a c-checkable execution, $Flag[0](S_0) = I_0$, and $Flag[1](S_0) = 1 - I_1$.
PROOF: S_0 exists by the Solo Execution Existence property, and it is a c-checkable execution. The values of $Flag[i](S_0)$ follow from step 5.

DEFINITION: Let a *check+remove* operation consist of a call to *check* followed by a call to *remove* if the *check* returns true.

9. If S is a solo execution in $\mathcal{A}$ of *check+remove* from S_0, then S performs the *remove* and $Flag[0](S_0 \circ S) = 1 - I_0$.
PROOF: That S performs the *remove* follows from the Solo Check Result property, since $N(S_0) = 1$, and steps 8 and 6 then imply that $Flag[0](S_0 \circ S) = 1 - I_0$.

10. If S is a solo execution in $\mathcal{A}$ of *deliver* from S_0, then $Flag[1](S_0 \circ S) = 1 - I_1$.
PROOF: Clearly, $S_0 \circ S$ is a c-checkable execution. Since *deliver* does not modify $Flag[0]$, step 8 implies $Flag[0](S_0 \circ S) = I_0$. Step 6 then implies $C(I_0, Flag[1](S_0 \circ S)) = \text{TRUE}$, so step 7 implies $Flag[1](S_0 \circ S) = 1 - I_1$.

11. If S is a sequence of steps containing a single execution of *deliver* and a single execution of *check+remove*, and $S_0 \circ S$ is an execution, then either
$Flag[0](S_0 \circ S) = I_0$ and $Flag[1](S_0 \circ S) = 1 - I_1$, or
$Flag[0](S_0 \circ S) = 1 - I_0$ and $Flag[1](S_0 \circ S) = I_1$.
PROOF: $S_0 \circ S$ is a c-checkable execution and simple arithmetic shows that $N(S_0 \circ S) > 0$, so this follows from steps 4 and 7.

Henceforth, "execution" no longer refers only to finite executions in $\mathcal{A}$ but to executions of $\mathcal{A}$ or of the following algorithm $\mathcal{C}$.

DEFINITION: Let $\mathcal{C}$ be the (set of executions that define the) following two-process algorithm. There is a copy of algorithm $\mathcal{A}$ initialized to the state at the end of S_0. Each process i has a procedure *propose*(v) that performs the following operations:

c1. process i writes v to a shared register $V[i]$ initialized to $\perp$;
c2. process i uses $\mathcal{A}$ to execute either *check+remove* if $i = 0$ or *deliver* if $i = 1$;

c3. if process i ends up with value $1 - I_i$ in $Flag[i]$ then process i reads $V[i]$ and returns its value; else process i reads $V[1-i]$ and returns its value.

The definition of $\mathcal{C}$ means that if G is an execution in $\mathcal{C}$ and S is the sequence of steps from G performed (by both processes) in operation c2, then $S_0 \circ S$ is an execution in $\mathcal{A}$, which we call $\pi(G)$.

12. Algorithm $\mathcal{C}$ solves the consensus problem.
PROOF: Let G be an execution of $\mathcal{C}$, and let v_j be the value with which process j calls *propose*, or $\perp$ if the process does not call *propose*. We show that $\mathcal{C}$ satisfies the Uniform-Agreement, Validity, and Termination properties of consensus (Section 5.1, page 26).

DEFINITION: For $i = 0, 1$, let G^i be the prefix of G up to when process i returns from *check+remove* or *deliver*, or let G^i equal $\perp$ if process i never returns from these operations in G.

For each process i, the following properties of execution G follow directly from the definition of $\mathcal{C}$.

CF1. If process i returns from *propose*, then $G^i \neq \perp$.

CF2. If process i returns w_i from *propose*, then w_i is a value that process i reads from $V[i]$ or $V[1-i]$ after the steps of G^i.

CF3. v_i and $\perp$ are the only values that $V[i]$ can equal during the execution, and it equals $\perp$ only if process i has not yet completed operation c1.

12.1. If process i returns some value w_i from *propose* in G, then $w_i \neq \perp$.
PROOF: We assume $w_i = \perp$ and obtain a contradiction. By CF1, $G^i \neq \perp$. By CF2, w_i is a value read from $V[i]$ or $V[1-i]$. Since process i can obviously not read $V[i] = \perp$ in operation c3, w_i must be read from $V[1-i]$. CF3 then implies that G^i does not contain a complete execution of operation c1 by process $1-i$. Hence, $\pi(G^i) = S_0 \circ S$ for some solo execution S in $\mathcal{A}$ of *check+remove* or *deliver* (by process i). Step 9 or 10 (depending on the value of i) implies $Flag[i](S \circ S_0) = 1 - I_i$. Since $Flag[i]$ is changed only in the execution of c2 by process i, this implies that i reads $Flag[i] = 1 - I_i$ in operation c3, so it does not read $V[1-i]$. This provides the required contradiction.

12.2. G satisfies the Validity property.
PROOF: Clear from CF2, CF3, and 12.1.

12.3. G satisfies the Uniform-Agreement property.

PROOF: It suffices to assume process 0 returns w_0 and process 1 returns w_1 from their executions of *propose* in G and prove $w_0 = w_1$. Let i be the last of the processes to finish operation c2. By CF1, $G^i \neq \perp$; by definition of the algorithm, $\pi(G^i) = S_0 \circ S$ for some sequence S that contains a single execution of *deliver* and a single execution of *check*+*remove*, such that $S_0 \circ S$ is an execution in $\mathcal{A}$. By step 11, either (1) $Flag[0](\pi(G^i)) = I_0$ and $Flag[1](\pi(G^i)) = 1 - I_1$ or (2) $Flag[0](\pi(G^i)) = 1 - I_0$ and $Flag[1](\pi(G^i)) = I_1$. Since $Flag[j]$ is changed only in operation c2 of process j, the value of $Flag[j]$ that each process j reads in c3 is $Flag[j](\pi(G^i))$. By CF3 and definition of operation c3, in case (1) we have $w_0 = w_1 = v_1$, and in case (2) we have $w_0 = w_1 = v_0$.

12.4. G satisfies the Termination property.
PROOF: We must show that each process i completes the execution of *propose* in a finite number of steps. Process i clearly completes operations c1 and c3 in a finite number of steps. It completes c2 in a finite number of steps because $\mathcal{A}$ is a non-blocking algorithm and i performs either one or two executions of procedures of $\mathcal{A}$.

12.5. Q.E.D.
PROOF: Steps 12.2–12.4 show that G satisfies the properties required of a consensus algorithm.

13. Q.E.D.
PROOF: Step 12 contradicts the well-known result that there is no wait-free consensus algorithm in which processes communicate only by shared read/write variables [10, Chapter 12].

5.3 Impossibility with Multi-Writer One-Bit Flag

Theorem 2 *There is no non-blocking algorithm that solves the mailbox problem with the access restriction when Flag is a one-bit multi-writer atomic register.*

PROOF: The proof is by contradiction. Let $\mathcal{A}$ be a non-blocking algorithm that solves the mailbox problem with the access restriction when $\mathcal{A}$'s variable *Flag* is a one-bit multi-writer atomic register. As in the proof of Theorem 1, we will use $\mathcal{A}$ to solve fault-tolerant consensus, which contradicts the impossibility result of Fischer, Lynch, and Paterson [10, Chapter 12].
Through step 9 below, "execution" means a finite execution in $\mathcal{A}$. For an execution E, we let $Flag(E)$ be the value of *Flag* after the last step of E.

1. We may assume that each process of $\mathcal{A}$ is deterministic, meaning that for each execution E of $\mathcal{A}$ and each process i, there is at most one step s of process i such that $E \circ \langle s \rangle$ is an execution of $\mathcal{A}$.
   PROOF: The same as for step 1 of Theorem 1.

2. We may assume that every time a process intends to write some value v to *Flag* in $\mathcal{A}$, the process first reads *Flag* and, if it finds the value to be v, then it does not perform the write.
   PROOF: We modify algorithm $\mathcal{A}$ by replacing each write of *Flag* with an operation that performs a read and then writes only if it would change the value. Because *Flag* is an atomic register, it is easy to see that this modification preserves correctness.

3. For any 1-bit value F, there is a Boolean $C(F)$ such that in every execution that includes an execution of *check*, if *Flag*=F throughout the execution of *check*, then that execution of *check* returns $C(F)$.
   PROOF: By the mailbox problem's access restriction, an execution of *check* cannot write to shared variables, and it returns a value that depends only on the values read from *Flag*. By assumption, the only value it reads is F. By step 1, for each F there is a single value that can be returned in this case.

DEFINITION: An execution E is c-checkable iff E is complete and *check*-enabled.

4. $C(\mathit{Flag}(E)) = (N(E) > 0)$, for every c-checkable execution E.
   PROOF: By the Solo Check Result property.

5. If E is a c-checkable execution, $N(E) = 0$, and S is a solo execution in $\mathcal{A}$ of *deliver* from E, then $\mathit{Flag}(E \circ S) = 1 - \mathit{Flag}(E)$.
   PROOF: Clearly, $E \circ S$ is a c-checkable execution. By step 4, $C(\mathit{Flag}(E \circ S)) \neq C(\mathit{Flag}(E))$. Hence, $\mathit{Flag}(E \circ S) \neq \mathit{Flag}(E)$, so $\mathit{Flag}(E \circ S) = 1 - \mathit{Flag}(E)$ because *Flag* is either 0 or 1.

DEFINITION: $I \triangleq \mathit{Flag}(\langle\,\rangle)$, the initial value of *Flag*.

6. $C(I)$=FALSE and $C(1 - I)$=TRUE
   PROOF: By the Solo Execution Existence property, we can choose a solo execution S in $\mathcal{A}$ of *deliver* from $\langle\,\rangle$. Clearly, S is a c-checkable execution. Since $N(\langle\,\rangle) = 0$ and I is defined to equal $\mathit{Flag}(\langle\,\rangle)$, step 4 implies $C(I) = C(\mathit{Flag}(\langle\,\rangle))$ = FALSE. By step 5, $\mathit{Flag}(S) = 1 - \mathit{Flag}(\langle\,\rangle) = 1 - I$. Since $N(S) = 1$, step 4 then implies $C(1 - I) = C(\mathit{Flag}(S))$ = TRUE.

7. Choose a solo execution S_0 in $\mathcal{A}$ of *deliver* from $\langle\,\rangle$. Then S_0 is a c-checkable execution and $\mathit{Flag}(S_0) = 1 - I$.

PROOF: S_0 exists by the Solo Execution Existence property, and clearly it is a c-checkable execution. Step 5 implies $Flag(S_0) = 1 - I$.

DEFINITION: Let a *check*+*remove*+*check* operation consist of the following, in order: (1) a call to *check*, (2) a call to *remove* if the *check* returns true, and (3) a call to *check*.

8. If S is a solo execution in $\mathcal{A}$ of *check*+*remove*+*check* from S_0, then S performs the *remove*.
   PROOF: By the Solo Check Result property, since $N(S_0) = 1$.

9. If S is a solo execution in $\mathcal{A}$ of *deliver* from S_0, then S has no steps that write I to *Flag*.
   PROOF: We assume S has a step that writes I to *Flag* and obtain a contradiction. Let S_1 be the prefix of S up to and including that step. Then $Flag(S_0 \circ S_1) = I$, and $S_0 \circ S_1$ is *check*-enabled (since S_0 is c-checkable). By the Solo Execution Existence property, we can choose a solo execution S_2 in $\mathcal{A}$ of *check* from $S_0 \circ S_1$. Since $N(S_0 \circ S_1) = 1$, by the Check-Correct property (Section 5.1, page 25), the *check* execution returns TRUE. But by steps 3 and 6, the *check* execution returns $C(Flag(S_0 \circ S_1)) = C(I) =$ FALSE, which is the required contradiction.

Henceforth, "execution" no longer refers only to finite executions in $\mathcal{A}$ but to executions of $\mathcal{A}$ or of the following algorithm $\mathcal{C}$.

DEFINITION: Let $\mathcal{C}$ be the (set of executions that define the) following two-process algorithm. There is a copy of algorithm $\mathcal{A}$ initialized to the state at the end of S_0. Each process i has a procedure *propose*(v) that performs the following operations:

Process 0:

c01. write v to a shared register $V[0]$ initialized to $\perp$;
c02. use $\mathcal{A}$ to execute *check*+*remove*+*check*;
c03. if the second *check* in operation c02 returned FALSE
     then read $V[0]$ and return its value,
     else read $V[1]$ and return its value

Process 1:

c11. write v to a shared register $V[1]$ initialized to $\perp$;
c12. use $\mathcal{A}$ to execute *deliver* until either
   (a) *deliver* returns, or
   (b) the process is about to write $1-I$ to *Flag*, in which case execution of *deliver* is stopped *before* the write occurs.

c13. if (a) occurs then read $V[1]$ and return its value
if (b) occurs then read $V[0]$ and return its value

Formally, the definition of $\mathcal{C}$ means that if G is an execution in $\mathcal{C}$ and S is the sequence of steps from G performed (by both processes) in operation c02 and c12, then $S_0 \circ S$ is an execution in $\mathcal{A}$, which we call $\pi(G)$.

10. Algorithm $\mathcal{C}$ solves the consensus problem.

PROOF: Let G be an execution of $\mathcal{C}$, and let v_j be the value with which process j calls *propose*, or $\perp$ if the process does not call *propose*. We show that $\mathcal{C}$ satisfies the Uniform-Agreement, Validity, and Termination properties of consensus (Section 5.1, page 26).

DEFINITION: For $i = 0, 1$, let G^i be the prefix of G up to when process i stops executing $\mathcal{A}$, or let G^i equal $\perp$ if process i never starts or never stops executing $\mathcal{A}$ in G.

For each process i, the following properties of execution G follow directly from the definition of $\mathcal{C}$.

CF1. If process i returns from *propose*, then $G^i \neq \perp$.

CF2. If process i returns w_i from *propose*, then w_i is a value that process i reads from $V[i]$ or $V[1-i]$ after the steps of G^i.

CF3. v_i and $\perp$ are the only values that $V[i]$ can equal during the execution, and it equals $\perp$ iff process i has not yet written to $V[i]$ in operation c01 or c11.

10.1. If process i returns some value w_i from *propose* in G, then $w_i \neq \perp$.
PROOF: We assume $w_i = \perp$ and obtain a contradiction. By CF1, $G^i \neq \perp$. By CF2, w_i is a value read from $V[i]$ or $V[1-i]$. Since process i can obviously not read $V[i] = \perp$ in operation c03 or c13, w_i must be read from $V[1-i]$. By CF3, process $1-i$ does not write to $V[1-i]$ in G^i, so process i completes operation c02 or c12 without process $1-i$ having started operation c12 or c02. Thus, $\pi(G^i) = S_0 \circ S$ where S is either a solo execution in $\mathcal{A}$ of *check*+*remove*+*check* from S_0 or the prefix[3] of a solo execution in $\mathcal{A}$ of *deliver* from S_0. Note that $Flag(G^i) = Flag(S \circ S_0)$ because *Flag* is not written outside algorithm $\mathcal{A}$. We now consider the two possible values of i:

10.1.1. CASE: $i = 0$
Then S is a solo execution in $\mathcal{A}$ of operation *check*+*remove*+*check* from S_0. By step 8, this operation performs the *remove*. By the

[3]Remember that $\mathcal{C}$ may stop *deliver* in the middle of its execution.

Solo Check Result property, the last *check* execution in $S_0 \circ S$ returns FALSE. By definition of $\mathcal{C}$, process 0 reads $V[0]$ and returns its value (in execution G). CF3 and the fact that process i writes to $V[i]$ in G^i implies $w_0 = v_0 \neq \perp$, which is the required contradiction that proves step 10.1 for $i = 0$.

10.1.2. CASE: $i = 1$

In S, process 1 executes *deliver* until one of these cases occur: (a) process 1 returns from *deliver* or (b) process 1 is about to write $1 - I$ to *Flag*. Hence, in S, process 1 does not write $1 - I$ to *Flag*. Step 9 implies that process 1 does not write I to *Flag* either, and we argued above that process 0 (process $1-i$) does not start operation c02 and hence does not write to *Flag* in G^1. Thus, if process 1 reads *Flag* in G^1, it obtains $Flag(S_0)$, which equals $1 - I$ by step 7. By step 2, a process writes $1 - I$ to *Flag* only if it reads I from *Flag*, which does not happen. Thus, case (b) above cannot occur. In case (a), the definition of $\mathcal{C}$ implies that process 1 reads $V[1]$ and returns its value. Thus, CF3 and the fact that process 1 writes to $V[1]$ in G^1 imply $w_1 = v_1 \neq \perp$, which is the required contradiction that proves step 10.1 for $i = 1$.

10.2. G satisfies the Validity property.

PROOF: Clear from CF2, CF3, and 10.1.

10.3. G satisfies the Uniform-Agreement property.

PROOF: It suffices to assume process 0 returns w_0 and process 1 returns w_1 from their executions of *propose* in G and prove $w_0 = w_1$. By CF1, $G^i \neq \perp$ for $i = 0, 1$. Moreover, $\pi(G) = S_0 \circ S$ for some sequence S such that $S_0 \circ S$ is in $\mathcal{A}$, and S contains a single execution of *check*+*remove*+*check* and the prefix of a single execution of *deliver*.

10.3.1. It is impossible to have $w_0 = v_0$ and $w_1 = v_1$.

PROOF: We assume $w_0 = v_0$ and $w_1 = v_1$ and obtain a contradiction. By definition of $\mathcal{C}$, in G process 0 executes *check*+*remove*+*check* and the second *check* returns FALSE. It is impossible that $Flag = 1 - I$ throughout the execution of the second *check*, otherwise it would have returned TRUE by steps 3 and 6. Thus, $Flag = I$ at some point in the execution of the second *check*. Process 0 does not write $1 - I$ to *Flag* in *check* by the mailbox problem's access restriction, and process 1 does not write $1 - I$ to *Flag* during *deliver* by definition of $\mathcal{C}$. Thus $Flag(S_0 \circ S) = I$. By the definition of $\mathcal{C}$ and the assumption $w_1 = v_1$, process 1 completes its *deliver* operation. Thus, $S_0 \circ S$

contains two *deliver* executions and one *remove* execution. Since the second *check* returns FALSE, execution $S_0 \circ S$ is c-checkable, so by step 4 $Flag(S_0 \circ S) = I$ implies $C(I) =$ TRUE. This contradicts step 6.

10.3.2. It is impossible to have $w_0 = v_1$ and $w_1 = v_0$.

PROOF: We assume $w_0 = v_1$ and $w_1 = v_0$, and we obtain a contradiction. By definition of algorithm $\mathcal{C}$, in G process 0 executes *check*+*remove*+*check* and the second *check* returns TRUE, while process 1 executes *deliver* and stops just before it is about to write $1 - I$ to *Flag*. By the Solo Execution Existence property, we can choose S_1, S_2, and S_3 as follows:

- S_1 is a solo execution in $\mathcal{A}$ of *remove* from $S_0 \circ S$ (by process 0);
- S_2 is a single step by process 1 that writes $1 - I$ to *Flag* (the step it was about to take at the end of $S_0 \circ S$); and
- S_3 is a solo execution in $\mathcal{A}$ of *check* from $S_0 \circ S \circ S_1 \circ S_2$ (by process 0).

Then, $M(S_0 \circ S \circ S_1 \circ S_2 \circ S_3) = 0$ by definition of M (Section 5.1, page 25) and of S_0, S, S_1, S_2, and S_3. Thus, by the Check-Correct property (Section 5.1, page 25), the *check* in S_3 returns FALSE. However, $Flag(S_0 \circ S \circ S_1 \circ S_2) = 1 - I$ by definition of S_2, so steps 3 and 6 imply that the *check* in S_3 returns TRUE. This is the required contradiction.

10.3.3. Q.E.D.

PROOF: CF3 and step 10.1 imply that each w_i equals some v_j. Steps 10.3.1 and 10.3.2 rule out the two possibilities in which $w_0 \neq w_1$, proving $w_0 = w_1$.

10.4. G satisfies the Termination property.

PROOF: We must show that each process i completes the execution of *propose* in a finite number of steps. Operations c01, c03, c11, and c13 clearly terminate. Operations c02 and c12 also terminate because $\mathcal{A}$ is a non-blocking algorithm and there are at most three operation executions of $\mathcal{A}$ in G. Hence, the execution of *propose* completes in a finite number of steps.

10.5. Q.E.D.

PROOF: Steps 10.2–10.4 show that G satisfies the properties required of a consensus algorithm.

11. Q.E.D.

PROOF: Step 10 contradicts the well-known result that there is no wait-free consensus algorithm in which processes communicate only by shared read/write variables [10, Chapter 12].

6 Conclusion

6.1 Related Work

The mailbox problem is the producer-consumer problem with an unbounded number of buffers, the postman being the producer and the owner's *check* operation being the consumer's test if there is a buffer to consume. With a bounded number of buffers, the producer also has a *check* operation that tests if there is an empty buffer. All previous solutions to the N-buffer producer-consumer problem that we know of with single-writer shared variables require each process's *check* operation to read flags that can hold at least $N + 1$ values [7].

We can use our bounded signaling algorithm to implement an N-buffer producer-consumer algorithm in which the shared variables read by *check* have size independent of N. We simply let the *read* procedure return a value indicating the result of both *check* operations. However, because the *write* operation of our signaling algorithm is non-blocking, not wait-free, the resulting producer-consumer algorithm is wait-free only with an $O(N)$ bound on waiting time. We can obtain a wait-free algorithm with constant waiting time as follows by using two instances of a wait-free mailbox algorithm. In the first instance, the postman is the producer and the letters in the mailbox represent filled buffers; in the second, the owner is the producer and the letters represent empty buffers. Initially, the first mailbox has no letters and the second has N letters. It is easy to fill in the details.

Ellen et al. [2] define a SNZI object that is similar to a mailbox. It consists of an internal counter accessed with *increment*, *decrement*, and *test-for-nonzero* operations. (They call these operations *arrive*, *depart*, and *query*, respectively.) They give a linearizable implementation for any number of processes in which the *test-for-nonzero* operation is efficient, just reading a single bit. At first glance, it appears that there is a trivial solution to the mailbox problem using a SNZI object: just implement *deliver* with *increment*, *check* with *test-for-nonzero*, and *remove* with *decrement*. However, this doesn't work because of the following restriction on the use of a SNZI object: if the counter is zero and *increment* is called, then *decrement* can be called only after *increment* returns. For example, if initially the postman calls *increment* and the owner calls *test-for-nonzero* and obtains TRUE, then

this restriction requires the owner to wait for the postman's call to return before she can call *decrement*. (The restriction does not affect the intended applications of SNZI objects, in which a process calls *decrement* only to cancel its own previous *increment*.) The SNZI implementation also differs from our algorithms because it uses compare-and-swap instructions rather than just read and write.

6.2 Summary and Open Problems

We have introduced the mailbox problem, which abstracts the use of interrupts to synchronize threads or hardware devices. The need for an efficient mechanism to determine if an interrupt has occurred is expressed by the mailbox problem's access restriction that *check* reads only a small amount of shared memory and returns a value that depends only on what it reads. We presented a bounded wait-free algorithm using two flags that assume 14 values each, and we gave impossibility results for 1-bit flags. We do not know if there is an algorithm with 1-bit flags under a weaker access restriction that allows the value returned by *check* to depend on process-local state that it remembers across executions of the procedure. Another open question is whether the space efficiency can be improved. Our algorithms use $\Theta(n \log n)$ bits of shared memory, where n is the number of executions of *deliver* and *remove*. We conjecture that there is a solution using logarithmic space.

We have also generalized the mailbox problem to the signaling problem, in which the two processes can each write its own value and read a function F of the two values. The size of the shared state accessed by the *read* operation must depend only on the size of the result obtained by applying F to the values, not on the size of the values themselves. We presented a non-blocking solution to this problem. We don't know if a wait-free solution exists.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: scalable nonzero indicators. In *PODC '07: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 13–22, New York, NY, USA, August 2007. ACM.

[3] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[4] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *Proceedings of the Fourteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 13–26, Munich, January 1987. ACM.

[5] Leslie Lamport. The pluscal algorithm language. URL `http://research.microsoft.com/users/lamport/tla/pluscal.html`. The page can also be found by searching the Web for the 25-letter string obtained by removing the "-" from `uid-lamportpluscalhomepage`.

[6] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[7] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[8] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[9] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[10] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, California, 1995.

[11] Jerome H. Saltzer. Traffic control in a multiplexed computer system. Project MAC Technical Report MAC-TR-30, M.I.T., June 1966.

[12] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Berlin, Heidelberg, New York, September 1999. Springer-Verlag. 10th IFIP wg 10.5 Advanced Research Working Conference, CHARME '99.