On the mailbox problem

Uri Abraham and Gal Amram
Departments of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, Israel.

May 11, 2014

Abstract

The Mailbox Problem was described and solved by Aguilera, Gafni, and Lamport in [4] with an algorithm that uses two flag registers that carry 14 values each. An interesting problem that they ask is whether there is a mailbox algorithm with smaller flag values. We give a positive answer by describing a mailbox algorithm with 6 and 4 values in the two flag registers.

1 Introduction: the mailbox problem

The Mailbox Problem is a theoretical synchronization problem that arises from analyzing the situation in which a processor must cater to occasional requests from some device. The problem, as presented (and solved) in [4] requires the implementation of three operations: deliver, check, and remove. The device executes a deliver operation whenever it wants to get the processor’s attention, and the processor executes from time to time check operations to find out if there are any unhandled device requests. After receiving a positive answer for its check operation the processor executes a remove operation to find-out the nature of the request and to clear the interrupt controller. It is required that a check operation C returns a positive answer if and only if the number of deliver occurrences that precede C is strictly greater than the number of remove operations executed before C. The Mailbox Problem is to design a deliver/check/remove algorithm in which the check operation is as efficient as possible, namely that it employs bounded registers (called “flags”) that are as small as possible.

In [4] the problem is presented first informally by means of a story involving two processes, a postman (which is the device) and a home owner (the processor), in which the postman delivers its letters, and the owner removes them one by one every time she approaches the mailbox. The problem is to find an algorithm that ensures that the home owner approaches her mailbox if and only if it is nonempty. The check function tells the
home-owner whether the mailbox is empty or not, and she approaches her mailbox only after receiving a “nonempty” response from a check execution. As noted in [4], depending on the assumptions made on the communication between the device and processor the mailbox problem can be extremely easy or surprisingly difficult. The following very easy solution (figure 1) shows that if the homeowner process can read an unbounded register then the mailbox problem becomes trivial. In this unbounded algorithm the postman adds its letter to \( Q \) (the queue of requests), and then it writes on its \( D_{\text{num}} \) register the number of letters so far added. The home-owner, in executing her check operation, reads register \( D_{\text{num}} \) to know how many letters were deposited, and determines the number of messages removed so far by consulting her remove-number local variable \( rn \), and then she concludes that the mailbox is nonempty if the number of letters deposited exceeds the number of letters removed.

Another easy solution to the mailbox problem can be obtained with stronger communication objects. For example, a simple algorithm is suggested in [4] in which the postman and home-owner employ a flag at the mailbox. The postman can atomically (in a single step) deliver mail to the box and raise the flag, and the owner atomically removes mail from the box and lowers the flag. The mailbox problem becomes highly non-trivial when limitations are imposed on the communication devices. Specifically, Aguilera et al. require in [4], for efficiency reasons, that the mailbox solutions use only the simplest possible means, and the check operation (which is possibly invoked at higher frequency) should access only a bounded register. As formulated in [4], the mailbox problem asks for solutions that satisfy the following requirements:

1. Only registers with read/write actions can be employed.

2. Whereas the deliver and remove operations are allowed unbounded registers, the home-owner can only read bounded value registers in check operation executions.

---

1In an interesting note in his list of publications home-page, Lamport tells that when he first thought about this problem he believed it has no solution under these requirements.
3. Moreover, in her check operations the home-owner cannot use persistent local variables, that is variables that retain their values from one invocation of the operation to the following one.

4. The algorithms for the three operations (deliver, check, and remove) are bounded wait-free.

A solution is presented in [4] in which each of the two processes uses unbounded and bounded registers (the bounded registers are called ‘flags’) and the check operation (as required) decides on the value to return by reading only the bounded flag registers. The algorithm of [4] needs 14 values in each of the two flag registers, and a question is posed there if leaner solutions exist. We give a positive answer here by describing an algorithm in which the flag registers of the postman and the home owner carry 6 and 4 values in each of the flag registers; that is 10 values in total as opposed to 28 values in [4].

We shall describe now in more details and greater formality the mailbox problem of [4]. The mailbox problem assumes two serial processes, a postman process and a home-owner process, and their mission is to implement three operations: deliver(), check(), and remove(). deliver() takes a letter as parameter, check returns a boolean value, and remove() returns a letter. It is required that the algorithm is bounded wait-free, which means that each operation completes before the process executing it has taken \( k \) (atomic) steps, for some fixed constant \( k \), irrespectively of what steps the other processes take.

The postman and the home-owner are serial processes which operate concurrently. The postman executes forever the following routine: he gets a letter \( \ell \) and (if the letter is addressed to the home owner) he executes the deliver(\( \ell \)) operation which adds the letter to the owner’s mailbox. So it is quite possible that the total number of deliver operations is finite. The home-owner process executes forever the following routine:

\[
\begin{align*}
\text{repeat} \\
\quad v &:= \text{check}() \\
\text{until } v &= \text{true;} \\
\quad \text{remove}(). \\
\end{align*}
\]

Thus the check operations are executed ad infinitum, although it is possible that only a finite number of them are positive (return the value true).

The safety property is expressed in [4] by first stating its sequential specification, and then requiring that a linearization exists which satisfies this sequential specification. This is the well-known approach to linearizability as defined by Herlihy and Wing in [6]. The following is the formulation in [4] for the sequential specification:

If the owner and postman never execute concurrently, then the value returned by an execution of check is true if and only if there are more deliver than remove executions before this execution of check.

3
1. The events are partitioned into deliver, check, and remove events, and are totally ordered by $\prec$ in the order-type of the natural numbers (if not finite).

2. For every check event $C$, $Val(C) \in \{\text{true}, \text{false}\}$. For every remove event $R$ there is a check event $C$ such that $Val(C) = \text{true}$, $C \prec R$ and there is no check or remove event $X$ with $C \prec X \prec R$.

3. For every check event $C$ let the removal number, $\text{removal\_num}(C)$, be the number of remove events $R$ with $R \prec C$, and let $\text{deliver\_num}(C)$ be the number of deliver events $D$ such that $D \prec C$. Then

$$Val(C) = \text{"removal\_num}(C) < \text{deliver\_num}(C),$$

that is to say the boolean value of $C$ is true iff the number of deliver events that precede $C$ exceeds the number of letters that were removed by remove events that precede $C$.

4. If $D_1 \prec D_2 \cdots$ and $R_1 \prec R_2 \cdots$ are the enumerations in increasing order of the deliver and of the remove events, then for every $i$ the letter removed by $R_i$ is the letter delivered by $D_i$.

Figure 2: Linear mailbox specification.

To this specification we add the obvious requirement that a queue is implemented, namely that the letters removed are those delivered, and that the letters are removed in the order of delivery. The original mailbox paper [4] mentions no queues in its algorithms because its authors decided to concentrate on the coordination problem. We prefer however to put the queue in the foreground, since it seems that the requirement that the home-owner receives the messages of the postman (the device) and receives them in order is important for the functionality of the system.

We sum-up the requirements of a linear mailbox in Figure 2.

As for the liveness requirements, [4] requires that the algorithm is bounded wait-free, which means (see [7] under the term loop-free, or [5]) that each operation completes before the process executing it has taken $k$ steps, for some fixed constant $k$.

For communication, the Mailbox Problem as formulated in [4] requires atomic single-writer registers (shared variables). Recall that a register is serial if its read/write events are totally ordered (by the precedence relation) and the value of any read action is equal to the value of the last write action that precedes it. A register is atomic if its read/write actions

---

4In section 2.1 of [4] we read: “The remove and deliver procedures are used only for synchronization; the actual addition and removal of letters to/from 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.”
are linearizable into a serial register. That is, the partially ordered precedence relation has an extension into a total ordering so that the resulting register is serial. In this paper we assume that all registers are serial. This simplifies somewhat the presentation of the correctness proof because we do not have to speak about extending the partial order into a linear one, but it evidently does not limit the applicability of our algorithm which works as well with atomic registers.

For any serial register $R$ we define a function $\omega$ over the read actions of register $R$, such that for any read $r$, $\omega(r)$ is the last write action on $R$ that precedes $r$. That is, $\omega(r) < r$ and there is no write action $w$ on $R$ with $\omega(r) < w < r$. Then $r$ and $\omega(r)$ have the same value: $\text{Val}(r) = \text{Val}(\omega(r))$. (To ensure that $\omega(r)$ is defined on all read actions, we have to assume an initial write event that precedes all read events.)

As we have said, the mailbox algorithm uses both unbounded and bounded registers, but the check operation can access only the bounded registers. Following [4] the bounded registers are called “flags”, and so we have the postman flag which we call $F_P$ and the home-owner flag which we call $F_H$. The check operation only reads these flag registers (and contains no write on any register).

An additional “access restriction” is made in [4] for efficiency’s sake which requires that the check operation uses no persistent private variables in a check operation. Namely, the owner’s decision on whether to approach the mailbox or not should depend just on her readings of the $F_P$ and $F_H$ values and not on any internal information sustained from some previous operation. While one may argue that a small persistent variable would not harm the efficiency of the check operations, keeping the access restriction allows a comparison of the different mailbox algorithms (which obey the same restrictions). In fact, if we allow a persistent variables into our check algorithm, then the algorithm would need just one postman flag register of 6 values and a boolean flag for the home-owner.

2 The 6/4 mailbox algorithm

In this section we define in Figure 4 a mailbox algorithm with 6 and 4 values in its two bounded flag registers $F_P$ and $F_H$. The algorithm uses only serial registers. Registers $D_{\text{num}}$, $T_P$ and $F_P$ are written by the postman process, and registers $R_{\text{num}}$, $T_H$ and $F_H$ are written by the home-owner. Both processes can read these registers, but the check procedure only reads the bounded registers: $F_H$, $T_H$, $F_P$ and $T_P$. Registers $D_{\text{num}}$ and $R_{\text{num}}$ are unbounded (they carry natural numbers). The bounded registers of the postman process, namely $T_P$ and $F_P$, are collectively its flag register. Since register $T_P$ carries two values and register $F_P$ three values, the combined flag of the postman carries six values. The bounded registers of the homeowner process, $T_H$ and $F_H$, are both boolean, so that there are four values in these two registers which are the flag of the homeowner.

We describe the data structures of the different registers in figure 2. $F_P$ values for example are in $\{0, 1, 2\}$. The initial values of the registers is also defined in this figure. The initial value of the $F_P$ register for example is 2.
| registers of the postman: | type          | initially |
|--------------------------|---------------|-----------|
| D_num                    | natural number | 0         |
| T_P                      | \{0, 1\}      | 0         |
| F_P                      | \{0, 1, 2\}   | 2         |

| registers of the homeowner: | type          | initially |
|-----------------------------|---------------|-----------|
| R_num                       | natural number | 0         |
| T_H                         | \{0, 1\}      | 0         |
| F_H                         | Boolean       | false     |

Figure 3: Registers, their types and initial values.

In addition to the registers, we have the FIFO queue $Q$ which supports two operations: addition of a letter (executed by the postman process), and removal of a letter (executed by the homeowner when $Q$ is nonempty). $Q$ is initially empty.

The local variables of the algorithm are as follows. (Variables with unspecified initial values can take any initial value.)

**Local variables of postman:** $dn$ is a natural number, initially 0. $rn$ is a natural number, and $t$ is in $\{0, 1\}$.

**Local variables of homeowner:** Procedure `check` uses variable $fh$ (Boolean), $th$ and $tp$ (in $\{0, 1\}$), and $fp$ (in $\{0, 1, 2\}$). The `remove` procedure uses $rn$ and $dn$ that are natural numbers, and $t \in \{0, 1\}$. Initially $rn = 0$. Local variables of the `postman` process are obviously different from those of the `home-owner` even when they have the same name.

In order to ensure that the pseudocode of figure 4 is well-understood, we shall go over some of its instructions, make some simple definitions (that will be used later), and then we shall explain intuitively some of the main ideas of the algorithm.

A `deliver` operation execution $D$ is an execution of lines 1–6 of that code. It is a high-level event, namely the set of lower-level actions which are the executions of the code instructions. Any `deliver` execution is invoked with some letter parameter, and the first line of the code is an enqueue operation in which this letter is added to $Q$ (the mailbox queue).

Variable $dn$ (the `delivery number`) is initially 0, so that if $D$ is the $i$th `deliver` operation execution ($i = 1, 2, \ldots$) and $dn(D)$ denotes the value of $dn$ after line 2 is executed in $D$, then $dn(D) = i$. Register $D\_num$ thus contains the current delivery number.

We shall use this sort of notation $dn(D)$ for other variables as well. We note that in our algorithms any local variable is assigned a value in a unique instruction. So if $v$ is a local variable and $E$ some operation execution that assigns a value to $v$, then the notation $v(E)$ for that value that $E$ assigns to $v$ is meaningful and well defined. Likewise, if $G$ is any register such that $E$ contains a write into $G$ then we denote with $G(E)$ the value of
deliver (letter):

1 add letter to Q;
2 \( \text{dn} := \text{dn} + 1; \text{D\_num} := \text{dn}; \)
3 \( \text{t} := \text{T\_H}; \)
4 \( \text{TP} := 1 - \text{t}; \)
5 \( \text{rn} := \text{R\_num}; \)
6 if \( \text{rn} < \text{dn} \) then \( \text{FP} := 1 - \text{t} \)
   else \( \text{FP} := 2; \)

check():

1 \( \text{fh} := \text{F\_H}; \) if \( \text{fh} \) return true;
2 \( \text{th} := \text{T\_H} ; \)
3 \( \text{tp} := \text{TP} ; \)
4 \( \text{fp} := \text{FP} ; \)
5 return \( \text{tp} \neq \text{th} \land \text{fp} = \text{tp} ; \)

remove( )

1 remove one letter from Q;
2 \( \text{rn} := \text{rn} + 1 ; \text{R\_num} := \text{rn}; \)
3 \( \text{t} := \text{TP} ; \)
4 \( \text{TH} := \text{t} ; \)
5 \( \text{dn} := \text{D\_num} ; \)
6 \( \text{FH} := "\text{rn} < \text{dn}" ; \)

Figure 4: The 6/4 Mailbox Algorithm.

that write. Again, since any operation execution contains at most one write action into any
register, this notation is well defined.

In line 3, register \( \text{T\_H} \) is read into variable \( \text{t} \) and then the opposite value is written onto
register \( \text{TP} \). So, the postman is always changing the color obtained from the homeowner
process, while the homeowner always copy the value obtained (see lines 3 and 4 in the
remove code).

In executing line 5, register \( \text{R\_num} \) is read into local variable \( \text{rn} \), and in line 6 condition
\( \text{rn} < \text{dn} \) is checked. If it holds then \( 1 - \text{t} \) is written in \( \text{FP} \), but otherwise the value 2 is
written. So 2 is an indication that the mailbox is empty.

There are two sorts of check operations. A “short” check \( C \) is one that returns true
immediately after line 1 is executed. In this line, the homeowner process reads her own
register \( \text{F\_H} \) and returns true if that register’s value is true. Note that line 1 is the only place
in the algorithm where this register is read, and hence the register is in fact dispensable
and a local homeowner variable could replace it. The access restriction however prohibits
persistent variables, and hence the need for this register which does nothing more than
replacing a persistent local variable.

A “longer” check \( C \) is one in which all lines 1 to 5 are executed. In lines 2, 3, 4 registers
\( \text{T\_H} \), \( \text{TP} \) and \( \text{FP} \) are read, and the value that \( C \) returns is a conjunction of two statements
that involve \( \text{tp} \), \( \text{th} \), and \( \text{fp} \). Note that \( \text{TP} \) and \( \text{FP} \) are registers of the postman process, but
\( \text{th} \) is the value of register \( \text{T\_H} \) that the previous remove operation determined or else is the
initial value of that register (which is 0) in case \( C \) has no previous remove operation.

A remove operation is an execution \( R \) of lines 1–6 of the remove code. First a letter
is dequeued (and we have to prove that the queue is nonempty when this instruction is executed) and then the current removal number \(rn(R)\) is written on register \(R.num\). For any remove operation execution \(R\), \(rn(R)\) is the value of variable \(rn\) after line 2 is executed in \(R\). We have already noted that this notation is well defined since \(rn\) is assigned a value in \(R\) only at the execution of line 2. It follows that \(rn(R)\) is equal to \(i\) where \(R\) is the \(i\)-th remove operation execution. In lines 3 and 4 the homeowner copies the value read in register \(T.P\) into register \(T.H\). Register \(D.num\) is then read into \(dn\) (line 5) and the boolean value “\(rn < dn\)” is written in register \(F.H\).

The differentiation between a short and longer check operations reflects a main idea of the algorithm, namely that if the homeowner realizes in executing remove operation \(R\) that “\(rn < dn\)” (namely that the queue is nonempty), then no subsequent postman operations can change this fact, and hence the first check operation that comes after \(R\) can rely on this information and return true in a short execution.

There are two or three main ideas that shape our mailbox algorithm. The first one (very roughly speaking) is that the inequality of registers \(T.P\) and \(T.H\) indicates a nonempty queue. Initially both registers are 0, and in any deliver operation the postman reads \(T.H\) and writes in \(T.P\) a different value, thus indicating that the mailbox is nonempty. The homeowner cancels this indication in any remove operation, but the equality of the values of registers \(T.P\) and \(T.H\) is not an assurance that the queue is empty. For example, after several letters were deposited, the homeowner removes a single letter, leaving the two registers with equal value, and yet the queue is still nonempty. Of course, registers \(R.num\) and \(D.num\) give an exact estimation of the number of letters in the mailbox (namely \(D.num - R.num\)), but since the check operation is not allowed to access these unbounded registers it has to rely on the bounded registers. The homeowner also checks the boolean value \(F.H\) and if it is true then the queue must be nonempty and the check operation is short in this case. (The queue is nonempty in this case because if the previous remove operation has established that \(D.num - R.num > 0\) then the mailbox is nonempty since no remove operations were executed between the previous remove and the present check.) If, however, \(F.H\) is false, the homeowner needs a more complex evidence in order to deduce that the mailbox is nonempty: the inequality of colors \(tp \neq th\), and the accordance \(fp = tp\) (which also indicates that \(fp \neq 2\)).

An example can be useful here to explain why this condition \(tp \neq th \land fp = tp\) cannot be replaced with the simpler condition \(tp \neq th\). We see in figure 5 the following course of events.

1. postman execute a deliver operation \(D_1\).
2. home-owner execute a check operation \(C_1\), since postman has just delivered a letter, \(C_1\) is longer and positive.
3. postman starts to execute a second deliver operation \(D_2\) and execute the commands in lines 1 and 2. It sends the letter, writes 2 into register \(D.num\) and stops for awhile.
4. home-owner execute a remove operation \(R_1\). This is the first remove operation and
home-owner reads in this operation 2 from $D_{num}$ (the value that postman wrote to $D_{num}$ in $D_2$). Hence, $R_1$ is positive.

5. home-owner execute a check operation. Since $R_1$ is positive, $C_2$ is short and positive.

6. home-owner execute a remove execution $R_2$. $rn(R_2) = 2$ and $dn(R_2) = 2$ (the value that postman wrote to $D_{num}$ in $D_2$). Thus, $R_2$ is negative.

7. $P_1$ completes the execution of $D_2$, and execute the commands in line 3-6. It reads a value $c$ from $T_H$ (this value has been written to $T_H$ during the execution of $R_2$) and writes to register $T_P$, $1 - c$.

8. home-owner execute a check operation $C_3$. home-owner reads the value $c$ from $T_H$ (written in the execution of $R_2$) and reads the value $1 - c$ from register $T_P$ (written in the execution of $D_2$). Since only condition $tp \neq th$ is checked in $C_3$, $C_3$ is positive. Since there are only two deliver events and only two remove events in this execution, and since all of these executions precedes $C_3$, $C_3$ should be negative. Thus, this is an incorrect execution.

![Figure 5: An example for an incorrect execution where a long check event only checks condition $tp \neq th$.](image)

### 3 Correctness of the algorithm

In order to prove that our algorithm implements a mailbox (as specified in Figure 2), we need to define some functions and predicates that will serve us in this proof. An action is an execution of an atomic instruction of the algorithm such as a read or a write of a register or a queue action. Since we assume that the registers are serial, and as the queue operations (to add or remove a letter) are also instantaneous, we have a total ordering $<$ on these actions. We write $a < b$ to say that $a$ precedes $b$ in this ordering. (A relation $<$ is a total ordering when it is a transitive and irreflexive relation such that for any two different members $a$ and $b$ in its domain we have $a < b$ or $b < a$.)
An operation execution is an execution of the deliver, check, or remove algorithm. Every operation execution is a high-level event, namely a set of lower-level actions (also called lower-level events, as in [8]). The total ordering $< \! \!$ on the lower-level actions induces a partial ordering on the operation executions: for operation executions $A$ and $B$ we define that $A < B$ if $a < b$ for every $a \in A$ and $b \in B$. It is also very convenient to relate high-level events and lower-level actions: $A < x$ for a high-level event $A$ and a lower-level event $x$ means that $a < x$ for every $a \in A$. And similarly $x < A$ is defined when $x < a$ for every $a$ in $A$. The fact that we use the same symbol $<$ to denote both the total ordering relation on the actions and the resulting partial ordering relation on the high-level events should not be a source of confusion.

The aim of the correctness proof is to define a total ordering $\prec \! \! \! \! \!$ on the operation executions that extends the partial ordering $<$, and then to prove that the specifications of Figure 2 hold.

We assume two initial high-level events $I_p$ and $I_h$ by the postman and home-owner processes that determine the initial values of the registers (defined in Figure 2) and the initial values of the variables. $I_p$ contains the initial write actions on registers $D_{\text{num}}, T_P,$ and $F_P$, and $I_h$ contains the initial write actions on registers $R_{\text{num}}, T_H$, and $F_H$. These initial high-level events are concurrent. That is, it is neither the case that $I_p < I_h$ nor that $I_h < I_p$.

If $a$ is any read/write action, then $[a]$ denotes that high level event to which $a$ belongs. (Every low level action belongs to some operation execution, except for the assumed initial write actions which belong to the initial events $I_h$ and $I_p$.)

We shall name the different actions that compose the three operations.

1. Let $D$ be a deliver operation execution (which completed execution of lines 1–6 of the deliver code of Figure 2). We shall name the different actions of $D$. First, the addition of the letter to the queue $Q$ is denoted $enq(D)$. $D$ contains three write actions denoted $w_1(D)$, $w_2(D)$, and $w_3(D)$ (corresponding to lines 2, 4, and 6 respectively, namely the writes on registers $D_{\text{num}}, T_P,$ and $F_P$). $D$ contains two read actions $r_1(D)$ and $r_2(D)$ (which correspond to lines 3 and 5, namely to the reads of registers $T_H$ and $R_{\text{num}}$).

2. There are two sorts of check executions. A short operation $C$ is an execution of line 1 that returns the value true. It contains a single read, denoted $r_0(C)$, of register $F_H$. A longer check operation is one that contains executions of lines 1–5, and so it contains three additional read actions denoted $r_1(C)$, $r_2(C)$, and $r_3(C)$. $r_1(C)$ is the read of register $T_H$, $r_2(C)$ is the read of register $T_P$, and $r_3(C)$ is a read of register $F_P$. A check operation contains no write actions.

3. A remove operation execution $R$ begins with a dequeue action on the mailbox queue $Q$ which is denoted $\text{deq}(R)$. An important part of the correctness proof is to prove that whenever $\text{deq}(R)$ is executed, $Q$ is nonempty. There are two read actions in $R$, $r_1(R)$ and $r_2(R)$ which correspond to lines 3 and 5. These are the reads of registers $T_P$ and $D_{\text{num}}$. Then we notate the three write actions: $w_1(R)$ is the write on register $R_{\text{num}}$, $w_2(R)$ is the write on register $T_H$, and $w_3(R)$ is the write on $F_H$. 

10
If \( X \) is a deliver (remove) operation execution, then \( X \) contains a read action of the R\_num (respectively D\_num) register. Specifically, \( r = r2(X) \) is the read of the R\_num (respectively D\_num) register, and then \( \omega(r) \) is the write action of that register which affected \( r \). That is, \( \omega(r) \) is the last write action on register R\_num (respectively D\_num) that precedes \( r \) (see section \[1\]). Any action belongs to a unique higher level event, and if \( Y \) is that higher level event that contains the write \( \omega(r) \), then we define \( Y = \alpha(X) \).

A succinct definition of the function \( \alpha \) can be given by the following equation. For any deliver or remove operation \( X \) we define

\[
\alpha(X) = [\omega(r2(X))].
\]

Recall that \([a] \) denotes the higher level event that contains action \( a \). In case \( X = D \) is a deliver operation execution, \([\omega(r2(D))] \) is that high-level event that contains \( \omega(r) \), and so \( \alpha(D) \) can either be an operation execution that contains \( \omega(r) \), or else the initial event \( I_h \) of the home-owner process in case \( \omega(r) \) is the assumed initial write.

In case \( R \) is a remove operation execution, we have that \( \alpha(R) = [\omega(r2(R))] \). So if \( r = r2(R) \) is the read of register D\_num in \( R \), then \( \omega(r) \) is the corresponding write action on that register. We shall prove in Proposition \[3.8\] that \( \omega(r) \) is not the initial write in \( I_p \), and so \( D = \alpha(R) \) is a deliver operation execution and thus \( \omega(r) = w1(D) \).

The following lemma is an easy consequence of the fact that the registers (and specifically the D\_num register) are serial.

**Lemma 3.1** If \( R_1 < R_2 \) are two remove operations, then \( \alpha(R_1) \leq \alpha(R_2) \).

We say that a check operation \( C \) is “positive” in case it returns the value true. We say that it is “negative” when it returns false. Likewise, a remove operation \( R \) is positive when it writes true on its \( F_H \) register (in executing line 6), and it is negative when it writes false. And, again, a deliver operation \( D \) is positive if condition “\( rn < dn \)” holds at line 6 of \( D \), and it is negative otherwise.

Now we define two functions, \( \text{pre}_\text{rem} \) and \( \rho \), on the check events.

**Definition 3.2** Let \( C \) be any check operation execution. Define \( \text{pre}_\text{rem}(C) \) as the last remove operation execution \( R \) such that \( R < C \) if there is such a remove execution that precedes \( C \), and \( \text{pre}_\text{rem}(C) = I_h \) as the assumed initial home-owner event otherwise.

We note that a short check operation is positive, and hence a check operation \( C \) is short if and only if \( \text{pre}_\text{rem}(C) \) is positive. Since the assumed initial homeowner event is negative (as the initial value of \( F_H \) is false), if \( C \) is short then \( \text{pre}_\text{rem}(C) \) is not the initial event– it is necessarily a positive remove operation execution.

The following is a key definition in our correctness proof. It relates every check operation \( C \) to \( \rho(C) \) which is the deliver operation (or initial \( I_p \) event) that \( C \) considers in order to calculate the value (true or false) to return.
Definition 3.3 For any check operation execution $C$ we define $\rho(C)$ as follows. In case $C$ is a short check operation let $R = \text{pre}_\text{rem}(C)$ (which is a remove operation execution as we noted) and then define $\rho(C) = [\omega(r2(R))]$. So $\rho(C) = \alpha(\text{pre}_\text{rem}(C))$ when $C$ is short. In case $C$ is a longer operation, define $\rho(C) = [\omega(r3(C))]$. ($r3(C)$ is the read of $F_p$ in $C$.)

We note that $C < \rho(C)$ is impossible, by properties of the $\omega$ function (namely by the fact that $\omega(r) < r$ for any read action $r$). The following is therefore established.

Proposition 3.4 If $C < D$ (where $C$ is a check and $D$ a deliver operation) then $\rho(C) < D$.

Lemma 3.5 Suppose that $C < R$ are a check and remove operation executions. Then $\rho(C) \leq \alpha(R)$.

Proof. If $C$ is short then $\rho(C) = \alpha(\text{pre}_\text{rem}(C))$, and since $R' = \text{pre}_\text{rem}(C) < C < R$, $R' < R$ follows and so the proof is concluded in this case with Lemma 3.1. Suppose next that $C$ is a longer check operation and $\rho(C) = D$. Then $D = [\omega(r3(C))]$ by definition of $\rho$. This implies that $D < r3(C)$. (Because if $D = I_p$ is the initial event then $D < C$, and if $D$ is a deliver operation then the fact that the write on $F_p$ is the last action in $D$ implies that $D < r3(C)$.) So $D < R$ and hence $D \leq \alpha(R)$. ☐

We remind the reader that if $E$ is any operation execution and $x$ a variable (or a register) whose value is assigned in $E$, then $x(E)$ denotes this value.

For any remove operation $R$, $\text{rn}(R)$ is the value of variable $\text{rn}$ that is determined in executing line 2 and is written on register $\text{R}_\text{num}$. We also set $\text{rn}(I_h) = 0$ (and the initial value of variable $\text{rn}$ is 0).

$\text{rn}(R)$ is called the “removal number”; it is the number of remove operations $R'$ such that $R' \leq R$. Clearly, if $R_1 < R_2 < \cdots$ is the sequence of remove operations in increasing order, then $\text{rn}(R_i) = i$.

The check code does not contain a variable named $\text{rn}$, and so the number $\text{rn}(C)$ for a check operation execution $C$ is defined directly as the number of remove operations $R$ such that $R < C$. In other words,

$$\text{rn}(C) = \# \{R \mid R \text{ is a remove operation and } R < C \}. \tag{3}$$

Where $\#A$ denotes the cardinality of the set $A$.

The “delivery number”, $\text{dn}(D)$, of a deliver operation $D$ is equal to the number of deliver operations $D'$ such that $D' \leq D$. Thus if $D_1 < D_2 < \cdots$ is the enumeration of the deliver operations in increasing order, then $\text{dn}(D_i) = i$. We also define $\text{dn}(I_p) = 0$.

It is convenient to define the “color” of operations. If $D$ is a deliver operation, then $\text{color}(D) = T_p(D) = 1 - t(D)$. That is, $\text{color}(D)$ is that value $c = 0, 1$ that is written into register $T_p$ when line 4 is executed in $D$. (If condition $\text{rn} < \text{dn}$ holds in line 6, then $\text{color}(D)$ is also the value that is written in register $F_p$.)
The color of any remove operation \( R \) is defined by \( \text{color}(R) = t(R) = T_H(R) \). That is, the color of \( R \) is the value read from register \( T_P \) and written into \( T_H \). The color of the initial event \( I_p \) is 0 which is the initial value of \( T_H \).

If \( C \) is a long check operation, then we define \( \text{color}(C) = tp(C) \). That is, the color of a long check operation is the value read from register \( T_P \).

Note that if \( C \) is a long check operation and \( D = \rho(C) \), if \( C \) is positive then \( D \) is positive and \( \text{color}(D) = \text{color}(C) \). Indeed, \( D = \rho(C) \) implies that the value read in register \( F_P \) in \( C \) (namely \( fp(C) \)) is the value written by \( D \). Hence this value is not 2 (because \( C \) is positive and condition \( tp = fp \) implies that \( fp = 0, 1 \)). Hence \( rn < dn \) holds in \( D \), and therefore \( D \) is positive, and \( \text{color}(D) = \text{color}(C) \) follows.

Note also that if \( C \) is a long check operation and \( S = \text{pre}_\text{rem}(C) \) (a remove operation or \( I_p \)), if \( C \) is positive then \( \text{color}(C) \neq \text{color}(S) \). This follows from equality \( tp \neq th \) which holds at line 5 if \( C \) is positive.

We gather these observations into the following.

**Lemma 3.6** If \( C \) is a long check operation and \( S = \text{pre}_\text{rem}(C) \), then \( C \) is positive iff \( \rho(C) \) is positive, \( \text{color}(C) \neq \text{color}(S) \), and \( \text{color}(\rho(C)) = \text{color}(C) \).

Our aim now is to prove some properties of the functions and predicates that we have defined above. These properties will be used to define a linear ordering (total ordering) \( \prec \) on the operation executions and to prove that the properties of Figure 2 hold.

**Lemma 3.7** Suppose that \( C \) is a long check operation and \( D = \rho(C) \). If \( S = \text{pre}_\text{rem}(C) \) is a remove operation such that \( w2(D) < r1(S) \), then \( C \) is negative.

Proof. Let \( c = \text{color}(D) \) be, as we have defined above, the value written into register \( T_P \), and assume for a contradiction that \( C \) is positive. So \( \text{color}(C) = \text{color}(D) \) by Lemma 3.6. Since \( w2(D) < r1(S) \) are a write and read actions on register \( T_P \), \( w2(D) \leq \omega(r1(S)) \).

Case 1. \( w2(D) = \omega(r1(S)) \). This entails that \( \text{color}(D) = \text{color}(S) \), and hence that \( \text{color}(C) = \text{color}(S) \) which implies by Lemma 3.6 that \( C \) is negative.

Case 2. \( w2(D) < \omega(r1(S)) \). This implies that \( D < D' \) where \( D' = [\omega(r1(S))] \), and \( \omega(r1(S)) = w2(D') \). So \( w2(D') < r1(S) \) (as \( \omega(r) < r \) for every read action \( r \)). Since it is not the case that \( w3(D') < r3(C) \) (as \( \rho(C) = D \)), we get that \( r3(C) < w3(D') \). Hence \( w2(D') < r1(S) < r2(C) < w3(D') \). This implies that \( r1(S) \) and \( r2(C) \) (which are both reads of register \( T_P \)) get the same value of the write \( w2(D') \). Hence \( \text{color}(S) = \text{color}(C) \) which implies, again by Lemma 3.6 that \( C \) is negative.

**Proposition 3.8** If \( C \) is a positive check operation and \( \rho(C) = D \), then \( D \) is a deliver operation execution and

\[
\text{rn}(C) < \text{dn}(D).
\] (4)
Proof. Assume first that \( C \) is a short check operation and let \( R = \text{pre}_{\text{rem}}(C) \) be the previous remove operation, which necessarily has set its register \( F_H \) to be true at line 6. So \( rn(R) = rn(C) \), and inequality

\[
rn(R) < dn(R)
\]  

(5)

holds. Let \( r = r2(R) \) be the read of register \( D_{\text{num}} \) which obtained the value \( dn(R) \). By definition of \( \rho(C) \) when \( C \) is short, \( D = \rho(C) = [\omega(r)] \), and \( dn(D) = dn(R) \) follows. Since \( dn(R) > 0 \) follows from \( 5 \) and as \( dn(I_p) = 0 \), \( D \neq I_p \) is concluded and necessarily \( D \) is a deliver operation execution and \( 11 \) follows.

Now suppose that \( C \) is a longer check operation, and let \( r2 = r2(C) \) and \( r3 = r3(C) \) be its reads of registers \( T_p \) and \( F_p \) (respectively). By definition of \( D = \rho(C), D = [\omega(r3)] \). Then \( D \) is either a deliver operation execution (in which case \( \omega(r3) = w3(D) \) is the write in register \( F_p \) or else is the initial event \( I_p \) in case \( \omega(r3) \in I_p \). We claim that \( D \) is not the initial event \( I_p \). Indeed, the initial value of \( F_p \) is 2, but as \( C \) is positive condition \( fp = tp \) holds in \( C \), which excludes the possibility that \( fp(C) = 2 \) (as \( tp(C) \in \{0, 1\} \)). Hence \( fp(C) = fp(D) \) is not 2 and so \( rn < dn \) is evaluated to true when line 6 is executed in \( D \). So

\[
rn(D) < dn(D)
\]

holds.

Define \( R = \alpha(D) \); that is \( R = [\omega(r2(D))] \). Then \( rn(D) = rn(R) \). We shall prove that \( R = \text{pre}_{\text{rem}}(C) \). This will show that \( rn(C) = rn(R) \), and hence that \( rn(C) < dn(D) \) as required. It thus remain to prove that \( R = \text{pre}_{\text{rem}}(C) \).

Suppose on the contrary that \( R \neq \text{pre}_{\text{rem}}(C) \), and then \( R < \text{pre}_{\text{rem}}(C) \) follows (from the fact that \( w1(R) < r2(D) < w3(D) < r3(C) \) which implies that \( R < C \)). Say \( S = \text{pre}_{\text{rem}}(C) \). Since \( \omega(r2(D)) \) is in \( R \), \( r2(D) < w1(S) \). But \( w2(D) < r2(D) \). Hence \( w2(D) < w1(S) < r1(S) \) and this implies by Lemma 3.7 that \( C \) is not positive, which yields a contradiction.

\[\Box\]

**Proposition 3.9** If \( D < C \) are a deliver and check operations such that \( rn(C) < dn(D) \), then \( C \) is positive.

Proof. A short check operation is always positive (it returns true), and hence we may assume that \( C \) is a longer check. Say \( R = \text{pre}_{\text{rem}}(C) \) and then

\[
rn(R) = rn(C).
\]

(6)

Suppose first that \( R = I_h \) is the initial event. In reading \( T_H \), \( C \) obtains the initial value 0. We shall prove that \( fp(C) = tp(C) = 1 \), and hence that \( C \) returns true at line 5, as required.

Define \( E = \rho(C) = [\omega(r3(C))] \). Then \( w3(E) < r3(C) \) (the write on \( F_p \) in \( E \) precedes the read of this register in \( C \)), and \( D < C \) implies that \( D \leq E \). Now \( \alpha(E) = I_h \) follows from the assumption that \( \text{pre}_{\text{rem}}(C) = I_h \). \( rn(E) \) is 0 (the initial value of \( R_{\text{num}} \)), but \( dn(E) > 0 \). So \( "rn < dn" \) holds in \( E \) when line 6 is executed in \( E \), and hence the value of \( w3(E) \) is
1 − t(E). But t(E) = 0 because the initial value of T_H is 0, and hence the value of w3(E) is 1. So fp(C) = 1. The proof that tp(C) = 1 is very similarly obtained by taking [ω(r2(C))].

So now we assume that R is a remove execution. In case w1(D) < r2(R), w1(D) ≤ ω(r2(R)) follows, and hence the read of D_num in R obtains the write in D or a later write. Hence dn(D) ≤ dn(R). The fact that rn(R) = rn(C) and our assumption that rn(C) < dn(D) imply that rn(R) < dn(R). So R is positive and C is a short positive check operation.

So we may assume that r2(R) < w1(D). It follows from this assumption that w2(R) < r1(D) < C. Say c = color(R) (that is, by definition, the value of w2(R), which is the write in T_H).

Claim. If E is any deliver operation such that w2(R) < r1(E) < r3(C) (the write on T_H in R precedes the read of T_H in E which itself precedes the end of C) then ω(r1(E)) = w2(R) and color(E) = 1 − c.

Proof of claim. Since r1(E) is before the end of C there is no write action on register T_H between w2(R) and r1(E). Hence w2(R) = ω(r1(E)). So color(E) = 1 − c.

In particular, if E_0 = ρ(C), then D ≤ E_0 and the conditions of the claim hold. (Recall that r3(C) is the read of register F_P in C, and ρ(C) = [ω(r3(C))]. Since D < C, D ≤ E_0. And as the write on F_P is the last action in E_0, E_0 < r3(C).) Thus color(E_0) = 1 − c. Moreover, R = α(E_0). To prove this fact note that w1(R) < w2(R) < r1(E_0) < r2(E_0) and r2(E_0) is before the end of C; this implies that w1(R) = ω(r2(E_0)) and hence that rn(E_0) = rn(R). But (6) and the lemma’s assumption give rn(R) = rn(C) < dn(D), and since D ≤ E_0 yields dn(D) ≤ dn(E_0), condition rn(E_0) < dn(E_0) holds. Hence the value of F_P that is written by E_0 is the color of E_0 which is 1 − c. Since E_0 = ρ(C), this implies that fp(C) = color(E_0) = 1 − c, and thus

\[ fp(C) = 1 − c. \]

Condition fp = tp holds in C by the following argument. tp(C) is the value of the read of T_P, namely the value of r2(C). Say E = [ω(r2(C))], that is w2(E) = ω(r2(C)). Since D < C, D ≤ E. Also, r1(E) is before the end of C. As we noted in the above claim, this implies that color(E) = 1 − c, and hence

\[ tp(C) = 1 − c. \]

In view of the formula displayed above, this yields that fp = tp holds in C.

Since color(R) = c, R writes c on T_H. But R = pre_rem(C), and so th(C) = c follows. Hence condition tp ≠ th holds in C because tp(C) = 1 − c but th(C) = c. So C is indeed positive.

We are now ready to define the linear ordering < on the deliver, check and remove operations. We shall define first a relation < that extends < on the operation executions, and then prove that < has no cycles, and that any linear ordering < that extends < satisfies the linear mailbox specifications of Figure 2. This will complete the proof. We define the
relation \(<^*\) as a union of \(<\) with the relation \(<\) that relates some check operations \(C\) and deliver operations \(D\) as follows.

\[
\begin{align*}
\langle C, D \rangle & = \{ (C, D) | C \text{ is negative and } \text{rn}(C) < \text{dn}(D) \} \\
 & \cup \{ (D, C) | C \text{ is positive and } \text{dn}(D) = \text{rn}(C) + 1 \}.
\end{align*}
\]

Before we proceed we want to explain the intuition behind this definition of \(<\). If \(C\) is a negative check operation and \(\text{rn}(C) < k\), if \(D\) is the \(k\)th deliver operation or a later deliver, then we surely want to have \(D\) after \(C\) in the linear ordering \(\prec\) that we look for. (Otherwise, if \(D\) is before \(C\), then \(C\) is required to be positive.) If, on the other hand, \(C\) is positive then among the operations that are before \(C\) in the \(\prec\) ordering we must have more deliver than remove operations and hence the \(k + 1\) deliver operation must be before \(C\).

**Lemma 3.10** If \(X < Y\) then it is not the case that \(Y < X\).

**Proof.** We have to check two cases as in the definition of \(X < Y\).

1. Suppose first that \(\langle X, Y \rangle = \langle C, D \rangle\) where \(C\) is a negative check operation and \(D\) is a deliver operation such that \(\text{rn}(C) < \text{dn}(D)\). We have to prove that it is not the case that \(D < C\). But if \(D < C\) then Proposition 3.9 implies that \(C\) is positive.

2. Suppose next that \(\langle X, Y \rangle = \langle D, C \rangle\) where \(C\) is a positive check operation and \(D\) is a deliver operation such that \(\text{dn}(D) = \text{rn}(C) + 1\). We have to prove that it is not the case that \(C < D\). But if \(C < D\), then \(D' = \rho(C) < D\) (by Proposition 3.4) and hence \(\text{dn}(D') < \text{dn}(D)\). So \(\text{dn}(D') \leq \text{rn}(C)\), in contradiction to Proposition 3.8.

\(\blacksquare\)

**Lemma 3.11** If \(C\) and \(C'\) are check operations and \(D\) is a deliver operation such that \(C < D < C'\), then \(C < C'\).

**Proof.** Since \(C < D\), the definition of \(<\) implies that \(C\) is negative, and

\[\text{rn}(C) < \text{dn}(D).\]

Now from \(D < C'\) we get that \(C'\) is positive and \(\text{dn}(D) = \text{rn}(C') + 1\). So, firstly, we infer that \(C \neq C'\) (one is negative and the other positive). If it is not the case that \(C < C'\), then \(C' < C\) holds. In this case, since \(C'\) is positive, there is a remove operation between \(C'\) and \(C\), and hence \(\text{rn}(C') < \text{rn}(C)\). So, \(\text{rn}(C') < \text{rn}(C) < \text{dn}(D)\) which is in contradiction to \(\text{dn}(D) = \text{rn}(C') + 1\).

\(\blacksquare\)

**Lemma 3.12** If \(D\) and \(D'\) are deliver operations and \(C\) a check operation, then \(D < C < D'\) is impossible.
Proof. $D \triangleleft C$ implies that $C$ is positive but $C \triangleleft D'$ implies that it is negative.

A cycle of length $k \geq 1$ in a relation $T$ is a sequence $X_1, \ldots, X_{k+1}$ so that $X_i T X_{i+1}$ for $1 \leq i \leq k$, and $X_{k+1} = X_1$. We say that $X_{i+1}$ is the successor of $X_i$ in this cycle.

**Lemma 3.13** Relation $<^* = (< \cup \triangleleft)$ has no cycles, and hence can be extended to a linear ordering of the operation executions.

Proof. By the definition of the union of two relations, $X <^* Y$ if $X < Y$ or $X \triangleleft Y$. Suppose on the contrary that there is a cycle $X_1 <^* X_2 <^* \cdots <^* X_n$ of length $n \geq 1$ in the $<^*$ relation. Take such a cycle of minimal length. Since $<$ is transitive, there are no two successive occurrences of the $<$ relation in this minimal cycle. But it is also impossible to have two successive occurrences of the $\triangleleft$ relation (by lemmas 3.11 and 3.12). The cycle is not of length one, since both $<$ and $\triangleleft$ are irreflexive. The cycle is not of length two (use Lemma 3.10 to see that it is not of the form $X \triangleleft Y < X$ or $X < Y \triangleleft X$).

We may assume that the cycle begins with the $<$ relation, and so it begins $X_1 < X_2 \triangleleft X_3 < X_4 \cdots$. But $X_2 \triangleleft X_3$ implies (by Lemma 3.10) that it is not the case that $X_3 < X_2$. So $\text{begin}(X_2) < \text{end}(X_3)$, where $\text{begin}(X)$ and $\text{end}(X)$ are the first and last actions in $X$. Hence $X_1 < X_4$ follows in contradiction to the minimality of the cycle.

As $<^*$ has no cycles it can be extended to a linear ordering.

**Theorem 3.14** Let $<$ be any linear ordering (total ordering) that extends $<^*$. Then the specifications of Figure 2 hold.

Proof. For any check operation $C$ we define $\text{Val}(C) = \text{true}$ if $C$ is a positive, and $\text{Val}(C) = \text{false}$ when $C$ is negative. We now check the four items of Figure 2.

1. $<$ is chosen to be a linear ordering that extends $<^*$, and hence it also extends the $<$ ordering on the operation executions. We want to show that for every operation execution $X$ the set $\{Y \mid Y < X\}$ is finite. This is a consequence of the finiteness property of the $<$ relation which says that for every event $X$ there is only a finite number of events $Y$ such that $X < Y$ does not hold. Hence for all but a finite number of events $X < Y$ holds.

2. If $R$ is any remove operation, then $R$ is preceded by a positive check operation $C$. This is a requirement on how the operations are invoked, and since the home-owner process is a serial process the two ordering $<$ and its extension $<$ agree on the operations of that process, and so there is no check or remove operation execution $X$ with $C < X < R$.

3. Recall that for every check operation execution $X$, $\text{deliver\_num}(X)$ and $\text{removal\_num}(X)$ are the number of deliver operations $D$ such that $D < X$, and (respectively) the number of remove operations $R$ such that $R < X$. We have defined (in (3)) the number $\text{rn}(C)$ as the number of remove operations $R$ such that $R < C$. Since the homeowner process is serial, relations $<$ and $<$ coincide on the homeowner events, and hence

$$
\text{rn}(C) = \text{removal\_num}(C). 
$$
And similarly, for any deliver $D$

$$dn(D) = deliver\_num(D).$$

(8)

We have to show that

$$Val(C) = "removal\_num(C) < deliver\_num(C)".$$  

(9)

(Where “ϕ” is the truth value of ϕ.) Consider first the case that $C$ is negative, and assume that in contradiction to (9) $removal\_num(C) < deliver\_num(C)$. Say $removal\_num(C) = k$. So

$$k < deliver\_num(C).$$

(10)

If $D_1 < D_2 \cdots$ is an enumeration in increasing $<$ order of the deliver operations, then $D_{k+1} < C$ (for otherwise, as $<$ is a linear ordering, $C < D_{k+1}$ and hence

$$\{D \mid D \text{ is a deliver operation and } D < C\} \subseteq \{D_1, \ldots, D_k\}$$

which implies that $deliver\_num(C) \leq k$ in contradiction to (10)). Yet, as $C$ is negative, $rn(C) = k$ and $dn(D_{k+1}) = k + 1$, the definition of $<$ dictates that $C < D_{k+1}$, which is in contradiction to $D_{k+1} < C$.

Consider now the case that $C$ is positive. Say $D = \rho(C)$. By Proposition 3.8 $rn(C) < dn(D)$. Hence we do have a deliver operation $D$ with $dn(D) = rn(C) + 1$. Then $D < C$ and hence $D < C$. This shows that $deliver\_num(D) \leq deliver\_num(C)$. But $rn(C) < dn(D)$ and equations (7) and (8) show that $removal\_num(C) < deliver\_num(D)$ and hence that (9) holds.

4. The fourth property of Figure 2 is that $R_i$ obtains the letter of $D_i$. Let $C$ be that positive check operation that precedes $R_i$. Then $rn(C) = i - 1$. Define $D = \rho(C)$. By Proposition 3.8 $rn(C) < dn(D)$. Hence $dn(D) \geq i$. So

$$D_i \leq D.$$  

This implies that $enq(D_i) < deq(R_i)$ (see below) and since this relation holds for every $i$ and as we assume that the queue $Q$ that the algorithm employs is a fifo queue, it follows that the value dequeued by $R_i$ is the value enqueued by $D_i$. Why $enq(D_i) < deq(R_i)$?

If this is not the case and $deq(R_i) < enq(D_i)$, then the fact that the enqueue action is the first in any deliver operation yields (together with $C < R_i$) that $C < D_i \leq D$. But $C < D$ is in contradiction to $D = \rho(C)$.

4 A note on the proof

Our correctness proof of the linearizability of the mailbox algorithm that was given in the previous section is clearly divided into two parts. The first part consists in defining relations
and functions such as $\alpha$ and $\rho$, and in proving properties of the operation executions that are expressed by means of these relations and functions. This part of the proof is extended from Lemma 3.1 to Proposition 3.9 and it relies on the text of the algorithm. The proof in the second part defines the linearization ordering $\prec$ and shows that it possesses the required properties (those that are displayed in Figure 2). In this part, the algorithm is not mentioned and only properties established in the first part are used in an abstract way. Although the proof of both parts was quite detailed and (we hope) convincing, we cannot claim that it is a formal proof because something very definite is lacking which we want to explicate. The correctness condition (linearizability) is about executions of the algorithm, but we never defined what these executions are; we never defined mathematical objects that represent executions and so we did not explicate in a precise way how to formulate and formally prove theorems about executions.

The standard way to define executions of a distributed algorithm is the following which is based on the notions of states, steps and runs. A state is, informally speaking, a description of the system as if frozen at a certain moment. Formally, a state is a function that assigns values to the state variables. Variables of our system are, for example, $PC_p$ (the postman program counter) which can take any of the values in \{1, …, 6\}, $PC_h$ (which is the homeowner program counter), $TH$ (which is the register with values in \{0, 1\}) etc. If $S$ is a state and $x$ is any of the state variables, then $S(x)$ denotes the value of $x$ in state $S$. An initial state is a state $S$ such that $S(\text{PC}_p) = 1$, $S(\text{D_num}) = 0$, and so on as in Figure 2.

A step is a pair of states $(S, T)$ that represents an execution of an (atomic) instruction by one of the processes. So, for example, a “read of register $TH$” by the postman process is a step $(S, T)$ such that $S(\text{PC}_p) = 3$, $T(\text{PC}_p) = 4$, $T(t_p) = S(TH)$ and for any variable $x$ different from $\text{PC}_p$ and $t_p$, $T(x) = S(x)$.

A run is defined to be a sequence of states $S_0, \ldots$ such that $S_0$ is an initial state and for every $i$ $(S_i, S_{i+1})$ is a step by one of the processes. Runs represent executions of the algorithm.

These runs cannot support the lemmas and propositions of the first part of our linearization proof and certainly they do not suffice for its second part, simply because the high level events, namely the operation executions, are not an integral part of these runs. Proposition 3.8 for example, requires the notion of check and deliver operations, as well as the functions $\alpha$, $dn$ and $rn$. Now, incorporating these higher level events and functions is nothing very deep. We can simply take a run with its actions (formed by the steps) and define sets of actions that form the operation executions. This yields a structure that contains both actions and higher level events, and the functions $\alpha$, $\rho$, etc. can be defined in this resulting structure as we did in the previous section. A detailed description of this process by which the extended run structure is obtained may be quite long, but it is quite straightforward. In fact, there are possibly more then one reasonable way to achieve this construction and a particular one can be found in [2] and [1].

If we denote with $H$ some run of the system, that is some sequence of states $H = (S_0, \ldots)$ so that every pair $(S_i, S_{i+1})$ is a step, and if we let $\overline{H}$ be the resulting extended structure that contains both the actions, the higher level operation executions and the required functions,
then all the lemmas and propositions of the first part of our proof refer to the structure $\mathcal{H}$ (or more correctly to the set of all structures $\mathcal{H}$ obtained from runs $\mathcal{H}$ of the system).

Now for the second part of the proof we no longer need the actions and references to the algorithm instructions. The structures that interest us are those obtained by forgetting all references to lower level actions and keeping only the higher level operation executions and the required functions and relations that are defined over them. Let $\mathcal{H}$ be the extended structure that results from a run $\mathcal{H}$. Then we can form a structure $\mathcal{M}$ by keeping only the operation executions (as members of the universe of $\mathcal{M}$), the precedence relation $<$ over these members and all functions and predicates that are defined over them. The resulting structure $\mathcal{M}$ is the one on which the second part of our linearization proof is about. $\mathcal{M}$ is a structure in the standard sense that is given in mathematical logic books. It is an interpretation of some definite relational language. Any structure $\mathcal{M}$ obtained in this way satisfies the properties that were established in Propositions 3.8 to 3.9 and some additional obvious properties, and the second part of the correctness proof establishes that any structure that satisfies these properties possesses a linearization as required by the linear mailbox specification of Figure 2. We refer to structures such as $\mathcal{M}$ as Tarskian system executions.

A careful reader would surely not be happy with our “additional obvious properties”, and she would rightly request a more detailed definition. What is needed (for a careful correctness proof) is a definition of a first-order language $L$ and a list of properties $PL$ that include not only those enunciated by the propositions but also all those additional properties that are required for the proof. Then the fact that the structures $\mathcal{M}$ are detached from the algorithm help us to check that indeed only the assumptions made in the list $PL$ (and all of these properties) are used in the second part of the proof. In our experience, this separation of the correctness proof into two parts with the corresponding separation of the modeling structures helps to improve the algorithms whose correctness we try to prove. What often happens is that when the second part of the proof is established and it is evident that only the properties listed in $PL$ are needed, then the algorithm itself can be changed and improved by the designer who knows that if only these properties of $PL$ still hold then the algorithm is correct.

To give an idea of what we have in mind for the list $PL$ we spell out in details such a list, but we first describe the language to which the statements of this list belong. The $L$ language is a multi-sorted language that contains the following elements.

1. There are two sorts: $Event$ and $Number$. (The role of sort $Event$ is to represent the operation executions, and the role of $Number$ is to represent the set of natural numbers.)

2. The following unary predicates are defined over $Event$: $deliver$, $check$, $remove$, $positive$ and $negative$.

3. A binary relation $<$ is defined on the $Event$ sort. (This is called the precedence relation.) The same symbol $<$ is also used for the “smaller than” relation on the $Number$ sort.

\[\text{This term was chosen in order to indicate that we incorporate here the notion of system execution defined by Lamport} \] with the work and ideas of Alfred Tarski.
The successor function \( x + 1 \) is also assumed here.

4. The functions \( r_n \) and \( d_n \) are defined over the Event sort and they take Number values.

5. The function \( \rho \) is defined on the Event sort and with values in this sort. (In fact, we are only interested in \( \rho(C) \) when \( C \) is a positive check event, and in this case \( \rho(C) \) is a deliver event.)

The PL properties are defined to be the following "axioms". (For simplicity we did not introduce queue events and did not relate the deliver and remove events to the queue events.)

1. Relation \(<\) is irreflexive and transitive on the Event sort, and it satisfies the following property\(^4\).

   (a) For every Event members \( X_1, X_2, X_3, X_4 \):
   
   if \( X_1 < X_2, X_3 < X_4 \) and \( X_2, X_3 \) are incomparable in \(<\), then \( X_1 < X_4 \).

   (b) For every event \( A \) there is a finite set of events \( F \) such that if \( Y \) is any event not in \( F \) then \( X < Y \).

2. The deliver, check, and remove predicates are disjoint. We write home-owner\((x)\) for \( check(x) \lor remove(x) \).

3. The deliver events are linearly ordered. That is, if deliver\((e_1)\) and deliver\((e_2)\), if \( e_1 \neq e_2 \), then \( e_1 < e_2 \) or \( e_2 < e_1 \).

   The function \( d_n \) is an enumeration of the deliver events in their ordering. That is, for every deliver event \( d \), \( d_n(d) \) is the number of deliver events \( d' \) such that \( d' \leq d \). (So \( d_n \) is one-to-one, into Number and with values \( > 0 \), so that for every deliver events \( d_1 \) and \( d_2 \), \( d_n(d_1) < d_n(d_2) \) iff \( d_1 < d_2 \), and if \( d_n(d) = k \) then for every \( 1 \leq j < k \) there exists some deliver \( d' \) with \( d_n(d') = j \).)

4. The home-owner set of events is linearly ordered, and if home-owner\((x)\) then \( r_n(x) \) is the number of remove events \( r \) such that \( r \leq x \).

5. We assume an initial event \( I \) and \( I < e \) for any other event \( e \).

6. Any check event is either positive or else negative. If \( C \) is a positive check event then there exists some remove event \( R \) such that \( C < R \) and there is no home-owner event \( X \) with \( C < X < R \).

   If \( R \) is a remove event then there is some positive check \( C \) such that \( C < R \) and there is no home-owner event \( X \) with \( C < X < R \).

\(^4\)This is the Russell–Wiener property which characterizes interval orderings.
7. If $C$ is a positive check event and $\rho(C) = D$, then $D$ is a deliver operation and $rn(C) < dn(D)$.

8. If $D < C$ are a deliver and (respectively) a check events such that $rn(C) < dn(D)$ then $C$ is positive.

9. If $C < D$ are a positive check and (respectively) a deliver events, then $\rho(C) < D$.

The last three items, 7,8 and 9, are the main properties and they were established in propositions 3.8, 3.9 and 3.4. The reader can return now to section 3 and re-read the second part of the proof, but now as if it were an abstract proof about arbitrary structures that posses the nine properties listed above. The reader can check that indeed only these properties are used in the proof and each one serves at some point. (The argument that involves the begin and end functions can be adapted to one the employs the Russell–Wiener property.)

The role of the function $\rho$ is intuitively evident. If $C$ is a positive check operation then it must be the case that $C$ relies on some deliver operation execution $D$ that ensured $C$ that it may return true. The function $D = \rho(C)$ gives this assurance, based on the inequality $dn(D) > rn(C)$. And of course, we cannot expect that $C$ relies on some future event: hence $C < \rho(C)$ is ruled out. It is not difficult to check that $\rho$ is not only intuitively appropriate, but it is in fact necessary in the sense that if we do have a mailbox algorithm for which a linear ordering $\prec$ exists that satisfies the condition of Figure 2 then a function $\rho$ can be defined that satisfies items 7 and 9.

5 Conclusion

In [4], Aguilera, Gafni, and Lamport define the Mailbox problem, and present a solution in which the check operation reads two registers (the “flag” registers) that can carry 14 values each. Moreover, they prove that there is no solution to the Mailbox problem with two binary flags. We have presented here a much simpler solution to the Mailbox problem with two flags that can carry 6 and 4 values each. The gap between the impossibility of solving the Mailbox problem with binary flags and our solution with flags that have 10 values in total is meaningful and it poses interesting theoretical questions: to improve on the lower bound of [4], and to find a better solution to the Mailbox problem than the one presented here.

Another problem from [4] is whether the space efficiency of the mailbox algorithm presented in that paper can be improved. The algorithm of [4] uses $\Theta(n \log n)$ bits of shared memory, where $n$ is the number of executions of deliver and remove. The authors of [4] conjecture that there is a solution using logarithmic space, and indeed our algorithm uses two registers $D_{\text{num}}$ and $R_{\text{num}}$ of width exactly $\log n$ for $n$ executions.

An interesting problem (connected with the Mailbox problem) is posed in [4]: the bounded, wait-free Signaling problem for which [4] gives only a non-blocking solution and leaves the wait-free problem open. The ideas developed in this paper have contributed to a solution of the wait-free Signaling problem which was obtained by the second author.
There are other problems around the Mailbox problem that seem to be quite interesting. Are there solutions to the mailbox problem in which all registers (not only the flag registers) are bounded? What solutions to the mailbox problem can be obtained in which the flags are simple registers but the other registers and queues can be more complex shared memory devices (for example queues that have consensus number 2).

The last section of our paper discusses the structure of the correctness proof and outlines a more abstract, two-stage proof in which the first stage investigates the algorithm and the resulting behavior of the higher level operation executions, and the second stage deals with abstract properties that are detached from the algorithm’s text. In our experience, this division of the correctness proof into two distinct parts has some marked benefit that justifies further investigation. Not only that the correction proof seems clearer in our eyes when its two parts are thus formally delineated, but the method helps to fashion better algorithms. In developing the algorithm there is a stage when the second part of the proof (its higher level, abstract part) is established but the algorithm itself is not yet completely determined; there are some features in the algorithm that can still be changed, some actions that can be omitted, and some data structures that can be reduced. When the designer of the algorithm has a clear and accessible aim in mind, namely when the higher level properties that the algorithm has to ensure are written down, then this process of improving the design of the algorithm follows a sure path. For example, in the process of designing the mailbox algorithm, once we understood that it suffices for the algorithm to satisfy the nine properties listed above in order to solve the mailbox problem we could play with changes and improvements knowing that as long as propositions remain correct we are on the right path.

References

[1] U. Abraham. Models for Concurrency. Gordon and Breach, 1999.

[2] U. Abraham. Logical Classification of Distributed Algorithms (Bakery Algorithms as an example). Theor. Comput. Sci. 412(25): 2724-2745 (2011)

[3] M. K. Aguilera, E. Gafni, L. Lamport. The Mailbox Problem (Extended Abstract). Disk 2008

[4] M. K. Aguilera, E. Gafni, L. Lamport. The Mailbox Problem. Distributed Computing, 23(2), pp. 113-134, October 2010.

[5] M. P. Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems, 13(1):124–149, Jan. 1991.

[6] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3):463-492, 1990.
[7] L. Lamport. A new solution of Dijkstra’s concurrent programming problem. Communications of the ACM, 17(8):453 - 455, Aug. 1974.

[8] L. Lamport. On Interprocess Communication, Part I: Basic formalism, Part II: Algorithms. Distributed Computing, Vol. 1, pp. 77 - 101. 1986.