**Special issue on NETYS 2019 manuscript No.**
(will be inserted by the editor)

# Recoverable Mutual Exclusion with Abortability

**Prasad Jayanti · Anup Joshi**

Received: date / Accepted: date

**Abstract** Recent advances in non-volatile main memory (NVRAM) technology have spurred research on designing algorithms that are resilient to process crashes. This paper is a fuller version of our conference paper [18], which presents the first Recoverable Mutual Exclusion (RME) algorithm that supports abortability. Our algorithm uses only the read, write, and CAS operations, which are commonly supported by multiprocessors. It satisfies FCFS and other standard properties.

Our algorithm is also adaptive. On DSM and Relaxed-CC multiprocessors, a process incurs $O(\min(k, \log n))$ RMRs in a passage and $O(f + \min(k, \log n))$ RMRs in an attempt, where $n$ is the number of processes that the algorithm is designed for, $k$ is the point contention of the passage or the attempt, and $f$ is the number of times that $p$ crashes during the attempt. On a Strict CC multiprocessor, the passage and attempt complexities are $O(n)$ and $O(f + n)$.

Attiya et al. proved that, with any mutual exclusion algorithm, a process incurs at least $\Omega(\log n)$ RMRs in a passage, if the algorithm uses only the read, write, and CAS operations [2]. This lower bound implies that the worst-case RMR complexity of our algorithm is optimal for the DSM and Relaxed CC multiprocessors.

**Keywords** concurrent algorithm, synchronization, mutual exclusion, recoverable algorithm, fault tolerance, non-volatile main memory, shared memory, multi-core algorithms

## 1 Introduction

Recent advances in non-volatile main memory (NVRAM) technology [11][25][29][30] have spurred research on designing algorithms that are resilient to process crashes. NVRAM is byte-addressable, so it replaces main memory, directly interfacing with the processor. This development is exciting because, if a process crashes and subsequently restarts, there is now hope that the process

The first author is grateful to the Frank family and Dartmouth College for their support through James Frank Family Professorship of Computer Science. The second author is grateful for the support from Dartmouth College.

Prasad Jayanti
Dartmouth College, Hanover NH 03755, USA
E-mail: prasad.jayanti@dartmouth.edu

Anup Joshi
Dartmouth College, Hanover NH 03755, USA
E-mail: anup.s.joshi.gr@dartmouth.edu

can somehow recover from the crash by consulting the contents of the NVRAM and resume its computation.

To leverage this advantage given by the NVRAM, there has been keen interest in reexamining the important distributed computing problems for which algorithms were designed in the past for the traditional (crash-free) model of an asynchronous shared memory multiprocessor. The goal is to design new algorithms that guarantee good properties even if processes crash at arbitrary points in the execution of the algorithm and subsequently restart and attempt to resume the execution of the algorithm. The challenge in designing such "recoverable" algorithms stems from the fact that when a process crashes, even though the shared variables that are stored in the NVRAM are unaffected, the crash wipes out the contents of the process' cache and CPU registers, including its program counter. So, when the process subsequently restarts, it can't have a precise knowledge of exactly where it crashed. For instance, if the last instruction that a process executes before a crash is a compare&swap (CAS) on a shared variable $X$, when it subsequently restarts, it can't tell whether the crash occurred just before or just after executing the CAS instruction and, if it did crash after the CAS, it won't know the response of the CAS (because the crash wipes out the register the CAS's response went into). The "recover" method, which a process is expected to execute when it restarts, has the arduous task of ensuring that the process can still somehow resume the execution of the algorithm seamlessly.

The mutual exclusion problem, formulated to enable multiple processes to share a resource that supports only one process at a time [6], has been thoroughly studied for over half a century for the traditional (crash-free) model, but its exploration for the crash-restart model is fairly recent. In the traditional version of the problem, each process $p$ is initially in the "remainder" section. When $p$ becomes interested in acquiring the resource, it executes the $\texttt{try}_p()$ method; and when this method completes, $p$ is in the "critical section" (CS). To give up the CS, $p$ invokes the $\texttt{exit}_p()$ method; and when this method completes, $p$ is back in the remainder section. An algorithm to this problem specifies the code for the try and exit methods so that at most one process is in the CS at any time and other desirable properties (such as starvation freedom, bounded exit, and First-Come-First-served, or FCFS) are also satisfied. Golab and Ramaraju were the first to reformulate this problem for the crash-restart model as *Recoverable Mutual Exclusion* (RME). In the RME problem, a process $p$ can crash at any time and subsequently restart [10]. If $p$ crashes while in try, CS, or exit, $p$'s cache and registers (aka local variables) are wiped out and $p$ returns to the remainder section (i.e., crash resets $p$'s program counter to its remainder section). When $p$ restarts after a crash, it is required to invoke a new method, named $\texttt{recover}_p()$, whose job is to "repair" the adverse effects of the crash and send $p$ to where it belongs. In particular, if $p$ crashed while in the CS, $\texttt{recover}_p()$ puts $p$ back in the CS (by returning IN_CS). On the other hand, if $p$ crashed while executing $\texttt{try}_p()$, $\texttt{recover}_p()$ has a choice—it can either roll $p$ back to the Remainder (by returning IN_REM) or put it in the CS (by returning IN_CS). Similarly, if $p$ crashed while executing $\texttt{exit}_p()$, $\texttt{recover}_p()$ has a choice of returning either IN_REM or IN_CS.

Golab and Ramaraju made a crucial observation that if $p$ crashes while in the CS, then no other process should be allowed into the CS until $p$ restarts and reenters the CS. This *Critical Section Reentry* (CSR) requirement was strengthed by Jayanti and Joshi's *Bounded CSR* requirement: if $p$ crashes while in the CS, when $p$ subsequently restarts and executes the recover method, the recover method should put $p$ back into the CS in a bounded number of its own steps [17]. There has been a flurry of research on RME algorithms in the recent years [3][5][8][9][10][14][15][17][18][19].

Orthogonal to this development of recoverable algorithms, motivated by the needs of real time systems and database systems, Scott and Scherer advocated the need for mutual exclusion algorithms to support the "abort" feature, whereby a process in the try section can quickly

quit the algorithm, if it so desires [27]. More specifically, if $p$ receives an abort signal from the environment while executing the try method, the try method should complete in a bounded number of $p$'s steps and either launch $p$ into the CS or send $p$ back to the remainder section. In the past two decades, there has been a lot of research on abortable mutual exclusion algorithms for the traditional (crash-free) model.

The possibility of crashes, together with the CSR requirement, renders abortability even more important in the crash-restart model, yet there have been no *abortable* recoverable algorithms until the conference publication of the algorithm in this submission [18]. There has since been one more algorithm, by Katzan and Morrison [19], and we will soon compare the two algorithms.

### 1.1 RMR complexity.

*Remote Memory Reference* (RMR) complexity is the standard complexity metric used for comparing mutual exclusion algorithms, so we explain it here. This metric is explained for the two prevalent models of multiprocessors—*Distributed Shared Memory* (DSM) and *Cache-Coherent* (CC) multiprocessors—as follows. In DSM, shared memory is partitioned into $n$ portions, one per process, and each shared variable resides in exactly one of the $n$ partitions. A step in which a process $p$ executes an instruction on a shared variable $X$ is considered an RMR if and only if $X$ is not in $p$'s partition of the shared memory.

In CC, the shared memory is remote to all processes, but every process has a local cache. A step in which a process $p$ executes an instruction *op* on a shared variable $X$ is considered an RMR if and only if *op* is *read* and $X$ is not in $p$'s cache, or *op* is any non-read operation (such as a *write* or *CAS*). If $p$ reads $X$ when $X$ is not present in $p$'s cache, $X$ is brought into $p$'s cache. If a process $q$ performs a non-read operation *op* while $X$ is in $p$'s cache, $X$'s copy in $p$'s cache is deleted in the *Strict CC model*, but in the *Relaxed CC model* it is deleted only if *op* changes $X$'s value. Thus, if $X$ is in $p$'s cache and $q$ performs an unsuccessful CAS on $X$, then $X$ continues to remain in $p$'s cache in the relaxed CC model.

A *passage* of a process $p$ starts when $p$ leaves the remainder section and completes at the earliest subsequent time when $p$ returns to the remainder (note that $p$ returns to the remainder either because of a crash or because of a normal return from try, exit or recover methods). An *attempt* of $p$ starts when $p$ leaves the remainder and completes at the earliest subsequent time when $p$ returns to the remainder "normally," i.e., not because of a crash. Note that each attempt includes one or more passages.

The *RMR complexity of a passage* (respectively, *attempt*) of a process $p$ is the number of RMRs that $p$ incurs in that passage (respectively, attempt).

### 1.2 Adaptive complexity.

A process is *active* if it is in the CS, or executing the try, exit, or recover methods, or crashed while in try, CS, exit, or recover and has not subsequently invoked the recover method. The *point contention* at any time $t$ is the number of active processes at $t$. The point contention of a passage (respectively, attempt) is the maximum point contention at any time in that passage (respectively, attempt). An algorithm is *adaptive* if the RMR complexity $r$ of each passage (or attempt) of a process $p$ is a function of that passage's (or attempt's) point contention $k$ such that $r = O(1)$ if $k = O(1)$.

### 1.3 Our contribution.

We present the first abortable RME algorithm. Our algorithm is based on the ideas underlying two earlier CAS-based algorithms—one that is recoverable but not abortable [17] and another that is abortable but not recoverable [13]. Our algorithm uses only the read, write, and CAS operations, which are commonly supported by multiprocessors. It satisfies FCFS and other standard properties (starvation-freedom, bounded exit, bounded CSR, and bounded abort). The algorithm's space complexity—the number of words of memory used—is $O(n)$.

Our algorithm is also adaptive. On DSM and Relaxed CC multiprocessors, a process $p$ incurs $O(\min(k, \log n))$ RMRs in a passage and $O(f + \min(k, \log n))$ RMRs in an attempt, where $n$ is the number of processes that the algorithm is designed for, $k$ is the point contention of the passage or the attempt, and $f$ is the number of times that $p$ crashes during the attempt. On a Strict CC multiprocessor, the passage and attempt complexities are $O(n)$ and $O(f + n)$.

Attiya et al. proved that, with any mutual exclusion algorithm (even if the algorithm does not have to satisfy recoverability or abortability), a process incurs at least $\Omega(\log n)$ RMRs in a passage, if the algorithm uses only the read, write, and CAS operations [2]. This lower bound implies that the worst-case RMR complexity of our algorithm is optimal for the DSM and Relaxed CC multiprocessors.

### 1.4 Comparison to Katzan and Morrison's algorithm.

To the best of our knowledge, there is only one other abortable RME algorithm, published recently by Katzan and Morrison [19]. They achieve sublogarithmic complexity: a process incurs at most $O(\min(k, \log n/\log\log n)$ RMRs in a passage and $O(f + \min(k, \log n/\log\log n)$ in an attempt. Furthermore, they achieve these bounds for even the Strict CC multiprocessor.

On the other hand, our work has the following merits. Unlike the CAS instruction employed in our algorithm, the fetch&add instruction, which their algorithm employs to beat Attiya et al's lower bound and achieve sublogarithmic complexity, is not commonly supported by current machines. Their algorithm does not satisfy FCFS and has a higher space complexity of $O(n \log^2 n/\log\log n)$. Their algorithm is stated to satisfy starvation-freedom if the total number of crashes in the run is finite. In contrast, our algorithm guarantees that each attempt completes even in the face of infinitely many crashes in the run, provided that there are only finitely many crashes during each attempt.

Finally, Katzan and Morrison correctly point out a shortcoming in our conference paper: our algorithm there admits starvation if there are infinitely many aborts in a run. The algorithm in this submission has been revised to eliminate this shortcoming.

### 1.5 Related Research.

All of the works on RME prior to the conference version of our paper [18] has focused on designing algorithms that do not provide abortability as a capability. Golab and Ramaraju [10] formalized the RME problem and designed several algorithms by adapting traditional mutual exclusion algorithms. Ramaraju [24], Jayanti and Joshi [17], and Jayanti et al. [14] designed RME algorithms that support the First-Come-First-Served property [20]. Golab and Hendler [8] presented an algorithm that has sub-logarithmic RMR complexity on CC machines. Jayanti et al. [15] presented a unified algorithm that has a sub-logarithmic RMR complexity on both CC and DSM machines. In another work, Golab and Hendler [9] presented an algorithm that has the ideal $O(1)$ passage complexity, but this result assumes that *all* processes in the system

crash *simultaneously*. Recently, Dhoked and Mittal [5] present an RME algorithm whose RMR complexity adapts to the number of crashes, and Chan and Woelfel [3] present an algorithm which has an O(1) amortized RMR complexity. Recently Katzan and Morrison [19] gave an abortable RME algorithm that incurs sub-logarithmic RMR on CC and DSM machines.

When it comes to abortability for classical mutual exclusion problem, Scott [26] and Scott and Scherer [28] designed abortable algorithms that build on the queue-based algorithms [4][22]. Jayanti [13] designed an algorithm based on read, write, and comparison primitives having $O(\log n)$ RMR complexity which is also optimal [2]. Lee [21] designed an algorithm for CC machines that uses the Fetch-and-Add and Fetch-and-Store primitives. Alon and Morrison [1] designed an algorithm for CC machines that has a sub-logarithmic RMR complexity and uses the read, write, Fetch-And-Store, and comparison primitives. Recently, Jayanti and Jayanti [16] designed an algorithm for the CC and DSM machines that has a constant amortized RMR complexity and uses the read, write, and Fetch-And-Store primitives. While the works mentioned so far have been deterministic algorithms, randomized versions of classical mutual exclusion with abortability exist. Pareek and Woelfel [23] give a sublogarithmic RMR complexity randomized algorithm and Giakkoupis and Woelfel [7] give an $O(1)$ expected amortized RMR complexity randomized algorithm.

## 2 Modeling an Abortable RME Algorithm and its runs

**Definition 1 (Abortable RME algorithm)** An *Abortable RME algorithm* is a tuple $(\mathcal{P}, \mathcal{X}, \mathit{Vals}, \mathcal{F}, \mathit{OP}, \Delta, \mathcal{M})$, where

- $\mathcal{P}$ is a set of processes. Each process $p \in \mathcal{P}$ has a set of registers, including a *program counter*, denoted $PC_p$, which points to an instruction in $p$'s code.
- $\mathcal{X}$ is a set of variables, which includes a Boolean variable AbortSignal[$p$], for each $p \in \mathcal{P}$. No process except $p$ can invoke any operation on AbortSignal[$p$], and $p$ can only invoke a read operation on AbortSignal[$p$].
  Intuitively, the "environment" sets AbortSignal[$p$] to *true* when it wishes to communicate to $p$ that it should abort its attempt to acquire the CS and return to the Remainder.
- *Vals* is a set of values (that each variable in $\mathcal{X}$ can possibly take on). For example, on a 64-bit machine, *Vals* would be the set of all 64-bit integers.
- $\mathcal{F}$ is a function that assigns a value from *Vals* to each variable in $\mathcal{X}$. For all $X \in \mathcal{X}$, $\mathcal{F}(X)$ is $X$'s *initial value*.
- *OP* is a set of operations that each variable in $\mathcal{X} - \{\text{AbortSignal}[p] \mid p \in \mathcal{P}\}$ supports.
  For the algorithm in this paper, $\mathit{OP} = \{\mathit{read}, \mathit{write}, \mathit{CAS}\}$, where $\text{CAS}(X, r, s)$, when executed by a process $p$ (and $X$ is a variable and $r$, $s$ are $p$'s registers), compares the values of $X$ and $r$; if they are equal, the operation writes in $X$ the value in $s$ and returns *true*; otherwise, the operation returns *false*, leaving $X$ unchanged.
- $\Delta$ is a partition of $\mathcal{X}$ into $|\mathcal{P}|$ sets, named $\Delta(p)$, for each $p \in \mathcal{P}$. Intuitively, $\Delta(p)$ is the set of variables that reside locally at process $p$'s partition on a DSM machine, but has no relevance on a CC machine.
- $\mathcal{M}$ is a set of methods, which includes three methods per process $p \in \mathcal{P}$, named $\texttt{try}_p()$, $\texttt{exit}_p()$, and $\texttt{recover}_p()$, such that:
  - In any instruction of any method, at most one operation is performed and it is performed on a single variable from $\mathcal{X}$.
  - The methods $\texttt{try}_p()$ and $\texttt{recover}_p()$ return a value from $\{\text{IN\_CS}, \text{IN\_REM}\}$, and $\texttt{exit}_p()$ has no return value.

– None of $\texttt{try}_p()$, $\texttt{exit}_p()$, or $\texttt{recover}_p()$ calls itself or the other two. (This assumption simplifies the model, but is not limiting in any way because it does not preclude the use of helper methods each of which can call itself or the other helper methods.)

For each process $p \in \mathcal{P}$, we model $p$'s code outside of the methods in $\mathcal{M}$ to consist of two disjoint sections, named $\texttt{remainder}_p()$ and $\texttt{cs}_p()$. Furthermore, we introduce the following *abstract* variables, which are not in $\mathcal{X}$ and not accessed by the methods in $\mathcal{M}$, but are helpful in defining the problem.

– $\mathit{status}_p \in \{\mathit{good}, \mathit{recover\text{-}from\text{-}try}, \mathit{recover\text{-}from\text{-}cs}, \mathit{recover\text{-}from\text{-}exit}, \mathit{recover\text{-}from\text{-}rem}\}$. Informally, $\mathit{status}_p$ models $p$'s "recovery status". If $\mathit{status}_p \neq \mathit{good}$, it means that either $p$ has crashed and not yet restarted or $p$ has restarted and invoked $\texttt{recover}_p()$ but has not yet completed $\texttt{recover}_p()$. The value of $\mathit{status}_p$ reveals the section of code where $p$ most recently crashed.
– $\textsc{Cache}_p$ holds a set of pairs of the form $(X, v)$, where $X \in \mathcal{X}$ and $v \in \mathit{Vals}$. Informally, if $(X, v)$ is present in the cache, $X$ is in $p$'s cache and $v$ is its current value. This abstract variable helps define what operations count as *remote memory references* (RMR) on CC machines.

**Definition 2 (State, Configuration, Initial Configuration)**

– A *state* of a process $p$ is a function that assigns a value to each of $p$'s registers, including $PC_p$, and a value to each of $\mathit{status}_p$, $\textsc{AbortSignal}[p]$, and $\textsc{Cache}_p$.
– A *configuration* is a function that assigns a state to each process in $\mathcal{P}$ and a value to each variable in $\mathcal{X}$. (Intuitively, a configuration is a snapshot of the states of processes and values of variables at a point in time.)
– An *initial configuration* is a configuration where, for each $p \in \mathcal{P}$, $PC_p = \texttt{remainder}_p()$, $\mathit{status}_p = \mathit{good}$, $\textsc{AbortSignal}[p] = \mathit{false}$, and $\textsc{Cache}_p = \emptyset$; and, for each $X \in \mathcal{X}$, $X = \mathcal{F}(X)$.

**Definition 3 (Run)** A *run* is a finite sequence $C_0, \alpha_1, C_1, \alpha_2, C_2, \ldots \alpha_k, C_k$, or an infinite sequence $C_0, \alpha_1, C_1, \alpha_2, C_2, \ldots$ such that:

1. $C_0$ is an initial configuration and, for each $i$, $C_i$ is a configuration and $\alpha_i$ is either $(p, \mathit{normal})$ or $(p, \mathit{crash})$, for some $p \in \mathcal{P}$.
   We call each triple $(C_{i-1}, \alpha_i, C_i)$ a *step*; it is a *normal step of* $p$ if $\alpha_i = (p, \mathit{normal})$, and a *crash step of* $p$ if $\alpha_i = (p, \mathit{crash})$.
2. For each normal step $(C_{i-1}, (p, \mathit{normal}), C_i)$, $C_i$ is the configuration that results when $p$ executes an enabled instruction of its code, explained as follows:
   – If $PC_p = \texttt{remainder}_p()$ and $\mathit{status}_p = \mathit{good}$ in $C_{i-1}$, then $p$ invokes either $\texttt{try}_p()$ or $\texttt{recover}_p()$.
   – If $PC_p = \texttt{remainder}_p()$ and $\mathit{status}_p \neq \mathit{good}$ in $C_{i-1}$, then $p$ invokes $\texttt{recover}_p()$.
   – If $PC_p = \texttt{cs}_p()$, then $p$ invokes $\texttt{exit}_p()$.
   – Otherwise, $p$ executes the instruction that $PC_p$ points to in $C_{i-1}$.
   If this instruction returns IN_CS (resp., IN_REM), $PC_p$ is set to $\texttt{cs}_p()$ (resp., $\texttt{remainder}_p()$).
   If the instruction causes $p$ to return from $\texttt{recover}_p()$, $\mathit{status}_p$ is set to *good* in $C_i$.
   If $p$ performs a read on $X$ and $X$ is not present in $\textsc{Cache}_p$ in $C_{i-1}$, then $(X, v)$ is inserted in $\textsc{Cache}_p$, where $v$ is $X$'s value in $C_{i-1}$.
   In the Strict-CC model, if $p$ performs a non-read operation on $X$, $X$ is removed from $\textsc{Cache}_q$, for all $q \in \mathcal{P}$.
   In the Relaxed-CC model, if $p$ performs a non-read operation on $X$ that changes $X$'s value, $X$ is removed from $\textsc{Cache}_q$, for all $q \in \mathcal{P}$.
3. For each crash step $(C_{i-1}, (p, \mathit{crash}), C_i)$, we have:

- In $C_i$, $PC_p$ is set to $\texttt{remainder}_p()$ and all other registers of $p$ are set to arbitrary values, and $\textsc{Cache}_p$ is set to $\emptyset$.
- If $status_p \neq good$ in $C_{i-1}$, then $status_p$ remains unchanged in $C_i$. Otherwise, if (in $C_{i-1}$) $p$ is in $\texttt{try}_p()$ (respectively, $\texttt{cs}_p()$, $\texttt{exit}_p()$, or $\texttt{recover}_p()$), then $status_p$ is set in $C_i$ to *recover-from-try* (respectively, *recover-from-cs*, *recover-from-exit*, or *recover-from-rem*).

Liveness of the algorithm, which guarantees that processes don't wait forever, can be realized only if the underlying model assures that every crashed process eventually restarts, no process stays in the CS forever, and no process permanently ceases to take steps when it is outside the Remainder section. Hence, "fair" runs of the algorithm where these assurances are kept are of interest, as captured by the next definition.

**Definition 4 (Fair run)** A run $R = C_0, \alpha_1, C_1, \alpha_2, C_2, \ldots$ is *fair* if and only if either $R$ is finite or, for all configurations $C_i$ and for all processes $p \in \mathcal{P}$, the following condition is satisfied: unless $PC_p = \texttt{remainder}_p()$ and $status_p = good$ in $C_i$, $p$ has a step in the suffix of $R$ from $C_i$.

**Definition 5 (Passage and Attempt)**

- A *passage* of a process $p$ is a contiguous sequence $\sigma$ of steps in a run such that $p$ leaves $\texttt{remainder}_p()$ in the first step of $\sigma$ and the last step of $\sigma$ is the earliest subsequent step in the run where $p$ reenters $\texttt{remainder}_p()$ (either because $p$ crashes or because $p$'s method returns IN_REM).
- An *attempt* of a process $p$ is a maximal contiguous sequence $\sigma$ of steps in a run such that $p$ leaves $\texttt{remainder}_p()$ in the first step of $\sigma$ with $status_p = good$ and the last step of $\sigma$ is the earliest subsequent normal step in the run that causes $p$ to reenter $\texttt{remainder}_p()$ (which would be a return from $\texttt{exit}_p$, or a return of IN_REM from $\texttt{try}_p$ or $\texttt{recover}_p$).

**Definition 6 (RMR)**

- A step of $p$ is an *RMR on a DSM machine* if and only if it is a normal step in which $p$ performs an operation on some variable that is not in $\Delta(p)$.
- A step of $p$ is an *RMR on a Strict or Relaxed CC machine* if and only if it is a normal step in which $p$ performs a non-read operation, or $p$ reads some variable that is not present in $p$'s cache.

**Definition 7 (Active)** A process $p$ is *active* in a configuration $C$ if the condition $(PC_p \neq \texttt{remainder}_p()) \vee (status_p \neq good)$ holds in $C$.

**Definition 8 (Point contention)** The *point contention* at a configuration $C$ is the number of active processes in $C$.

## 3 Properties of an abortable RME algorithm

We state the properties required of an abortable RME algorithm which, for easy comprenhensibility, we have divided into four categories: basic safety, responsiveness, liveness, and fairness.

**Basic safety properties**

P1. Mutual Exclusion: At most one process is in the CS in any configuration of any run.
P2. Critical Section Reentry (CSR) [10]: In any run, if a process $p$ crashes while in the CS, no other process enters the CS until $p$ subsequently reenters the CS.

P3. No Trivial Aborts: In any run, if AbortSignal[$p$] is *false* when a process $p$ invokes $\texttt{try}_p()$ and it remains *false* during the execution of $\texttt{try}_p()$, then $\texttt{try}_p()$ does not return IN_REM.

**Responsiveness properties**

Once a process leaves the CS, it should be able to return to the remainder section without having to wait on other processes, as captured by the next property.

P3. Bounded Exit: There is an integer $b$ such that if in any run any process $p$ invokes and executes $\texttt{exit}_p()$ without crashing, the method completes in at most $b$ steps of $p$.

The next property formalizes the requirement that, if the environment signals a waiting process to abort (and maintains that signal so that it is not missed), then the process should be able to quit the try method (i.e., either return to the remainder or capture the CS) without being obstructed by others.

P4. Bounded Abort [13]: There is an integer $b$ such that if at any point in any run a process $p$ is in $\texttt{try}_p()$ or $\texttt{recover}_p()$ with $status_p$ = *recover-from-try*, and from that point on AbortSignal[$p$] stays *true* and $p$ executes steps without crashing, then $\texttt{try}_p()$ or $\texttt{recover}_p()$ returns in at most $b$ steps of $p$.

A process $p$ finds itself in the remainder section either because it crashed while executing the algorithm or because it returned normally from the algorithm. In the former case, when $p$ restarts, it is *required* to execute $\texttt{recover}_p()$, but in the latter case, $p$ has a choice—it can execute either $\texttt{try}_p()$ or $\texttt{recover}_p()$. If $p$ is unsure whether it is restarting from a crashed state, it can harmlessly "probe" by executing $\texttt{recover}_p()$. However, if $p$ executes $\texttt{recover}_p()$ in the latter case, for efficiency we require $\texttt{recover}_p()$ to complete quickly (and return IN_REM).

P5. Fast Probing: There is an absolute constant $b$, i.e., a constant independent of $|\mathcal{P}|$, such that if in any run any process $p$ executes $\texttt{recover}_p()$ without crashing and with $status_p \in \{$*good*, *recover-from-rem*$\}$, the method completes in at most $b$ steps of $p$.

If $p$ crashes while in the CS, the CSR property stated earlier prohibits others from entering the CS until $p$ reenters the CS. Therefore, when $p$ restarts and executes $\texttt{recover}_p()$, we would want $p$ to be able to complete $\texttt{recover}_p()$ (and return IN_CS) without being obstructed by other processes [17]. Similarly, when $p$ executes $\texttt{recover}_p()$ following a crash in the exit section, $p$ should be able to complete $\texttt{recover}_p()$ (returning IN_CS or IN_REM) without having to wait on others. On the other hand, if $p$ crashes while executing $\texttt{try}_p()$, the execution of $\texttt{recover}_p()$ upon restart has two options: either it gives up the attempt to acquire the CS and returns IN_REM or it tries once again to acquire the CS. In the latter case, waiting is unavoidable, but in the former case we require that $p$ completes $\texttt{recover}_p()$ without having to wait on others. The next property formalizes these requirements.

P6. Bounded Recovery: There is an integer $b$ such that if in any run any process $p$ executes $\texttt{recover}_p()$ without crashing and either with $status_p \in \{$*recover-from-cs*, *recover-from-exit*$\}$ or with $status_p$ = *recover-from-try* and the method returns IN_REM, the method completes in at most $b$ steps of $p$.

**Liveness property**

For the traditional mutual exclusion problem, the liveness condition is usually starvation-freedom, which states that if a process $p$ is in $\texttt{try}_p()$ at any point in a fair infinite run, it is in the CS at a later point. We adapt this definition to allow for aborts and crashes. To accommodate

aborting, we relax the phrase "it is in the CS at a later point" in the definition to "it returns from $\texttt{try}_p()$ at a later point." Furthermore, since a non-aborting waiting process cannot enter the CS if the process in the CS fails repeatedly (infinitely many times), we could require progress only when there are finitely many crashes:

> Starvation Freedom: In every fair infinite run in which there are only finitely many crash steps, if a process $p$ is in $\texttt{try}_p()$ in a configuration, $p$ subsequently returns from $\texttt{try}_p()$.

Our algorithm satisfies a stronger property that guarantees progress even when there are infinitely many crashes in the run, provided that there are only finitely crashes in each attempt of the run.

P7. Strong Starvation Freedom: In every fair infinite run in which every attempt contains only finitely many crash steps (counting the crash steps of all processes), if a process $p$ is in $\texttt{try}_p()$ in a configuration, $p$ subsequently returns from $\texttt{try}_p()$.

**Fairness property**

For the traditional mutual exclusion problem, a standard fairness property, known as First-Come-First-Served (FCFS) [20], informally states that if a process $p$ completes its request for the CS before $q$ begins its request, then $q$ does not enter the CS before $p$. Formally, $p$'s request for the CS consists of the first $b$ steps of $\texttt{try}_p()$, where $b$ is a bound that depends only on the algorithm. Thus, the formal definition of FCFS states that there is an integer $b$ such that if $p$ performs $b$ steps of $\texttt{try}_p()$ before $q$ invokes $\texttt{try}_q()$, then $q$ does not enter the CS before $p$.

To accommodate aborts, the condition is adapted to: "If $p$ completes the request before $q$ begins its request *and $p$ does not abort*, then $q$ won't enter the CS before $p$" [17]

When adapting this condition further to accommodate crashes, two interesting issues come up. First, $p$ might invoke $\texttt{try}_p()$, crash before performing $b$ steps, restart and crash yet again before performing $b$ steps, and so on. Given this possibility, we deem $p$ to have completed its request before $q$ begins its request if $p$ performs at least $b$ steps between some two of its successive crashes (in $\texttt{try}_p()$ or in $\texttt{recover}_p()$ with $status_p$ = *recover-from-try*) before $q$ invokes $\texttt{try}_q()$. Second, suppose that $p$ completes its request before $q$ begins its request, and suppose further that $p$ does not receive the abort signal. Should $q$ be necessarily prohibited from entering the CS before $p$? Rather surprisingly, the answer is no. To understand why, suppose that $p$ crashes after having requested the CS. When $p$ restarts and executes $\texttt{recover}_p()$, there are two possibilities: (1) $p$ completes the recovery method in a bounded number of its own steps, thereby either returning to the remainder or capturing the CS, or (2) $p$ waits in the recovery method until eventually capturing the CS. The algorithm in this paper is designed to realize the first possibility, which is attractive because, in case the recovery sends $p$ to the remainder, $p$ can choose whether or not to execute the algorithm afresh. On the other hand, just as legitimately, the algorithm designer might prefer the second possibility, where even after the crash $p$ persists in its resolve to acquire the CS. Turning our attention back to FCFS, in the first possibility, since $p$ is assured of a bounded recovery, there is no reason to prohibit $q$ from entering the CS, but in the second possibility, $q$ should not be allowed to enter the CS ahead of $p$. The definition below incorporates all of these elements, where $RR'$ denotes a concatenation of sequences $R$ and $R'$.

P8. FCFS: There is an integer $b$ such that if (i) $R$ and $RR'$ are finite runs, (ii) $p$ starts an attempt $A$ in $R$ but does not complete it in $R$, (iii) $q$ starts an attempt $A'$ in $R$ and is in the CS in $A'$ at the end of $R$, (iv) before $q$ starts the attempt $A'$, $p$ performs at least $b$ normal steps in $A$ without any intervening crash steps of $p$ in $A$, and (v) $p$ has no crash steps in $R'$ and has at least $b$ steps in $R'$, then $p$ enters the CS or the remainder section in $A$ in $RR'$.

## 4 A key building block: the min-array object [12]

The design of a mutual exclusion algorithm requires a facility by which processes can quickly identify a most deserving (i.e., a highest priority or a longest waiting process) among the waiting processes that should be launched into the CS next. When an algorithm is restricted to using only the read, write, and CAS operations, Jayanti's *min-array construction* [12] has proved useful for this purpose in some earlier algorithms [13] [17]. Our algorithm is also based on the min-array object.

A min-array object $X$ of $n$ locations supports two operations: $X[p].\texttt{write}(v)$, which can only be executed by process $p \in \{1, 2, \ldots, n\}$, writes $v$ in $X[p]$; and $X.\texttt{findmin}()$ returns the minimum value among $X[1], X[2], \ldots, X[n]$. The construction in [12] presents a linearizable and wait-free implementation of this object using only the read, write, and CAS operations. The following properties of this implementation are what makes it useful for our algorithm:

- The implementation has adaptive and small worst-case step complexity. Specifically, a process $p$ completes $X.\texttt{findmin}()$ in $O(1)$ steps and $X[p].\texttt{write}(v)$ in $O(\min(k, \log n))$ steps, where $k$ is the maximum point contention during the execution of $X[p].\texttt{write}(v)$.
- Suppose that $p$ invokes $X[p].\texttt{write}(v)$ and crashes before completing the method; when it restarts, suppose that it invokes $X[p].\texttt{write}(v)$ once more and yet again crashes before completing the method. Suppose this pattern repeats $f$ times before $p$ invokes $X[p].\texttt{write}(v)$ and executes it to completion. Despite the many partial executions before the full execution, the implementation ensures that the $X[p].\texttt{write}(v)$ operation appears to take effect exactly once. Furthermore, the total number of $p$'s steps, over all of the partial executions and the final full execution, is $O(f + \min(k, \log n))$.
- Suppose that $p$ invokes $X[p].\texttt{write}(v)$ and crashes before completing it. When $p$ subsequently restarts, suppose that $p$ chooses to abandon that write operation and instead executes $X[p].\texttt{write}(v')$ to completion, for some $v' \neq v$. Then, the implementation guarantees that either $X[p].\texttt{write}(v)$ does not take effect and only $X[p].\texttt{write}(v')$ takes effect, or $X[p].\texttt{write}(v)$ takes effect before $X[p].\texttt{write}(v')$ takes effect.
- The implementation has $O(n)$ space complexity (i.e., uses only $O(n)$ memory words).

## 5 The Algorithm and its intuitive description

We present in Figure 1 our abortable RME algorithm for the set of processes $\mathcal{P} = \{1, 2, \ldots, n\}$. All the shared variables used by our algorithm are stored in NVRAM. Variables with a subscript of $p$ to their name are local to process $p$, and are stored in $p$'s registers or volatile memory. We begin by describing the role played by each of the shared variables used in the algorithm.

- Token is an unbounded positive integer. A process $p$ reads this variable at the beginning of $\texttt{try}_p()$ to obtain its token and then increments, thereby ensuring that processes that invoke the try method later will get a strictly bigger token.

- CSStatus and Seq: These two shared variables are used in conjunction, with Seq holding an unbounded integer and CSStatus holding a pair, which is either $(\mathit{true}, p)$ (for some $p \in \mathcal{P}$) or $(\mathit{false}, \text{Seq})$. If CSStatus $= (\mathit{true}, p)$, it means that $p$ owns the CS and, if CSStatus $= (\mathit{false}, \text{Seq})$, it means that no process owns the CS. If Seq has a value $s$ while $p$ is the CS, when exiting the CS $p$ increments Seq to $s + 1$ and writes $(0, s + 1)$ in CSStatus. As we explain later, this act is crucial to ensuring that no process will be made the owner of the CS after it has moved back to the remainder.

**Persistent variables (stored in NVRAM)**
REGISTRY[1 . . . |$\mathcal{P}$|] : A min-array; initially REGISTRY[$p$] = $(p, \infty)$, for all $p \in \mathcal{P}$.
CSSTATUS $\in \{0\} \times (\{0\} \cup \mathbb{N}^+) \cup \{1\} \times \mathcal{P}$; initially $(0, 1)$.
SEQ $\in \mathbb{N}$; initially 1.
$\forall p \in \mathcal{P}$, GO[$p$] $\in \mathbb{N}^+ \cup \{-1, 0\}$, initially $\perp$.
TOKEN $\in \mathbb{N}$, initially 1.

**1.** `Remainder Section`

**procedure** `try`$_p$():
**2.** $tok_p \leftarrow$ TOKEN
**3.** CAS(TOKEN, $tok_p, tok_p + 1$)
**4.** GO[$p$] $\leftarrow tok_p$
**5.** REGISTRY[$p$].`write`$((p, tok_p))$
**6.** `promote`$_p(false)$
**7.** **wait till** GO[$p$] $= 0 \vee$ ABORTSIGNAL[$p$]
**8.** **if** GO[$p$] $= 0$: **return** IN_CS
**9.** **return** `abort`$_p()$

**10.** `Critical Section`

**procedure** `exit`$_p$():
**11.** REGISTRY[$p$].`write`$((p, \infty))$
**12.** $s_p \leftarrow$ SEQ
**13.** SEQ $\leftarrow s_p + 1$
**14.** CSSTATUS $\leftarrow (0, s_p + 1)$
**15.** `promote`$_p(false)$
**16.** GO[$p$] $\leftarrow -1$

**procedure** `recover`$_p$():
**17.** **if** GO[$p$] $= -1$: **return** IN_REM
**18.** **return** `abort`$_p()$

**procedure** `abort`$_p$():
**19.** REGISTRY[$p$].`write`$((p, \infty))$
**20.** `promote`$_p(true)$
**21.** **if** CSSTATUS $= (1, p)$: **return** IN_CS
**22.** GO[$p$] $\leftarrow -1$
**23.** **return** IN_REM

**procedure** `promote`$_p$(**boolean** $flag_p$):
**24.** $(b_p, s_p) \leftarrow$ CSSTATUS; **if** $b_p = 1$: { $peer_p \leftarrow s_p$; **go to** Line **27** }
**25.** $(peer_p, tok_p) \leftarrow$ REGISTRY.`findmin`(); **if** $tok_p = \infty \wedge flag_p$: $peer_p \leftarrow p$ **else if** $tok_p = \infty$: **return**
**26.** **if** $\neg$CAS(CSSTATUS, $(0, s_p), (1, peer_p)$): **return**
**27.** $g_p \leftarrow$ GO[$peer_p$]; **if** $g_p \in \{-1, 0\}$: **return**
**28.** **if** CSSTATUS $\neq (1, peer_p)$: **return**
**29.** CAS(GO[$peer_p$], $g_p, 0$)

Fig. 1: Abortable RME Algorithm for CC and DSM machines. Code for process $p$.

– GO[$p$] has one of three values — $-1$, 0, or $p$'s token. The algorithm ensures that GO[$p$] $= -1$ whenever $p$ is in the remainder "normally", i.e., not because of a crash but because the try, exit, or recover method returned normally. If GO[$p$] $= 0$, it means that $p$ is made the owner of CS, hence $p$ has the permission to enter the CS. After $p$ obtains a token in `try`$_p()$, $p$ writes its token in GO[$p$] and, subsequently when $p$ must wait for its turn to enter the CS, it spins until either GO[$p$] turns 0 or it receives a signal to abort.

– REGISTRY is a min-array object [12] of $n$ locations. After $p$ obtains a token $t$ in $\texttt{try}_p()$, it announces its interest in capturing the CS by writing the pair $(p, t)$ in REGISTRY$[p]$, and when no longer interested, it removes the token by writing $(p, \infty)$ in REGISTRY$[p]$. The "less than" relation on pairs is defined as follows: $(p, t) < (p', t')$ if and only if $t < t'$ or $(t = t') \wedge (p < p')$.

Next we present an intuitive understanding of the algorithm, explaining the lines of code and, more importantly, drawing attention to potential race conditions and how the algorithm overcomes them.

**Understanding $\texttt{try}_p()$**

After a process $p$ invokes $\texttt{try}_p()$, it reads and then attempts to increments TOKEN (Lines **2**, **3**). The attempt to increment serves two purposes. First, if a different process $q$ invokes $\texttt{try}_q()$ later, it gets a strictly larger token, which helps realize FCFS. Second, if $p$ were to abort its curent attempt $A$, it will obtain a strictly larger token in its next attempt $A'$, which, as we will see, helps ensure that any process $q$ that might attempt to release $p$ from its busy-wait in the attempt $A$ will not accidentally release $p$ from its busy-wait in the attempt $A'$. Process $p$ writes its token in GO$[p]$ (Line **4**), where it will later busy-wait until some process changes GO$[p]$ to 0, and then announces its interest in the CS by changing REGISTRY$[p]$ from $(p, \infty)$ to $(p,$ its token) (Line **5**). It then calls the $\texttt{promote}_p()$ procedure, which is crucial to ensuring livelock-freedom (Line **6**).

**Understanding $\texttt{promote}_p()$**

The $\texttt{promote}_p()$ procedure's purpose is to push a waiting process into the CS, if the CS is unoccupied. To this end, $p$ reads CSSTATUS (Line **24**). If it finds that the CS is already owned (i.e., $b_p = 1$), since it is possible that the owner $peer_p$ is still busywaiting unaware of its ownership, $p$ jumps to Line **27**, where the code to release $peer_p$ starts. On the other hand, if the CS is unoccupied (i.e., $b_p = 0$), it executes Line **25** to find out the process that has the smallest token in the REGISTRY, i.e., the process $peer_p$ that has been waiting the longest. Since $\texttt{promote}_p()$ is called from $p$'s Line **6**, at which point REGISTRY$[p]$ has a finite token number for $p$, at Line **25** we have $tok_p \neq \infty$. So, $p$ proceeds to Line **26**, where it attempts to launch $peer_p$ into the CS. If $p$'s CAS fails, it means that someone else must have succeeded in launching a process into the CS between $p$'s Line **24** and Line **26**; in this case $p$ has no further role to play, so it returns from the procedure. On the other hand, if $p$'s CAS succeeds, which means that $peer_p$ has been made the CS owner, $p$ has a responsibility to release $peer_p$ from its busywait, i.e., $p$ must write 0 in GO$[peer_p]$. However, there is potential for a nasty race condition here, as explained by the following scenario: some process different from $p$ releases $peer_p$ from its busywait; $peer_p$ enters the CS and then exits to the remainder; some other process $q$ is now in the CS; $peer_p$ executes the try method once more and proceeds up to the point of busy-waiting. Recall that $p$ is poised to write 0 in GO$[peer_p]$. If $p$ does this writing, $peer_p$ will be released from its busywait, so $peer_p$ proceeds to the CS, where $q$ is already present. So, mutual exclusion is violated! Our algorithm averts this disaster by exploiting the fact that, while $peer_p$ busywaits, GO$[peer_p]$'s value is never the same between different attempts of $peer_p$. Specifically, $p$ reads GO$[peer_p]$ (Line **27**); if $g_p$ is $-1$ or 0, it means that $peer_p$ is not busywaiting, so $p$ has no role to play, hence it returns. If things have moved on and $peer_p$ no longer owns the CS, then too $p$ has no role to play, hence it returns (Line **28**). Otherwise, there are two possibilities: either GO$[peer_p]$ is still $g_p$ or it has changed. In the former case, $peer_p$ must be busywaiting, so it is imperative that $p$ takes the responsibility to release $peer_p$ (by changing GO$[peer_p]$ to 0). In the latter case, $peer_p$ requires no help from $p$, so $p$ must not change GO$[peer_p]$ (in order to avoid the race condition described above). This is precisely what the CAS at Line **29** accomplishes.

**The rest of $\texttt{try}_p()$**

Upon returning from $\texttt{promote}_p()$, $p$ busywaits until it reads a 0 in $\text{GO}[p]$ or it receives a request to abort (Line **7**). If $p$ reads a 0 in $\text{GO}[p]$, $p$ infers that it owns the CS, so $\texttt{try}_p()$ returns IN_CS (Line **8**). If $p$ receives a request to abort, it calls $\texttt{abort}_p()$ (Line **9**), which we describe next.

**Understanding $\texttt{abort}_p()$**

To abort, $p$ writes $(p, \infty)$ to make it known to all that it has no interest in capturing the CS (Line **19**). If any process will invoke the promote procedure after this point, it will not find $p$ in REGISTRY, so it will not attempt to launch $p$ into the CS. Does this mean that $p$ can now return to the remainder section? The answer is a thundering no because there are two nasty race conditions that need to be overcome.

First, it is possible that, before $p$ performed Line **19**, some process $q$ performed its Line **25** to find $p$ in REGISTRY, and then successfully launched $p$ into the CS (by writing $(1, p)$ in CSSTATUS). Taking care of this scenario is easy: $p$ can read CSSTATUS and if $p$ finds that it owns the CS, it can abort by simply returning IN_CS.

The second potential race is more subtle and harder to overcome. As in the earlier scenario, suppose that, before $p$ performed Line **19**, some process $q$ performed its Line **25** to find $p$ in REGISTRY (i.e., $peer_q = p$). Furthermore, suppose that $q$ is now at Line **26** and CSSTATUS $= (0, s_q)$. So, after performing Line **19**, if $p$ naively returns to the remainder and then $q$ performs Line **26**, we would be in a situation where $p$ has been made the CS owner after it was back in the remainder!

To overcome the above two race conditions, $p$ calls $\texttt{promote}_p(\mathit{true})$ (Line **20**).

The parameter *true* conveys that the call is made by $p$ while aborting, and has the following impact on how $p$ executes $\texttt{promote}_p()$: if $p$ finds the CS to be unoccupied at Line **24** and finds REGISTRY to be empty at Line **25**, to preempt the second race condition discussed above (where some process $q$ is poised to launch $p$ into the CS), $p$ will attempt to launch itself into the CS (by setting $peer_p$ to $p$ at Line **25** and attempting to change CSSTATUS to $(1, peer_p)$). The key insight is that, after $p$ performs the CAS at Line **26**, only two possibilities remain: either $p$ is already launched into the CS (i.e., CSSTATUS $= (1, p)$) or it is guaranteed that no process will launch $p$ into the CS. In the former case, $\texttt{abort}_p()$ returns IN_CS at Line **21**; and in the latter case, since it is safe for $p$ to return to the remainder, $\texttt{abort}_p()$ returns IN_REM at Line **23** after setting $\text{GO}[p]$ to $-1$ at Line **22** (in order to respect the earlier mentioned invariant that $\text{GO}[p] = -1$ whenever $p$ returns to the remainder normally).

**Understanding $\texttt{exit}_p()$**

There are two routes by which $p$ might enter the CS. One is the "normal" route where $p$ executes $\texttt{try}_p()$ without aborting or crashing, and $\texttt{try}_p()$ returns IN_CS, thereby sending $p$ to the CS. The second route is where $p$ receives an abort signal, calls at Line **9** $\texttt{abort}_p()$, which returns IN_CS at Line **21**, causing $\texttt{try}_p()$ also to return IN_CS at Line **9**. When $p$ is in the CS, $p$'s announcement in $\text{REGISTRY}[p]$ (made at Line **5**), would no longer be there if it entered the CS by the second route (because of Line **19**), but it would still be there if it entered the CS by the first route. So, when $p$ exits the CS, it removes its announcement in $\text{REGISTRY}[p]$ (Line **11**). It then increments the number in SEQ and gives up its ownership of the CS by changing CSSTATUS from $(1, p)$ to $(0, \text{SEQ})$ (Lines **12**, **13**, **14**). To launch a waiting process, if any, into the just vacated CS, $p$ then executes $\texttt{promote}_p()$ (Line **15**), and returns to the remainder after setting $\text{GO}[p]$ to $-1$ at Line **16** (in order to respect the earlier mentioned invariant that $\text{GO}[p] = -1$ whenever $p$ returns to the remainder normally).

**Understanding $\texttt{recover}_p()$**

Process $p$ executes $\texttt{recover}_p()$ when it restarts after a crash. If GO$[p]$ has $-1$, $p$ infers that either $\texttt{recover}_p()$ was called when $status_p = good$ or the most recent crash had occured early in $\texttt{try}_p()$, so $\texttt{recover}_p()$ simply sends $p$ back to the remainder (Line **17**). Otherwise, $\texttt{recover}_p()$ simply calls $\texttt{abort}_p()$ (Line **17**), which does the needful. In particular, if $p$ was in the CS at the most recent crash, then CSSTATUS would have $(1, p)$, which causes $\texttt{abort}_p()$ to send $p$ back to the CS. Otherwise, $\texttt{abort}_p()$ extricates $p$ from the algorithm, sending it either to the CS or to the remainder.

## 6 The invariant

Figure 2 presents the invariant satisfied by the Abortable RME algorithm given in Figure 1. We have written the 13 statements comprising the invariant with the following conventions. All statements about process $p$ are universally quantified, i.e., $\forall p \in \mathcal{P}$ is implicit (these are Statements 3 through 11, and Statement 13). The program counter for a process $p$, i.e., $PC_p$, can take any of the values from the set $[\mathbf{1}, \mathbf{29}]$. However, when a call to procedure $\texttt{promote}_p()$ is made by $p$ and $p$ is executing one of the steps from Lines **24**-**29**, for clearly conveying where the call was made from, we prefix the value of $PC_p$ with the line number from where $\texttt{promote}_p()$ was called, along with the scope resolution operator from C++, namely, "::". Thus, $PC_p = \mathbf{6}::\mathbf{27}$ means $p$ called $\texttt{promote}_p()$ from Line **6** and is now executing Line **27** in that call. Sometimes, in the interest of brevity, we use the range operator, i.e., $[a, b]$, to convey something more than just saying the range of values from $a$ to $b$ (inclusive). That is, if $PC_p \in [\mathbf{6}, \mathbf{8}]$, we also mean that $PC_p$ could take on values from $[\mathbf{6}::\mathbf{24}, \mathbf{6}::\mathbf{29}]$ because there is a call to $\texttt{promote}_p()$ at Line **6**. Similarly, the range $[\mathbf{5}, \mathbf{6}]$ includes Line **5** as well as the lines in the range $[\mathbf{6}::\mathbf{24}, \mathbf{6}::\mathbf{29}]$ because, again, there is a call to $\texttt{promote}_p()$ at Line **6**.

The lemma below asserts that the invariant is correct. Its proof is presented in Appendix A.

**Lemma 1** *The algorithm in Figure 1 satisfies the invariant in Figure 2 (i.e., the conjunction of all the conditions stated in Figure 2 holds in every configuration of every run).*

## 7 Proof of the properties and the main theorem

Using the invariant, we now prove that the algorithm satisfies all of the properties stated in Section 3 and has adaptive and worst-case logarithmic RMR complexity on DSM and Relaxed-CC machines.

**Lemma 2 (Mutual Exclusion)** *At most one process is in the CS in any configuration of any run.*

*Proof* Assume to the contrary that there is a configuration $C$ where two distinct processes $p$ and $q$ are in the CS, i.e., $PC_p = PC_q = \mathbf{10}$. By Condition 5, CSSTATUS $= (1, p)$ and CSSTATUS $= (1, q)$ in $C$, which means CSSTATUS has two different values in the same configuration, a contradiction.

**Lemma 3 (Critical Section Reentry)** *In any run, if a process $p$ crashes while in the CS, no other process enters the CS until $p$ subsequently reenters the CS.*

*Proof* Suppose that $p$ crashes while in the CS, thereby moving from the CS to the remainder section with $status_p$ set to *recover-from-cs*. The value of $status_p$ remains *recover-from-cs* until $p$ subsequently restarts and executes $\texttt{recover}_p()$ to completion. It follows from Condition 5 of

**Conditions:**

1. $\textsc{Token} \geq 1$
2. $(\text{CSStatus} = (0, \textsc{Seq})) \vee (\exists q \in \mathcal{P},\ \text{CSStatus} = (1, q))$
3. $(-1 \leq \textsc{Go}[p] < \textsc{Token}) \wedge (PC_p = \mathbf{5} \Rightarrow \textsc{Go}[p] = tok_p) \wedge (PC_p \in [\mathbf{6}, \mathbf{8}] \Rightarrow \textsc{Go}[p] \in \{0, tok_p\})$
   $\wedge\ (PC_p \in \{\mathbf{9}\text{-}\mathbf{16}, \mathbf{18}\text{-}\mathbf{22}, \mathbf{24}\text{-}\mathbf{29}\} \Rightarrow \textsc{Go}[p] \neq -1)$
   $\wedge\ ((PC_p \in \{\mathbf{2}\text{-}\mathbf{4}, \mathbf{23}\} \vee (PC_p \in \{\mathbf{1}, \mathbf{17}\} \wedge status_p \in \{good, recover\text{-}from\text{-}rem\})) \Rightarrow \textsc{Go}[p] = -1)$
4. $(\exists t \in [1, \textsc{Token} - 1] \cup \{\infty\}, \textsc{Registry}[p] = (p, t))$
   $\wedge\ (PC_p \in [\mathbf{6}, \mathbf{8}] \Rightarrow \textsc{Registry}[p] = (p, tok_p))$
   $\wedge\ ((PC_p \in \{\mathbf{5}, \mathbf{12}\text{-}\mathbf{16}, \mathbf{20}\text{-}\mathbf{22}\} \vee \textsc{Go}[p] = -1) \Rightarrow \textsc{Registry}[p] = (p, \infty))$
5. $(((PC_p \in [\mathbf{6}, \mathbf{8}] \wedge \textsc{Go}[p] = 0) \vee PC_p \in [\mathbf{10}, \mathbf{14}] \vee status_p = recover\text{-}from\text{-}cs) \Rightarrow \text{CSStatus} = (1, p))$
   $\wedge\ ((PC_p \in \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}] \vee \textsc{Go}[p] = -1) \Rightarrow \text{CSStatus} \neq (1, p))$
6. This condition states what values local variables of process $p$ take on.
   $(PC_p = \mathbf{3} \Rightarrow 1 \leq tok_p \leq \textsc{Token}) \wedge (PC_p \in [\mathbf{4}, \mathbf{8}] \Rightarrow 1 \leq tok_p < \textsc{Token})$
   $\wedge\ (PC_p = \mathbf{13} \Rightarrow s_p = \textsc{Seq}) \wedge (PC_p = \mathbf{14} \Rightarrow s_p = \textsc{Seq} - 1)$
   $\wedge\ (PC_p \in [\mathbf{6}..\mathbf{24}, \mathbf{6}..\mathbf{29}] \cup [\mathbf{15}..\mathbf{24}, \mathbf{15}..\mathbf{29}] \Rightarrow flag_p = false) \wedge (PC_p \in [\mathbf{20}..\mathbf{24}, \mathbf{20}..\mathbf{29}] \Rightarrow flag_p = true)$
   $\wedge\ (PC_p \in [\mathbf{26}, \mathbf{29}] \Rightarrow peer_p \in \mathcal{P})$
   $\wedge\ (PC_p \in [\mathbf{2}, \mathbf{16}] \Rightarrow status_p = good)$
7. $(PC_p = \mathbf{8} \Rightarrow (\textsc{Go}[p] = 0 \vee \text{abort was requested})) \wedge (PC_p = \mathbf{9} \Rightarrow \text{abort was requested})$
8. $PC_p \in \{\mathbf{25}, \mathbf{26}\} \Rightarrow (s_p \leq \textsc{Seq} \wedge (\forall q, PC_q \in \{\mathbf{13}, \mathbf{14}\} \Rightarrow s_p \leq s_q))$
9. $((PC_p = \mathbf{25} \wedge \text{CSStatus} = (0, s_p)) \Rightarrow$
   $\forall q, (\textsc{Registry}[q] \neq (q, \infty) \Rightarrow (PC_q \in \{\mathbf{6}\text{-}\mathbf{9}, \mathbf{18}, \mathbf{19}\} \vee (PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \textsc{Go}[q] \neq -1))))$
   $\wedge\ ((PC_p = \mathbf{26} \wedge \text{CSStatus} = (0, s_p)) \Rightarrow (PC_{peer_p} \in [\mathbf{6}, \mathbf{8}] \cup \{\mathbf{18}\text{-}\mathbf{20}, \mathbf{20}..\mathbf{24}\}$
   $\vee\ (PC_{peer_p} \in \{\mathbf{20}..\mathbf{25}, \mathbf{20}..\mathbf{26}\} \wedge s_{peer_p} = s_p)$
   $\vee\ (PC_{peer_p} \in \{\mathbf{1}, \mathbf{17}\} \wedge \textsc{Go}[peer_p] \neq -1)))$
10. $PC_p = \{\mathbf{28}, \mathbf{29}\} \Rightarrow 1 \leq g_p < \textsc{Token}$
11. $PC_p = \mathbf{29} \Rightarrow ((PC_{peer_p} \in \{\mathbf{3}, \mathbf{4}\} \Rightarrow 1 \leq g_p < tok_{peer_p})$
    $\wedge\ (PC_{peer_p} = \mathbf{5} \Rightarrow 1 \leq g_p < \textsc{Go}[peer_p])$
    $\wedge\ ((PC_{peer_p} \in \{\mathbf{6}, \mathbf{7}, \mathbf{8}\} \wedge g_p = \textsc{Go}[peer_p]) \Rightarrow \text{CSStatus} = (1, peer_p)))$
12. If a process is registered, some $q$ is either in CS or can be counted on to launch a waiting process into CS.

    $\min(\textsc{Registry}) \neq (*, \infty) \Rightarrow \exists q, (\text{CSStatus} = (1, q)$
    $\vee\ (PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \textsc{Go}[q] \neq -1) \vee PC_q \in \{\mathbf{6}, \mathbf{15}, \mathbf{18}\text{-}\mathbf{20}, \mathbf{24}\}$
    $\vee\ (PC_q \in \{\mathbf{25}, \mathbf{26}\} \wedge \text{CSStatus} = (0, s_q)))$
13. If $p$ has the ownership of CS but $\textsc{Go}[p] \neq 0$, then there is some $q$ that can be counted on to set $\textsc{Go}[p]$ to 0.

    $(\text{CSStatus} = (1, p) \wedge \textsc{Go}[p] \neq 0) \Rightarrow \exists q, (PC_q \in \{\mathbf{18}\text{-}\mathbf{20}, \mathbf{24}\} \vee (PC_q = \mathbf{27} \wedge peer_q = p)$
    $\vee\ (PC_q \in \{\mathbf{28}, \mathbf{29}\} \wedge peer_q = p \wedge g_q = \textsc{Go}[p])$
    $\vee\ (PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \textsc{Go}[q] \neq -1))$

Fig. 2: Invariant of the Abortable RME Algorithm from Figure 1.

the invariant that $\textsc{Go}[p] \neq -1$ and $\text{CSStatus} = (1, p)$ through this interval, all the way until $\texttt{recover}_p()$ completes. Therefore, when $p$ executes $\texttt{recover}_p()$, the condition on Line 17 does not apply, and when $p$ proceeds to Line 21 (after executing Lines 18, 19, and the procedure called at 20), it reads $(1, p)$ at Line 22, which causes it to reenter the CS. Hence, we conclude that $\text{CSStatus} = (1, p)$ from the time of $p$'s crash to the time of its reentering the CS. Therefore, if the lemma is false and some process $q$ enters the CS before this reentry of $p$ to the CS, at the configuration where $q$ enters the CS, $\text{CSStatus} = (1, q)$ (by Condition 5 of the invariant) and $\text{CSStatus} = (1, p)$ (as we just argued), which is a contradiction.

**Lemma 4 (No Trivial Aborts)** *In any run, if* $\textsc{AbortSignal}[p]$ *is false when a process* $p$ *invokes* $\texttt{try}_p()$ *and it remains false during the execution of* $\texttt{try}_p()$*, then* $\texttt{try}_p()$ *does not return IN_REM.*

*Proof* If $\textsc{AbortSignal}[p]$ is *false* during the entire execution of $\texttt{try}_p()$, it is clear from an inspection of the code that, when $p$ quits the loop at Line 7, $\textsc{Go}[p] = 0$. Since no process other than $p$ ever changes $\textsc{Go}[p]$ from 0, if $p$ doesn't crash and proceeds to execute Line 8, the

condition on Line 8 holds, so $\texttt{try}_p()$ returns IN_CS. Hence, we conclude that $\texttt{try}_p()$ does not return IN_REM.

**Lemma 5 (Responsiveness)** *The algorithm satisfies Bounded Exit, Bounded Abort, Bounded Recovery, and Fast Probing.*

*Proof* The *Bounded Exit*, *Bounded Abort*, and *Bounded Recovery* properties are immediate from the observation that any process $p$ can complete the methods $\texttt{promote}_p()$, $\texttt{exit}_p()$, $\texttt{abort}_p()$, and $\texttt{recover}_p()$ in $O(\log n)$ steps. *Fast Probing* is immediate from an inspection of Line 17 and the observation that Condition 3 of the invariant assures that $\text{GO}[p] = -1$ if $PC_p = 17$ and $status_p \in \{good, recover\text{-}from\text{-}rem\}$.

**Lemma 6 (Starvation-Freedom)** *The algorithm satisfies Strong Starvation-Freedom.*

*Proof* Assume to the contrary that the algorithm does not satisfy strong starvation-freedom. Then, there is a fair, infinite run $R$ and an attempt $A$ of a process $p$ such that there are finitely many crashes in $A$ and $p$ is stuck at Line 7 forever in $A$. Let $\tau$ be the token that $p$ writes in the registry in $A$. Then, there is a time $t$ in the attempt $A$ such that $p$ is at Line 7 at $t$ and no process crashes beyond $t$. For each process $q$, if $q$ has an active attempt at $t$ with a token $\tau'$ such that $(q, \tau') < (p, \tau)$, then move further in time such that either that attempt of $q$ completes or $q$ is stuck forever. It follows that there is a far enough time $t'$ such that (1) the set $S$ of active attempts stuck at Line 7 forever is nonempty (because $A \in S$), (2) for each active attempt $A' \in S$, if the process $q$ that executes $A'$ obtained a token $\tau'$ in $A'$ such that $(q, \tau') < (p, \tau)$, then $q$ is stuck at Line 7 forever, (3) If $PC_q = 26$ (for some $q$), then $(peer_q, tok_q) \leq (p, \tau)$ and $peer_q$'s attempt is in $S$, and (4) no process crashes beyond $t'$. Since the registry has $p$'s finite token, Condition 12 of the invariant asserts that there is a process $q$ such that one of the four disjuncts of that condition applies. Regardless of which disjunct applies, as $q$ takes steps, it is easy to see that $q$ calls $\texttt{promote}_q$ and performs Lines 24, 25, and 26, thereby launching some process $r$ from $S$ into CSStatus. Then, by Condition 13 of the invariant, there exists a process $q$ that writes 0 in $\text{GO}[r]$. Thus, $\text{GO}[r]$ gets set to 0, for some process $r$ that has an active attempt in $S$, contradicting that $r$ is stuck forever in the run.

**Lemma 7 (FCFS)** *The algorithm satisfies FCFS: there is an integer $b$ such that if (i) $R$ and $RR'$ are finite runs, (ii) $p$ starts an attempt $A$ in $R$ but does not complete it in $R$, (iii) $q$ starts an attempt $A'$ in $R$ and is in the CS in $A'$ at the end of $R$, (iv) before $q$ starts the attempt $A'$, $p$ performs at least $b$ normal steps in $A$ without any intervening crash steps of $p$ in $A$, and (v) $p$ has no crash steps in $R'$ and has at least $b$ steps in $R'$, then $p$ enters the CS or the remainder section in $A$ in $RR'$.*

*Proof* Let $b$ be the number of steps that it takes to execute all of Lines 2 through 6 and all of Lines 17 through 29, which makes $b = O(\lg n)$. Assume for a contradiction that the lemma is false, i.e., there are $R$, $R'$, $p$, $q$, $A$, and $A'$ that satisfy the conditions (i) through (v) in the lemma, yet $p$ does not enter the CS or the remainder section in $A$ in $RR'$. Let $C$ be the configuration in $R$ when $q$ starts its attempt $A'$, and $C'$ be the configuration at the end of $R$. We derive a contradiction through the following sequence of observations, where each observation is accompanied immediately by a justification.

O1. $p$ is at the busy-wait loop (Line 7) at $C'$.

Since $p$ does not enter the CS or the remainder in its attempt $A$ in $RR'$, given the definition of $b$, at $C'$ process $p$ must be at Lines 7, 8, 9, or 11 through 29 or at Line 1 with $status_p \neq good$. However, given the definition of $b$, if at $C'$ process $p$ were at any of Lines 11 through 29 or at

Line 1 with $status_p \neq good$, then $p$ would return IN_REM or IN_CS in $R'$, contradicting our assumption. Hence, the observation.

O2. $(p, tok_p) < (q, tok_q)$, where $(p, tok_p)$ and $(q, tok_q)$ are the values that $p$ and $q$ write in the registry in their attempts $A$ and $A'$, respectively, and $\text{REGISTRY}[p] = (p, tok_p)$ at all intermediate configurations of the run $R$, from $C$ to $C'$.

This observation follows from the fact that $p$ performs Lines 2 through 5 in $A$ before $q$ starts $A'$.

O3. $\text{CSSTATUS} \neq (1, q)$, $PC_q = 1$, and $\text{GO}[q] = -1$ at $C$.

Since $q$ starts an attempt at $C$, $q$ must in the remainder (i.e., $PC_q = 1$) with $status_q = good$. This fact, together with Condition 5 of the invariant, implies that $\text{CSSTATUS} \neq (1, q)$. The same fact, together with Condition 3 of the invariant, implies that $\text{GO}[q] = -1$.

O4. $\text{CSSTATUS} = (1, q)$ at $C'$.

Since $q$ is in the CS at $C'$ (i.e., $PC_q = 10$), Condition 5 of the invariant implies that $\text{CSSTATUS} = (1, q)$.

O5. Let $\sigma_{26}$ be the earliest step that occurs after $C$ (and before $C'$) where some process $r$ performs a successful CAS at Line 26, changing CSSTATUS from $(0, s_r)$ to $(1, q)$. Let $\sigma_{24}$ and $\sigma_{25}$ denote the steps where $r$ executes the prceding Lines 24 and 25. Then, it must be the case that $r$ reads $(0, s_r)$ in CSSTATUS at $\sigma_{24}$ and $r$ reads $(q, tok_r)$ in REGISTRY (i.e., $peer_r = q$) at $\sigma_{25}$.

O6. $\text{CSSTATUS} = (0, s_r)$ during the interval from $\sigma_{24}$ to $\sigma_{26}$.

We know that CSSTATUS has $(0, s_r)$ at $\sigma_{24}$ and at $\sigma_{26}$. It could not taken on a different value in the middle because, if it did, CSSTATUS would have had to change from $(0, s_r)$ to $(1, x)$, and then eventually back to $(0, y)$, but then $y$ would be strictly more than $s_r$, a contradiction.

O7. $PC_r \neq 26$ at $C$.

Assume to the contrary that $PC_r = 26$ at $C$. Then, it follows from Observation (4) that $\text{CSSTATUS} = (0, s_r)$ at $C$. Thus, at $C$, we have $PC_r = 26$, $\text{CSSTATUS} = (0, s_r)$, $peer_r = q$ (from Observation 3), and (from Observation 1) $PC_q = 1$ and $\text{GO}[q] = -1$, which contradicts Condition 9 of the invariant.

O8. The step $\sigma_{25}$ is performed after $C$.

This observation follows from Observations 3 and 5.

It follows from Observations O8 and O2 that $r$ could not have obtained $(q, *)$ at the step $\sigma_{25}$, contradicting Observation O5. Hence, we have the lemma.

The theorem below summarizes the result of our paper.

**Theorem 1** *The algorithm in Figure 1 is an abortable RME algorithm for $n$ processes using read, write, and CAS operations. It satisfies properties P1-P8 stated in Section 3. A process incurs $O(\min(k, \log n))$ RMRs per passage on DSM and Relaxed-CC machines and $O(n)$ RMRs per passage on Strict-CC machines, where $k$ is the maximum point contention during the passage. If a process $p$ crashes $f$ times during its attempt and $k$ is the maximum point contention during the attempt, on DSM and Relaxed-CC machines $p$ incurs $O(f + \min(k, \log n))$ RMRs in that attempt, and on Strict-CC machines $p$ incurs $O(f + n)$ RMRs in that attempt. The algorithm's space complexity is $O(n)$.*

*Proof* The lemmas of this section have established that the algorithm satisfies properties P1-P8. We now analyze the complexity. Consider the RMRs that a process $p$ incurs due to its busy-wait

at Line 7. On DSM machines, where Go[$p$] is assigned to $p$'s partition of NVRAM, $p$ does not incur any RMRs at Line 7. On Relaxed-CC machines, one RMR is incurred when bringing Go[$p$] to $p$'s cache at the start of executing Line 7, spinning on Go[$p$] incurs no RMRs, one RMR is incurred when some process changes Go[$p$] to 0, and possibly one more RMR is incurred to read that 0 in Go[$p$]. Thus, on Relaxed-CC machines, $p$ incurs only $O(1)$ RMRs at Line 7. On Strict-CC machines, while $p$ spins on Go[$p$] at Line 7, $O(n)$ processes could be at Line 29, each poised to perform a CAS on Go[$p$]. At most one of these succeeds in its CAS, but every one of them makes $p$ incur an RMR with their CAS (albeit the CAS is unsuccessful). Thus, $p$ can incur $O(n)$ RMRs at Line 7.

As explained in Section 4, each of Lines 5 and 19 incurs $O(\min(k, \log n))$ RMRs. Every other line in the code (except Line 7 that is already analyzed) incurs at most one RMR. Hence, we have the RMR complexity stated in the lemma.

The space complexity of $O(n)$ immediate from the observation that Go[$p$] array takes $O(n)$ space and Registry takes $O(n)$ space, as explained in Section 4, other shared variables take $O(1)$ space, and $O(1)$ local variables per process.

## 8 Conclusion

In this paper, we have introduced the notion of a mutual exclusion lock that is both recoverable and abortable. Our algorithm demonstrates a curious relation between recoverability and abortability: an algorithm designed only to be recoverable can easily incorporate abortability if only the recover method were carefully designed to be bounded even when recovering from a crash that occurs in the try method. This idea works because aborting can then be implemented by feigning a crash and executing the recover method. In fact, our algorithm showcased that this idea leads to an optimal RMR algorithm for DSM and Relaxed-CC machines when only the commonly available read, write, CAS operations may be used.

It would be interesting to explore if the logarithmic RMR complexity, shown here for DSM and Relaxed-CC machines, is also attainable for Strict-CC machines.

**Acknowledgment**: We thank Siddhartha Jayanti for his critical reading and comments and the NETYS '19 reviewers for their helpful feedback.

## References

1. Alon, A., and Morrison, A. Deterministic abortable mutual exclusion with sublogarithmic adaptive rmr complexity. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2018), PODC '18, ACM, pp. 27–36.
2. Attiya, H., Hendler, D., and Woelfel, P. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *Proc. of the Fortieth ACM Symposium on Theory of Computing* (New York, NY, USA, 2008), STOC '08, ACM, pp. 217–226.
3. Chan, D. Y. C., and Woelfel, P. Recoverable mutual exclusion with constant amortized rmr complexity from standard primitives. In *Proceedings of the 39th Symposium on Principles of Distributed Computing* (New York, NY, USA, 2020), PODC '20, Association for Computing Machinery, p. 181190.
4. Craig, T. S. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Tech. Rep. TR-93-02-02, Department of Computer Science, University of Washington, February 1993.
5. Dhoked, S., and Mittal, N. An adaptive approach to recoverable mutual exclusion. In *Proceedings of the 39th Symposium on Principles of Distributed Computing* (New York, NY, USA, 2020), PODC '20, Association for Computing Machinery, p. 110.
6. Dijkstra, E. W. Solution of a Problem in Concurrent Programming Control. *Commun. ACM 8*, 9 (Sept. 1965), 569–.

7. Giakkoupis, G., and Woelfel, P. Randomized abortable mutual exclusion with constant amortized rmr complexity on the cc model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2017), PODC '17, ACM, pp. 221–229.
8. Golab, W., and Hendler, D. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2017), PODC '17, ACM, pp. 211–220.
9. Golab, W., and Hendler, D. Recoverable Mutual Exclusion Under System-Wide Failures. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2018), PODC '18, ACM, pp. 17–26.
10. Golab, W., and Ramaraju, A. Recoverable Mutual Exclusion: [Extended Abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2016), PODC '16, ACM, pp. 65–74.
11. Intel. Intel® Optane™ DC Persistent Memory Product Brief. `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf`, 2019 (accessed November 26, 2020).
12. Jayanti, P. $f$-arrays: Implementation and Applications. In *Proceedings of the Twenty-first Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), PODC '02, ACM, pp. 270–279.
13. Jayanti, P. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2003), PODC '03, ACM, pp. 295–304.
14. Jayanti, P., Jayanti, S., and Joshi, A. Optimal Recoverable Mutual Exclusion using only FASAS. In *The 6th Edition of The International Conference on Networked Systems* (2018), NETYS 2018.
15. Jayanti, P., Jayanti, S., and Joshi, A. A recoverable mutex algorithm with sub-logarithmic rmr on both cc and dsm. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2019), PODC 19, Association for Computing Machinery, p. 177186.
16. Jayanti, P., and Jayanti, S. V. Constant Amortized RMR Complexity Deterministic Abortable Mutual Exclusion Algorithm for CC and DSM Models. In *Accepted for publication in PODC' 19* (2019).
17. Jayanti, P., and Joshi, A. Recoverable FCFS mutual exclusion with wait-free recovery. In *31st International Symposium on Distributed Computing* (2017), DISC 2017, pp. 30:1–30:15.
18. Jayanti, P., and Joshi, A. Recoverable mutual exclusion with abortability. In *Networked Systems* (Cham, 2019), M. F. Atig and A. A. Schwarzmann, Eds., Springer International Publishing, pp. 217–232.
19. Katzan, D., and Morrison, A. Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity. In *Proceedings of The International Conference on Principles of Distributed Systems (OPODIS 2020)* (2020), OPODIS 2020.
20. Lamport, L. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM 17*, 8 (Aug. 1974), 453–455.
21. Lee, H. Fast local-spin abortable mutual exclusion with bounded space. In *Proceedings of the 14th International Conference on Principles of Distributed Systems* (Berlin, Heidelberg, 2010), OPODIS'10, Springer-Verlag, pp. 364–379.
22. Mellor-Crummey, J. M., and Scott, M. L. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (Feb. 1991), 21–65.
23. Pareek, A., and Woelfel, P. Rmr-efficient randomized abortable mutual exclusion. In *Distributed Computing* (Berlin, Heidelberg, 2012), M. K. Aguilera, Ed., Springer Berlin Heidelberg, pp. 267–281.
24. Ramaraju, A. RGLock: Recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo, 2015.
25. Raoux, S., Burr, G. W., Breitwisch, M. J., Rettner, C. T., Chen, Y.-C., Shelby, R. M., Salinga, M., Krebs, D., Chen, S.-H., Lung, H.-L., et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4/5 (2008), 465.
26. Scott, M. L. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), PODC '02, ACM, pp. 31–40.
27. Scott, M. L., and Scherer, W. N. Scalable queue-based spin locks with timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming* (New York, NY, USA, 2001), PPoPP '01, ACM, pp. 44–52.
28. Scott, M. L., and Scherer, W. N. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming* (New York, NY, USA, 2001), PPoPP '01, ACM, pp. 44–52.
29. Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. The missing memristor found. *nature 453*, 7191 (2008), 80.
30. Tehrani, S., Slaughter, J. M., Deherrera, M., Engel, B. N., Rizzo, N. D., Salter, J., Durlam, M., Dave, R. W., Janesky, J., Butcher, B., et al. Magnetoresistive random access memory using magnetic tunnel junctions. *Proceedings of the IEEE 91*, 5 (2003), 703–714.

## A Invariant proof

**Lemma 8** *The algorithm in Figure 1 satisfies the invariant (i.e., the conjunction of all the conditions) stated in Figure 2, i.e., the invariant holds in every configuration of every run of the algorithm.*

*Proof* By induction.

For the base case, we verify that the conditions hold in the initial configuration $C_0$ of any run. Condition 1 holds in $C_0$ because TOKEN is initialized to 1. Condition 2 holds in $C_0$ because SEQ is initialized to 1 and CSSTATUS is initialized to $(0, 1)$. Condition 3 holds in $C_0$ because $PC_p$ is initialized to **1**, GO$[p]$ to $-1$, and $status_p$ to *good*. Condition 4 holds in $C_0$ because REGISTRY$[p]$ is initialized to $(p, \infty)$ and $PC_p$ is initially **1**. Condition 5 holds in $C_0$ because initially $PC_p = \mathbf{1}$, GO$[p] = -1$, and CSSTATUS $= (0, 1)$. Conditions 6 through 11 hold trivially because $PC_p = \mathbf{1}$ initially. Conditions 12 and 13 also hold because the REGISTRY is empty and CSSTATUS $= (0, 1)$ initially.

For the induction step, let $C$ be an arbitrary configuration of an arbitrary run. The induction hypothesis states that Conditions 1 through 13 hold in $C$. Let $(C, \sigma, C')$ be an arbitrary (normal or crash) step of a process in $\mathcal{P}$. We now establish the induction step by arguing that each of Conditions 1 through 13 holds in $C'$.

We use the following notation in the proof: (i) IH denotes the induction hypothesis and, for all $i \in [1, 13]$, IH:$i$ is the part of IH that states that Condition $i$ of the invariant holds in $C$, and (ii) If $D$ is any configuration and $x$ is any variable, $D.x$ is the value of $x$ in $D$.

1. Proof that Condition 1 holds in $C'$
   TOKEN changes only when a process $p$ executes Line **3**. If the CAS operation at that line succeeds, TOKEN increases by 1; otherwise, it remains unchanged. Therefore, $C'$.TOKEN $\geq C$.TOKEN. Since $C$.TOKEN $\geq 1$ by IH:1, we have $C'$.TOKEN $\geq 1$. Thus, Condition 1 holds in $C'$.
2. Proof that Condition 2 holds in $C'$
   (a) Lines **14** and **26** are the only lines of code that affect CSSTATUS.
   (b) If $PC_p = \mathbf{14}$, IH:6 implies that $C.s_p = C$.SEQ $- 1$. So, if a process $p$ writes $(0, s_p + 1)$ in CSSTATUS at Line **14**, Condition 2 holds in $C'$.
   (c) If $PC_p = \mathbf{26}$ and $p$ executes a successful CAS to change CSSTATUS to $(1, peer_p)$, IH:6 implies that $C.peer_p \in \mathcal{P}$. So, Condition 2 holds in $C'$.
3. Proof that Condition 3 holds in $C'$
   We establish each of the conjuncts of Condition 3.
   (a) $-1 \leq C'$.GO$[p] <$ TOKEN
   We prove this statement through the following observations.
      i. Lines **4**, **16**, **22**, and **29** are the only lines of code that affect GO$[p]$.
      ii. If $PC_p = \mathbf{4}$, IH:6 implies that $1 \leq C.tok_p <$ TOKEN. So, if $p$ executes Line **4**, $-1 \leq$ GO$[p] <$ TOKEN holds in $C'$.
      iii. If $p$ executes Line **16** or **22**, GO$[p]$ becomes $-1$, so $-1 \leq$ GO$[p] <$ TOKEN holds in $C'$.
      iv. If some $q \in \mathcal{P}$ performs a successful CAS at Line **29** and $peer_q = p$, GO$[p]$ becomes 0. Furthermore, IH:1 implies that $0 <$ TOKEN. So, $-1 \leq$ GO$[p] <$ TOKEN holds in $C'$.

   (b) $C'.PC_p = \mathbf{5} \Rightarrow C'$.GO$[p] = C'.tok_p$
   To prove this implication, assume that $C'.PC_p = \mathbf{5}$.

   In case $C.PC_p = \mathbf{5}$, by IH:3 $C$.GO$[p] = tok_p$ and the step from $C$ to $C'$ is by some $q \neq p$. If $q$ executes Line **29** with $peer_q = p$, GO$[p]$ remains unchanged because by IH:11, $g_q <$ GO$[p]$ and hence the CAS by $p$ will fail. if $q$ executes any other line or executes a crash step, GO$[p]$ remains unchanged. Therefore, $C'$.GO$[p] = C$.GO$[p]$.

   In case $C.PC_p = \mathbf{4}$, $p$ sets GO$[p]$ to $tok_p$ at Line **4**, we have $C'$.GO$[p] = tok_p$.

   (c) $C'.PC_p \in [\mathbf{6}, \mathbf{8}] \Rightarrow C'$.GO$[p] \in \{0, C'.tok_p\}$
   To prove this implication, assume that $C'.PC_p \in [\mathbf{6}, \mathbf{8}]$. We consider two cases: (i) $C.PC_p \notin [\mathbf{6}, \mathbf{8}]$, or (ii) $C.PC_p \in [\mathbf{6}, \mathbf{8}]$.
   In Case (i), the step from $C$ to $C'$ must be $p$'s execution of Line **5**. By IH:3 $C$.GO$[p] = tok_p$ and the step doesn't change the value of GO$[p]$, we have $C'$.GO$[p] = tok_p$.
   In Case (ii), we make two observations:
      i. $C$.GO$[p] \in \{0, tok_p\}$ (by IH:3).
      ii. $C'$.GO$[p] \in \{0, C$.GO$[p]\}$, as argued below.

      If the step from $C$ to $C'$ is by $p$, it must be an execution of one of Lines **6**, **6**::**24** -**6**::**28**, or **7**. Since $p$ does not change GO$[p]$ at these lines, $C'$.GO$[p] = C$.GO$[p]$. GO$[p]$ has the same value in $C'$ as it does in $C$.

If the step from $C$ to $C'$ is by some $q$ and $q$ executes Line **29**, Go[$p$] either remains unchanged or changes to 0; if $q$ executes any other line or executes a crash step, Go[$p$] remains unchanged. Therefore, $C'.\text{Go}[p] \in \{0, C.\text{Go}[p]\}$.

It is immediate from the above two observations that $C'.\text{Go}[p] \in \{0, tok_p\}$.

(d) $C'.PC_p \in \{\textbf{9-16}, \textbf{18-22}, \textbf{24-29}\} \Rightarrow C'.\text{Go}[p] \neq -1$:
To prove this implication, assume that in $C'.PC_p \in \{\textbf{9-16}, \textbf{18-22}, \textbf{24-29}\}$. We consider two cases: (i) $C.PC_p \notin \{\textbf{9-16}, \textbf{18-22}, \textbf{24-29}\}$, or (ii) $C.PC_p \in \{\textbf{9-16}, \textbf{18-22}, \textbf{24-29}\}$.
In Case (i), the step from $C$ to $C'$ must be $p$'s execution of Line **8** or of Line **17** such that the step causes $PC_p$ to become **18**, or $p$'s calling of $\texttt{promote}_p()$ at one of Lines **6**, **15**, or **20**, which causes $PC_p$ to become **24**. We now consider each of these possibilities.

i. If $p$ executes Line **8**, it does not change Go[$p$], so $C'.\text{Go}[p] = C.\text{Go}[p]$. Furthermore, IH:3 implies $C.\text{Go}[p] \in \{0, tok_p\}$ and IH:6 implies that $C.tok_p \neq -1$. Hence, $C'.\text{Go}[p] \neq -1$.
ii. If $p$ executes Line **17** such that $C'.PC_p = \textbf{18}$, the instruction at Line **17** implies that $C'.\text{Go}[p] \neq -1$.
iii. If $p$ calls $\texttt{promote}_p()$ at Line **6**, IH:3 implies that $C.\text{Go}[p] \in \{0, tok_p\}$ and IH:6 implies that $C.tok_p \neq -1$. Hence, $C'.\text{Go}[p] \neq -1$.
iv. If $p$ calls $\texttt{promote}_p()$ at Line **15** or Line **20**, IH:3 implies that $C.\text{Go}[p] \neq -1$. Hence, $C'.\text{Go}[p] \neq -1$.

Turning now to Case (ii), we make the following observations:

i. $C.\text{Go}[p] \neq -1$ (by IH:3).
ii. $C'.\text{Go}[p] \in \{0, C.\text{Go}[p]\}$, as argued below.

If the step from $C$ to $C'$ is by $p$, since $PC_p \in \{\textbf{9-16}, \textbf{18-22}, \textbf{24-29}\}$ in both $C$ and $C'$, the step must be an execution of one of Lines **9-15**, **18-21**, or **24-29**. These lines, with the exception of Line **29**, do not change Go[$p$]; and Line **29**, if it changes Go[$p$], sets it to 0. Therefore, $C'.\text{Go}[p] \in \{0, C.\text{Go}[p]\}$.

If the step from $C$ to $C'$ is by some $q \neq p$, if $q$ executes Line **29**, Go[$p$] either remains unchanged or changes to 0; if $q$ executes any other line or executes a crash step, Go[$p$] remains unchanged. Therefore, $C'.\text{Go}[p] \in \{0, C.\text{Go}[p]\}$.

It is immediate from the above two observations that $C'.\text{Go}[p] \neq -1$.

(e) $(C'.PC_p \in \{\textbf{2-4}, \textbf{23}\} \vee (C'.PC_p \in \{\textbf{1}, \textbf{17}\} \wedge C'.status_p \in \{good, recover\text{-}from\text{-}rem\})) \Rightarrow C'.\text{Go}[p] = -1$
We prove this implication in two parts.

i. $C'.PC_p \in \{\textbf{2-4}, \textbf{23}\} \Rightarrow C'.\text{Go}[p] = -1$
Assume that $C'.PC_p \in \{\textbf{2-4}, \textbf{23}\}$. Then, there are the following three subcases to analyze.

Subcase A: Either $C.PC_p = \textbf{1} \wedge C.status_p = good$ or $C.PC_p = \textbf{22}$, and $p$ executes a normal step.

In this subcase, if $C.PC_p = \textbf{1} \wedge C.status_p = good$, IH:3 implies that $C.\text{Go}[p] = -1$, so $C'.\text{Go}[p] = -1$. In (A), if $C.PC_p = \textbf{22}$, $p$ sets Go[$p$] to $-1$ in the step, so $C'.\text{Go}[p] = -1$.

Subcase B: $C.PC_p \in \{\textbf{2-4}, \textbf{23}\}$ and $p$ executes Line **2** or **3**.

In this subcase, IH:3 implies that $C.\text{Go}[p] = -1$ and $p$'s step does not affect Go[$p$], so $C'.\text{Go}[p] = -1$.

Subcase C: $C.PC_p \in \{\textbf{2-4}, \textbf{23}\}$ and some $q \neq p$ executes a (normal or crash) step.

In this subcase, IH:3 implies that $C.\text{Go}[p] = -1$. The only line of code by which $q$ can change Go[$p$] is via a successful CAS at Line **29**. However, IH:10 implies that $C.g_q \geq 0$, so $C.g_q \neq C.\text{Go}[p]$. Therefore, $q$'s execution of Line **29** cannot affect Go[$p$]. Hence, $C'.\text{Go}[p] = C.\text{Go}[p] = -1$.

We conclude that $PC_p \in \{\textbf{2-4}, \textbf{23}\} \Rightarrow \text{Go}[p] = -1$.

ii. $(C'.PC_p \in \{\textbf{1}, \textbf{17}\} \wedge C'.status_p \in \{good, recover\text{-}from\text{-}rem\}) \Rightarrow C'.\text{Go}[p] = -1$
Assume that $C'.PC_p \in \{\textbf{1}, \textbf{17}\} \wedge C'.status_p \in \{good, recover\text{-}from\text{-}rem\}$. Then, there are the following four subcases to analyze:

Subcase A: $C.PC_p \in \{\textbf{1}, \textbf{17}\} \wedge C.status_p \in \{good, recover\text{-}from\text{-}rem\}$ and some process $q \neq p$ executes a (normal or crash) step.

In this case, IH:3 implies that $C.\text{Go}[p] = -1$. The only line of code by which $q$ can change Go[$p$] is via a successful CAS at Line **29**. However, IH:10 implies that $C.g_q \geq 0$, so $C.g_q \neq C.\text{Go}[p]$. Therefore, $q$'s execution of Line **29** cannot affect Go[$p$]. Hence, $C'.\text{Go}[p] = C.\text{Go}[p] = -1$.

Subcase B: $C.PC_p$ is **16**, **23**, or **17** (and $\text{Go}[p] = -1$), and $p$ executes a normal step (causing $PC_p$ to become **1**).

If $C.PC_p = \mathbf{16}$ and $p$ executes a normal step, $p$ sets $\text{GO}[p]$ to $-1$, so $C'.\text{GO}[p] = -1$.

If $C.PC_p = \mathbf{23}$, IH:3 implies that $C.\text{GO}[p] = -1$; if $p$ executes a normal step, $\text{GO}[p]$ remains unchanged. Hence, $C'.\text{GO}[p] = -1$.

If $C.PC_p = \mathbf{17}$, $C.\text{GO}[p] = -1$, and $p$ executes a normal step, it is immediate that $C'.\text{GO}[p] = -1$.

Subcase C: $C.PC_p = \mathbf{1} \wedge C.status_p \in \{good, recover\text{-}from\text{-}rem\}$ and $p$ invokes $\texttt{recover}_p()$ (causing $PC_p$ to become $\mathbf{17}$).

IH:3 implies that $C.\text{GO}[p] = -1$. Since $p$'s invocation of $\texttt{recover}_p()$ does not affect $\text{GO}[p]$, it follows that $C'.\text{GO}[p] = -1$.

Subcase D: $C.PC_p = \mathbf{17} \wedge C.status_p \in \{good, recover\text{-}from\text{-}rem\}$ and $p$ executes a crash step (causing $PC_p$ to become $\mathbf{1}$ and $status_p$ to become $recover\text{-}from\text{-}rem$).

IH:3 implies that $C.\text{GO}[p] = -1$. Since $p$'s crash step does not affect $\text{GO}[p]$, it follows that $C'.\text{GO}[p] = -1$.

We conclude that $(C'.PC_p \in \{\mathbf{1}, \mathbf{17}\} \wedge C'.status_p \in \{good, recover\text{-}from\text{-}rem\}) \Rightarrow C'.\text{GO}[p] = -1$.

From (a), (b), (c), (d), and (e), we conclude that Condition 3 holds in $C'$.

4. Proof that Condition 4 holds in $C'$

In the following, we establish each of the conjuncts of Condition 4.

(a) $\exists t \in [1, C'.\text{TOKEN} - 1] \cup \{\infty\}, C'.\text{REGISTRY}[p] = (p, t)$

We observe that $\text{REGISTRY}[p]$ is changed only by $p$ and only at Lines **5**, **11**, and **19**. If $PC_p = \mathbf{5}$, $p$ writes $(p, tok_p)$ in $\text{REGISTRY}[p]$, and IH:6 implies that $1 \leq C.tok_p < C.\text{TOKEN}$; and if $PC_p$ is **11** or **19**, $p$ writes $(p, \infty)$ in $\text{REGISTRY}[p]$. Since $C'.\text{TOKEN} = C.\text{TOKEN}$, we conclude that $\exists t \in [1, C'.\text{TOKEN} - 1] \cup \{\infty\}, C'.\text{REGISTRY}[p] = (p, t)$.

(b) $C'.PC_p \in [\mathbf{6}, \mathbf{8}] \Rightarrow C'.\text{REGISTRY}[p] = (p, C'.tok_p))$

When $p$ executes Line **5**, causing $PC_p$ to become **6**, it writes $(p, tok_p)$ in $\text{REGISTRY}[p]$. Furthermore, neither $\text{REGISTRY}[p]$ nor $tok_p$ is changed by $p$ at Lines **6** and **7**, or by any other process at any line. Therefore, we have $C'.PC_p \in [\mathbf{6}, \mathbf{8}] \Rightarrow \text{REGISTRY}[p] = (p, C'.tok_p)$.

(c) $(C'.PC_p \in \{\mathbf{5}, \mathbf{12\text{-}16}, \mathbf{20\text{-}22}\} \vee C'.\text{GO}[p] = -1) \Rightarrow C'.\text{REGISTRY}[p] = (p, \infty)$

We prove this implication in two parts.

i. $C'.PC_p \in \{\mathbf{5}, \mathbf{12\text{-}16}, \mathbf{20\text{-}22}\} \Rightarrow C'.\text{REGISTRY}[p] = (p, \infty)$

To prove this implication, assume $C'.PC_p \in \{\mathbf{5}, \mathbf{12\text{-}16}, \mathbf{20\text{-}22}\}$. There are the following two subcases to analyze.

Subcase A: Assume that $C.PC_p \notin \{\mathbf{5}, \mathbf{12\text{-}16}, \mathbf{20\text{-}22}\}$
In this subcase, the step from $C$ to $C'$ must be $p$'s execution of any of Lines **4**, **11**, or **19**. If $p$ executes Line **4**, IH:3 implies that $C.\text{GO}[p] = -1$, which together with IH:4, implies that $C.\text{REGISTRY}[p] = (p, \infty)$. Since $p$ does not change $\text{REGISTRY}[p]$ at Line **4**, we have $C'.\text{REGISTRY}[p] = C.\text{REGISTRY}[p] = (p, \infty)$.

If $p$ executes Line **11** or **19**, since it writes $(p, \infty)$ in $\text{REGISTRY}[p]$ at these lines, we have $C'.\text{REGISTRY}[p] = (p, \infty)$.

Subcase B: Assume that $C.PC_p \in \{\mathbf{5}, \mathbf{12\text{-}16}, \mathbf{20\text{-}22}\}$
In this subcase, since $p$ does not change $\text{REGISTRY}[p]$ at Lines **12 -15**, and at Lines **20** and **21**, and any $q \neq p$ does not change $\text{REGISTRY}[p]$ (regardless of $q$'s step, normal or crash), we have $C'.\text{REGISTRY}[p] = C.\text{REGISTRY}[p]$. Furthermore, we have $C.\text{REGISTRY}[p] = (p, \infty)$ by IH:4. It follows that $C'.\text{REGISTRY}[p] = (p, \infty)$.

ii. $C'.\text{GO}[p] = -1 \Rightarrow C'.\text{REGISTRY}[p] = (p, \infty)$

To prove this implication, assume $C'.\text{GO}[p] = -1$. There are the following two subcases to analyze.

Subcase A: Assume that $C.\text{GO}[p] = -1$.
In this subcase, IH:4 implies that $C.\text{REGISTRY}[p] = (p, \infty)$. Furthermore, since $C.\text{GO}[p] = -1$, IH:3 and IH:6 implies that $C.PC_p \neq \mathbf{5}$, so the step from $C$ to $C'$ is not $p$'s execution of Line **5**. Since the execution of Line **5** by $p$ is the only way to write a non-$\infty$ value in $\text{REGISTRY}[p]$, it follows that $C'.\text{REGISTRY}[p] = (p, \infty)$.

Subcase B: Assume that $C.\text{GO}[p] \neq -1$.
In this subcase, since $C.\text{GO}[p] \neq -1$ and $C'.\text{GO}[p] = -1$, it follows that the step from $C$ to $C'$ is $p$'s execution of Line **16** or Line **22**. Thus, $C.PC_p \in \{\mathbf{16}, \mathbf{22}\}$. It follows from IH:4 that $C.\text{REGISTRY}[p] =$

$(p, \infty)$. Furthermore, since Lines **16** and **22** don't affect REGISTRY$[p]$, we have $C'$.REGISTRY$[p]$ = $C$.REGISTRY$[p] = (p, \infty)$.

From (a), (b), and (c), we conclude that Condition 4 holds in $C'$.

5. Proof that Condition 5 holds in $C'$
We establish Condition 5 in parts.

(a) Prove $(C'.PC_p \in [\mathbf{6}, \mathbf{8}] \wedge C'.\text{GO}[p] = 0) \Rightarrow C'.\text{CSSTATUS} = (1, p)$
To prove this implication, assume that $C'.PC_p \in [\mathbf{6}, \mathbf{8}] \wedge C'.\text{GO}[p] = 0$. We consider the following cases.

- Case 1: $C.PC_p \in [\mathbf{6}, \mathbf{8}] \wedge C.\text{GO}[p] = 0$
In this case, IH:5 implies that $C$.CSSTATUS $= (1, p)$. The only lines of code that can potentially change CSSTATUS are Lines **14** and **26**. However, since $C$.CSSTATUS $= (1, p)$, if any process executes Line **26** from $C$, the CAS instruction at that line fails and CSSTATUS remains unchanged. Since $C.PC_p \in [\mathbf{6}, \mathbf{8}]$, the step from $C$ to $C'$ cannot be $p$'s execution of Line **14**. Since $C$.CSSTATUS $= (1, p)$, we have CSSTATUS $\neq (1, q)$ for any $q \neq p$. Then, IH:5 implies that $PC_q \neq \mathbf{14}$. So, the step from $C$ to $C'$ cannot be $q$'s execution of Line **14**. We conclude that $C'$.CSSTATUS $= C$.CSSTATUS $= (1, p)$.
- Case 2: $C.PC_p \in [\mathbf{6}, \mathbf{8}] \wedge C.\text{GO}[p] \neq 0$
Since $C.\text{GO}[p] \neq 0$ and $C'.\text{GO}[p] = 0$, the step from $C$ to $C'$ must be the successful execution of the CAS instruction at Line **29** by some process $q$ with $peer_q = p$, as Line **29** is the only line of code where 0 is written into a GO variable. Then, IH:11 implies that $C$.CSSTATUS $= (1, p)$. Since $q$'s execution of Line **29** does not affect CSSTATUS, it follows that $C'$.CSSTATUS $= C$.CSSTATUS $= (1, p)$.
- Case 3: $C.PC_p \notin [\mathbf{6}, \mathbf{8}]$
Since $C.PC_p \notin [\mathbf{6}, \mathbf{8}]$ and $C'.PC_p \in [\mathbf{6}, \mathbf{8}]$, the step from $C$ to $C'$ must be $p$'s execution of Line **5**. Thus, $C.PC_p = \mathbf{5}$. Then, IH:3 implies that $C.\text{GO}[p] = tok_p$ and IH:6 implies that $tok_p \geq 1$. It follows that $C.\text{GO}[p] \neq 0$. Since $p$'s execution of Line **5** does not modify GO$[p]$, it follows that $C'.\text{GO}[p] = C.\text{GO}[p] \neq 0$, contradicting our earlier assumption that $C'.\text{GO}[p] = 0$. We conclude that Case 3 does not arise.

(b) Prove $C'.PC_p \in [\mathbf{10}, \mathbf{14}] \Rightarrow C'.\text{CSSTATUS} = (1, p))$
Assume that $C'.PC_p \in [\mathbf{10}, \mathbf{14}]$. There are two cases to analyze.

- Case 1: $C.PC_p \notin [\mathbf{10}, \mathbf{14}]$
Since $PC_p \in [\mathbf{10}, \mathbf{14}]$ is false in $C$ and true in $C'$, the step from $C$ to $C'$ must be $p$'s execution of Line **8** with GO$[p] = 0$, or of Line **21** with CSSTATUS $= (1, p)$. In the former case, IH:5 implies that $C$.CSSTATUS $= (1, p)$. Since Lines **8** and **21** do not modify CSSTATUS, it follows that $C'$.CSSTATUS = $C$.CSSTATUS $= (1, p)$.
- Case 2: $C.PC_p \in [\mathbf{10}, \mathbf{14}]$
In this case, IH:5 implies that $C$.CSSTATUS $= (1, p)$. The only lines of code that can potentially change CSSTATUS are Lines **14** and **26**. However, since $C$.CSSTATUS $= (1, p)$, if any process executes Line **26** from $C$, the CAS instruction at that line fails and CSSTATUS remains unchanged. Since $C'.PC_p \in [\mathbf{10}, \mathbf{14}]$, the step from $C$ to $C'$ cannot be $p$'s execution of Line **14**. Since $C$.CSSTATUS = $(1, p)$, we have CSSTATUS $\neq (1, q)$ for any $q \neq p$. Then, IH:5 implies that $PC_q \neq \mathbf{14}$. So, the step from $C$ to $C'$ cannot be $q$'s execution of Line **14**. We conclude that $C'$.CSSTATUS $= C$.CSSTATUS $= (1, p)$.

(c) Prove $(C'.status_p = \textit{recover-from-cs}) \Rightarrow C'.\text{CSSTATUS} = (1, p)$
Assume that $C'.status_p = \textit{recover-from-cs}$. There are two cases to analyze.

- Case 1: $C.status_p \neq \textit{recover-from-cs}$
Since $C.status_p \neq \textit{recover-from-cs}$ and $C'.status_p = \textit{recover-from-cs}$, it must be the case that $p$ is in CS (i.e., $C.PC_p = \mathbf{10}$) in $C$ and the step from $C$ to $C'$ must be $p$'s crash step. Then, IH:5 implies that $C$.CSSTATUS $= (1, p)$. Since CSSTATUS is a non-volatile variable, $C'$.CSSTATUS $= (1, p)$.
- Case 2: $C.status_p = \textit{recover-from-cs}$
In this case, IH:5 implies that $C$.CSSTATUS $= (1, p)$. The only lines of code that can potentially change CSSTATUS are Lines **14** and **26**. However, since $C$.CSSTATUS $= (1, p)$, if any process executes Line **26** from $C$, the CAS instruction at that line fails and CSSTATUS remains unchanged. Since $C.status_p = \textit{recover-from-cs}$, IH:6 implies that $C.PC_p \neq \mathbf{14}$. Hence, the step from $C$ to $C'$ cannot be $p$'s execution of Line **14**. Since $C$.CSSTATUS $= (1, p)$, we have CSSTATUS $\neq (1, q)$ for any $q \neq p$. Then, IH:5 implies that $PC_q \neq \mathbf{14}$. So, the step from $C$ to $C'$ cannot be $q$'s execution of Line **14**. We conclude that $C'$.CSSTATUS $= C$.CSSTATUS $= (1, p)$.

(d) Prove $C'.PC_p \in \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}] \Rightarrow C'.\text{CSSTATUS} \neq (1, p)$
Assume that $C'.PC_p \in \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}]$. There are two cases to analyze.

- Case 1: $C.PC_p \notin \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}]$
Since $PC_p \in \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}]$ is false in $C$ and true in $C'$, the step from $C$ to $C'$ must be $p$'s execution of Line **4**, Line **21** (with CSSTATUS $\neq (1, p)$), or Line **14**.

If $p$ executes Line **4**, IH:3 implies that $C.\text{GO}[p] = -1$, which together with IH:5 implies that $C.\text{CSSTATUS} \neq (1, p)$. Since $p$ does not modify CSSTATUS at Line **4**, it follows that $C'.\text{CSSTATUS} = C.\text{CSSTATUS} \neq (1, p)$.

If $p$ executes Line **21** with $\text{CSSTATUS} \neq (1, p)$, then it is trivially the case that $C'.\text{CSSTATUS} \neq (1, p)$.

If $p$ executes Line **14**, $p$ writes $(0, s_p + 1)$ in CSSTATUS. Hence, $C'.\text{CSSTATUS} \neq (1, p)$.

- Case 2: $C.PC_p \in \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}]$
  Since $C.PC_p \in \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}]$, IH:5 implies that $C.\text{CSSTATUS} \neq (1, p)$. The only line of code that can potentially change CSSTATUS's value to $(1, p)$ is Line **26**. For a process $q$ (possibly $q = p$) to execute a successful CAS at Line **26** and change CSSTATUS to $(1, p)$, it must be the case that $C.PC_q = \mathbf{26}$, $\text{CSSTATUS} = (0, s_q)$, and $peer_q = p$. Then, IH:9 implies that $C.PC_p \in [\mathbf{6}, \mathbf{8}] \cup \{\mathbf{18} - \mathbf{20}, \mathbf{20}::\mathbf{24}, \mathbf{20}::\mathbf{25}, \mathbf{20}::\mathbf{26}, \mathbf{1}, \mathbf{17}\}$. This set of values for $C.PC_p$ is incompatible with the case we are considering because $C.PC_p \in \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}]$ in the case we are considering. We conclude that the step from $C$ to $C'$ cannot change the value in CSSTATUS to $(1, p)$. Hence, $C'.\text{CSSTATUS} \neq (1, p)$.

(e) Prove $C'.\text{GO}[p] = -1 \Rightarrow C'.\text{CSSTATUS} \neq (1, p)$
Assume that $C'.\text{GO}[p] = -1$. There are two cases to analyze.

- Case 1: $C.\text{GO}[p] \neq -1$
  Since GO[$p$] is -1 in $C'$ but not in $C$, the step from $C$ to $C'$ must be $p$'s execution of Line **16** or Line **22**. Thus, $C.PC_p \in \{\mathbf{16}, \mathbf{22}\}$. Then, IH:5 implies that $C.\text{CSSTATUS} \neq (1, p)$. Since $p$ does not change CSSTATUS in Line **16** or **22**, it follows that $C'.\text{CSSTATUS} \neq (1, p) = C.\text{CSSTATUS} \neq (1, p)$.
- Case 2: $C.\text{GO}[p] = -1$
  Since $C.\text{GO}[p] = -1$, IH:5 implies that $C.\text{CSSTATUS} \neq (1, p)$. The only line of code that can potentially change CSSTATUS's value to $(1, p)$ is Line **26**. For a process $q$ (possibly $q = p$) to execute a successful CAS at Line **26** and change CSSTATUS to $(1, p)$, it must be the case that $C.PC_q = \mathbf{26}$, $\text{CSSTATUS} = (0, s_q)$, and $peer_q = p$. This, together with IH:9 and $C.\text{GO}_p = -1$, implies that $C.PC_p \in [\mathbf{6}, \mathbf{8}] \cup \{\mathbf{18}\text{-}\mathbf{20}, \mathbf{20}::\mathbf{24}, \mathbf{20}::\mathbf{25}, \mathbf{20}::\mathbf{26}\}$. This set of values for $C.PC_p$'s is incompatible with the case we are considering because IH:3, together with $C.\text{GO}[p] = -1$, implies that $C.PC_p \in \{\mathbf{1}\text{-}\mathbf{4}, \mathbf{17}, \mathbf{23}\}$. We conclude that the step from $C$ to $C'$ cannot change the value in CSSTATUS to $(1, p)$. Hence, $C'.\text{CSSTATUS} \neq (1, p)$.

6. Proof that Condition 6 holds in $C'$
This condition mostly concerns local variables—variables that are modified only when $p$ executes a line of code or a crash step. We establish the condition in parts.

(a) Proof of $C'.PC_p = \mathbf{3} \Rightarrow 1 \leq C'.tok_p \leq C'.\text{TOKEN}$
If $C.PC_p \neq \mathbf{3}$, the step from $C$ to $C'$ must be $p$'s execution of Line **2**. Inspection of Line **2**, together with IH:1, implies that $1 \leq C'.tok_p \leq C'.\text{TOKEN}$.

On the other hand, if $C.PC_p = \mathbf{3}$, IH:6 implies that $1 \leq C.tok_p \leq C.\text{TOKEN}$. Since $PC_p = \mathbf{3}$ in both $C$ and $C'$, the only step from $C$ to $C'$ that can affect the condition is the execution of Line **3** by some process $q \neq p$. However, $q$'s execution of Line **3** does not decrease TOKEN's value. It follows that, since $1 \leq tok_p \leq \text{TOKEN}$ holds in $C$, it continues to hold in $C'$.

(b) Proof of $C'.PC_p \in [\mathbf{4}, \mathbf{8}] \Rightarrow 1 \leq C'.tok_p < C'.\text{TOKEN}$
Assume that $C'.PC_p \in [\mathbf{4}, \mathbf{8}]$. We consider two cases.

- Case 1: $PC_p \in [\mathbf{4}, \mathbf{8}]$ is false in $C$
  Since $PC_p \in [\mathbf{4}, \mathbf{8}]$ is false in $C$ and true in $C'$, the step from $C$ to $C'$ must be $p$'s execution of Line **3**. Thus, $C.PC_p = \mathbf{3}$ and IH:6 implies that either $1 \leq C.tok_p = C.\text{TOKEN}$ or $1 \leq C.tok_p < C.\text{TOKEN}$. In the former case, $p$'s execution of Line **3** increments TOKEN by 1. Thus, in either case, we have $1 \leq C'.tok_p < C'.\text{TOKEN}$.
- Case 2: $PC_p \in [\mathbf{4}, \mathbf{8}]$ is true in $C$
  Since $PC_p \in [\mathbf{4}, \mathbf{8}]$ is true in $C$, IH:6 implies that $1 \leq tok_p < \text{TOKEN}$ holds in $C$. Since $PC_p \in [\mathbf{4}, \mathbf{8}]$ in $C$ and $C'$, if $p$ takes a step from $C$, it executes one of Lines **4**, **5**, **6**, or **7**, which do not affect $tok_p$ or TOKEN. If a process $q \neq p$ takes a step from $C$, it can possibly change TOKEN by executing Line **3**, but the CAS instruction at Line **3** ensures that TOKEN's value does not decrease. Therefore, since $1 \leq tok_p < \text{TOKEN}$ holds in $C$, it continues to hold in $C'$.

(c) Proof of $C'.PC_p = \mathbf{13} \Rightarrow C'.s_p = C'.\text{SEQ}$
Assume that $C'.PC_p = \mathbf{13}$.

Suppose that $PC_p \neq \mathbf{13}$ in $C$. Then, the step from $C$ must be $p$'s execution of Line **12**. Hence, $s_p = \text{SEQ}$ in $C'$.

Suppose that $PC_p = \mathbf{13}$ in $C$. Then, IH:6 implies that $C.s_p = C.\text{SEQ}$ and IH:5 implies that $C.\text{CSSTATUS} = (1, p)$. Since $PC_p = \mathbf{13}$ in both $C$ and $C'$, the process that takes the step from $C$ is some $q \neq p$. Since

Line **13** is the only line that changes SEQ, if $q$ changes SEQ in its step from $C$, $C.PC_q$ must be **13**. Then, IH:5 implies that $C$.CSSTATUS $= (1, q)$, contradicting that $C$.CSSTATUS $= (1, p)$. We conclude that $q$'s step does not change SEQ. So, $s_p =$ SEQ, which holds in $C$, remains true in $C'$.

(d) Proof of $PC_p = \mathbf{14} \Rightarrow s_p =$ SEQ $- 1$

Assume that $C'.PC_p = \mathbf{14}$.

Suppose that $PC_p \neq \mathbf{14}$ in $C$. Then, the step from $C$ must be $p$'s execution of Line **13**. Hence, IH:6 implies that $C$.SEQ $= C.s_p$ and the increment of $s_p$ at Line **13** ensures $s_p =$ SEQ $- 1$ in $C'$.

Suppose that $PC_p = \mathbf{14}$ in $C$. Then, IH:6 implies that $C.s_p = C$.SEQ $- 1$ and IH:5 implies that $C$.CSSTATUS $= (1, p)$. Since $PC_p = \mathbf{14}$ in both $C$ and $C'$, the process that takes the step from $C$ is some $q \neq p$. Since Line **13** is the only line that changes SEQ, if $q$ changes SEQ in its step from $C$, $C.PC_q$ must be **13**. Then, IH:5 implies that $C$.CSSTATUS $= (1, q)$, contradicting that $C$.CSSTATUS $= (1, p)$. We conclude that $q$'s step does not change SEQ. So, $s_p =$ SEQ $- 1$, which holds in $C$, remains true in $C'$.

(e) Proof of $PC_p \in [\mathbf{6}..\mathbf{24}, \mathbf{6}..\mathbf{29}] \cup [\mathbf{15}..\mathbf{24}, \mathbf{15}..\mathbf{29}] \Rightarrow flag_p = false$

This statement follows from the observation that `promote`() is called at Line **6** and at Line **15** with *false* as the parameter.

(f) Proof of $PC_p \in [\mathbf{20}..\mathbf{24}, \mathbf{20}..\mathbf{29}] \Rightarrow flag_p = true$

This statement follows from the observation that `promote`() is called at Line **20** with *true* as the parameter.

(g) Proof of $PC_p \in [\mathbf{26}, \mathbf{29}] \Rightarrow peer_p \in \mathcal{P}$

This statement follows from the observation that, as $p$ enters Line **26** from Line **25**, it sets $peer_p$ to either $p$ or the first component of the value it found in REGISTRY$[p]$. In either case, $peer_p \in \mathcal{P}$.

(h) Proof of $C'.PC_p \in [\mathbf{2}, \mathbf{16}] \Rightarrow C'.status_p = good$

Assume that $C'.PC_p \in [\mathbf{2}, \mathbf{16}]$. We consider two cases.

Suppose that $C.PC_p \notin [\mathbf{2}, \mathbf{16}]$. Then, the step from $C$ must be either $p$'s invocation of $\texttt{try}_p()$ (from the remainder section of Line **1**) when $status_p = good$, or entry into the CS (Line **10**) because $\texttt{recover}_p()$ returns IN_CS at Line **18**, simultaneously setting $status_p$ to *good*. In either case, we have $C'.status_p = good$.

Suppose that $C.PC_p \in [\mathbf{2}, \mathbf{16}]$. By IH:6, $C.status_p = good$. If $p$ takes a crash step from $C$, $PC_p$ becomes **17**, contradicting that $C'.PC_p \in [\mathbf{2}, \mathbf{16}]$. If $p$ takes a normal step from $C$ or a different process takes a (normal or crash) step from $C$, $C'.status_p = good$ because $C.status_p = good$.

7. Proof that Condition 7 holds in $C'$

We prove this condition in two parts.

(a) Proof of $C'.PC_p = \mathbf{8} \Rightarrow (C'.\text{GO}[p] = 0 \vee$ abort was requested)

Assume that $C'.PC_p = \mathbf{8}$. We consider two cases.

Suppose that $C.PC_p \neq \mathbf{8}$. Then, the step from $C$ must be $p$'s read of 0 in GO$[p]$ or of *true* in ABORTSIGNAL$[p]$ at Line **7**. Hence, we have $C'.\text{GO}[p] = 0 \vee$ abort was requested.

Suppose that $C.PC_p = \mathbf{8}$. Then, IH:7 implies that $C.\text{GO}[p] = 0 \vee$ abort was requested. Since $PC_p = \mathbf{8}$ in both $C$ and $C'$, the process that takes the step from $C$ is some $q \neq p$. There is no line of code where $q$ can possibly change GO$[p]$ to any non-zero value. Therefore, the condition GO$[p] = 0 \vee$ abort was requested, which holds in $C$, continues to hold in $C'$.

(b) Proof of $C'.PC_p = \mathbf{9} \Rightarrow$ abort was requested

When $PC_p = \mathbf{8}$, IH:7 implies that GO$[p] = 0 \vee$ abort was requested. So, when $p$ executes the **if** statement at Line **8** and enters Line **9**, abort was requested is true.

8. Proof that Condition 8 holds in $C'$

We prove this condition in parts.

(a) Proof of $PC_p \in \{\mathbf{25}, \mathbf{26}\} \Rightarrow s_p \leq$ SEQ

Suppose that $p$ executes the step from $C$, and in the step, $p$ executes Line **24** and enters Line **25**. It follows that at Line **24**, $p$ reads $(0, s_p)$ and, by IH:2, $s_p = C$.SEQ. Since IH:6 ensures that Line **13** does not decrease the value of SEQ, it follows that $PC_p \in \{\mathbf{25}, \mathbf{26}\} \Rightarrow s_p \leq$ SEQ.

(b) $\forall q \in \mathcal{P}, (PC_p \in \{\mathbf{25}, \mathbf{26}\} \wedge PC_q \in \{\mathbf{13}, \mathbf{14}\}) \Rightarrow s_p \leq s_q$

Assume that $C'.PC_p \in \{\mathbf{25}, \mathbf{26}\} \wedge C'.PC_q \in \{\mathbf{13}, \mathbf{14}\}$, for an arbitrary $q \in \mathcal{P}$. We consider two cases.

- Case 1: The condition $PC_p \in \{\mathbf{25}, \mathbf{26}\} \wedge PC_q \in \{\mathbf{13}, \mathbf{14}\}$ is false in $C$.
  Since the condition is true in $C'$, it must be the case that either $PC_q \in \{\mathbf{13}, \mathbf{14}\}$ is true in $C$ and $p$ executes Line **24** from $C$ (to enter Line **25**), or $PC_p \in \{\mathbf{25}, \mathbf{26}\}$ is true in $C$ and $q$ executes Line **12** from $C$ (to enter Line **13**). In the former case, IH:5 implies that $C$.CSSTATUS $= (1, q)$, so in its step from $C$, $p$ reads $(1, q)$ at Line **24** and goes to Line **27**, contradicting that $p$ moves to Line **25**. In the latter case, $q$'s step from $C$ sets $s_q$ to SEQ, and IH:8 implies that $C.s_p \leq C$.SEQ. Hence, we have $s_p \leq s_q$ in $C'$.

– Case 2: The condition $PC_p \in \{\mathbf{25}, \mathbf{26}\} \wedge PC_q \in \{\mathbf{13}, \mathbf{14}\}$ is true in $C$.
Since the condition is true in $C$, IH:8 implies that $s_p \leq s_q$ is true in $C$. Furthermore, since the condition is true in both $C$ and $C'$, it follows that the step from $C$ is $p$'s execution of Line **25**, or $q$'s execution of Line **13**, or an arbitrary step of a process different from $p$ and $q$. In all these cases, $s_p$ and $s_q$ are unaffected, so $s_p \leq s_q$ remains true in $C'$.

9. Proof that Condition 9 holds in $C'$
We prove this condition in parts.
(a) Assume $C'.PC_p = \mathbf{25} \wedge C'.\text{CSSTATUS} = (0, s_p)$.
Let $q$ be any process in $\mathcal{P}$ (including possibly $p$), and assume that $C'.\text{REGISTRY}[q] \neq (q, \infty)$. Since $C'.PC_p = \mathbf{25}$, $C.PC_p$ must be either **24** or **25**. We analyze these cases below and show that in each case, $C'.PC_q \in \{\mathbf{6-9}, \mathbf{18}, \mathbf{19}\}$ or $C'.PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge C'.\text{GO}[q] \neq -1$.
– Case 1: $C.PC_p = \mathbf{24}$
Since $PC_p$ is **24** in $C$ and **25** in $C'$, the step from $C$ to $C'$ must be $p$'s execution of Line **24**, where it finds $(0, s_p)$ in CSSTATUS. Since Line **24** does not change REGISTRY, we have $C.\text{REGISTRY}[q] = C'.\text{REGISTRY}[q] \neq (q, \infty)$. It follows from IH:4 that $C.PC_q \notin \{\mathbf{5}, \mathbf{12-16}, \mathbf{20-22}\}$ and $C.\text{GO}[q] \neq -1$. Furthermore, since $C.\text{CSSTATUS} = (0, s_p) \neq (1, q)$, it follows from IH:5 that $C.PC_q \notin [\mathbf{10}, \mathbf{14}]$. Since $\text{GO}[q] \neq -1$, it follows from IH:3 that $C.PC_q \notin \{\mathbf{2-4}, \mathbf{23}\}$. Together, the above conditions imply $C.PC_q \in \{\mathbf{6-9}, \mathbf{18}, \mathbf{19}\}$ or $C.PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \text{GO}[q] \neq -1$. Regardless of whether $q = p$ or not, the same condition holds in $C'$, i.e., $C'.PC_q \in \{\mathbf{6-9}, \mathbf{18}, \mathbf{19}\} \vee (C'.PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \text{GO}[q] \neq -1)$.
– Case 2: $C.PC_p = \mathbf{25} \wedge C.\text{CSSTATUS} \neq (0, s_p)$
In this case, since $C.\text{CSSTATUS} \neq (0, s_p)$ and $C'.\text{CSSTATUS} = (0, s_p)$, the step from $C$ must be the execution of Line **14** by some process $r$. However, IH:8 implies that $C.s_p \leq C.s_r$. Since $r$ writes $(0, s_r + 1)$ in CSSTATUS, it follows that $C'.\text{CSSTATUS} \neq (0, s_p)$, contradicting our initial assumption. We conclude that Case 2 cannot arise.
– Case 3: $C.PC_p = \mathbf{25} \wedge C.\text{CSSTATUS} = (0, s_p)$
Suppose that $C.\text{REGISTRY}[q] = (q, \infty)$. Then, since $C'.\text{REGISTRY}[q] \neq (q, \infty)$, the step from $C$ must be $q$'s execution of Line **5**. Hence, $C'.PC_q = \mathbf{6}$.

Next, suppose that $C.\text{REGISTRY}[q] \neq (q, \infty)$. Then, IH:9 implies that $C.PC_q \in \{\mathbf{6-9}, \mathbf{18}, \mathbf{19}\} \vee (C.PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge C.\text{GO}[q] \neq -1)$. The only way that this condition becomes *false* in $C'$ is when $C.PC_q = \mathbf{19}$ and $q$ executes a step. However, $q$'s step causes $C'.\text{REGISTRY}[q]$ to be $(q, \infty)$, contradicting our assumption.

(b) Assume $C'.PC_p = \mathbf{26} \wedge C'.\text{CSSTATUS} = (0, s_p)$
Since $C'.PC_p = \mathbf{26}$, $C.PC_p$ must be either **25** or **26**. We analyze by cases below.
– Case 1: $C.PC_p = \mathbf{25}$
Since $PC_p$ is **25** in $C$ and **26** in $C'$, the step from $C$ to $C'$ must be $p$'s execution of Line **25**, where `findmin`() returns $(peer_p, tok_p)$ and either $tok_p \neq \infty$ or $C.flag_p = true$ and $C'.peer_p = p$. We consider each of these possibilities below.
Suppose that `findmin`() returns $(peer_p, tok_p)$ and $tok_p \neq \infty$. This implies that $C.\text{REGISTRY}[peer_p] \neq (peer_p, \infty)$. It follows from IH:9 that $C.PC_{peer_p} \in \{\mathbf{6-9}, \mathbf{18}, \mathbf{19}\} \vee (C.PC_{peer_p} \in \{\mathbf{1}, \mathbf{17}\} \wedge C.\text{GO}[peer_p] \neq -1)$. If $peer_p \neq p$, the same condition holds for $C'$. If $peer_p = p$, then the condition implies that $p$'s step from $C$ is the execution of Line **25**, as part of $\texttt{promote}_p()$ called from Line **6**, and the condition remains true for $C'$. Thus, we have $C'.PC_{peer_p} \in \{\mathbf{6-9}, \mathbf{18}, \mathbf{19}\} \vee (C'.PC_{peer_p} \in \{\mathbf{1}, \mathbf{17}\} \wedge C'.\text{GO}[peer_p] \neq -1)$.

Next, suppose that $C.flag_p = true$ and $C'.peer_p = p$. Then, $C'.PC_{peer_p} = C'.PC_p = \mathbf{20}::\mathbf{26}$, and $s_{peer_p} = s_p$.
– Case 2: $C.PC_p = \mathbf{26} \wedge C.\text{CSSTATUS} \neq (0, s_p)$
In this case, since $C.\text{CSSTATUS} \neq (0, s_p)$ and $C'.\text{CSSTATUS} = (0, s_p)$, the step from $C$ must be the execution of Line **14** by some process $r$. However, IH:8 implies that $C.s_p \leq C.s_r$. Since $r$ writes $(0, s_r + 1)$ in CSSTATUS, it follows that $C'.\text{CSSTATUS} \neq (0, s_p)$, contradicting our initial assumption. We conclude that Case 2 cannot arise.
– Case 3: $C.PC_p = \mathbf{26} \wedge C.\text{CSSTATUS} = (0, s_p)$
IH:9 implies that $C.PC_{peer_p} \in \{\mathbf{6-8}, \mathbf{18}, \mathbf{19}, \mathbf{20}, \mathbf{20}::\mathbf{24}\} \vee (C.PC_{peer_p} \in \{\mathbf{20}::\mathbf{25}, \mathbf{20}::\mathbf{26}\} \wedge s_{peer_p} = s_p) \vee (C.PC_{peer_p} \in \{\mathbf{1}, \mathbf{17}\} \wedge C.\text{GO}[q] \neq -1)$. Even if $peer_p$ takes a step from $C$, this condition remains true in $C'$, as we explain below.
  – Suppose that $PC_{peer_p}$ changes from **8** in $C$ to **10** in $C'$. Then, IH:5 implies that $C'.\text{CSSTATUS} = (1, peer_p)$, which is impossible because $C.\text{CSSTATUS} = (0, s_p)$ and the step from $C$ is the execution of Line **8**.
  – Suppose that $peer_p$ takes a step from $C$ when $C.PC_{peer_p} = \mathbf{20}::\mathbf{24}$. Since $C.\text{CSSTATUS} = (0, s_p)$, it follows that $C'.PC_{peer_p} = \mathbf{20}::\mathbf{25}$ and $C'.s_{peer_p} = C'.s_p$.

- Suppose that $peer_p$ takes a step from $C$ when $C.PC_{peer_p} = \mathbf{20}$::$\mathbf{26}$. Then, since $C.\text{CSStatus} = (0, s_p)$ and IH:9 implies that $C.s_{peer_p} = C.s_p$, it follows that $C'.\text{CSStatus} = (1, peer_p)$, contradicting our assumption that $C'.\text{CSStatus} = (0, s_p)$.
- Suppose that $peer_p$ takes a step from $C$ when $C.PC_{peer_p} = \mathbf{1}$. Then, IH:9 implies that $\text{Go}[peer_p] \neq -1$, which together with IH:3 implies that $status_{peer_p} \neq good$. It follows that, in its step from $C$, $peer_p$ invokes $\texttt{recover}_{peer_p}()$ method; so, $C'.PC_{peer_p} = \mathbf{17}$ (and $C'.\text{Go}[peer_p] \neq -1$).

10. Proof that Condition 10 holds in $C'$
Assume that $C'.PC_p \in \{\mathbf{28}, \mathbf{29}\}$. We consider two cases.
- Case 1: $C.PC_p \notin \{\mathbf{28}, \mathbf{29}\}$
Since $C.PC_p \notin \{\mathbf{28}, \mathbf{29}\}$, the step from $C$ must be $p$'s execution of Line **27** and $C'.g_p = C.\text{Go}[peer_p] \notin \{0, -1\}$. Since IH:3 implies that $-1 \leq C.\text{Go}[peer_p] < C.\text{Token}$, it follows that $1 \leq C'.g_p < C'.\text{Token}$.
- Case 2: $C.PC_p \in \{\mathbf{28}, \mathbf{29}\}$
In this case, IH:10 implies that $1 \leq g_p < \text{Token}$ in $C$. Since Token is possibly modified only at Line **3**, where the CAS instruction ensures that the value of Token does not decrease, the inequality continues to hold in $C'$, i.e., $1 \leq C'.g_p < C'.\text{Token}$.

11. Proof that Condition 11 holds in $C'$
Assume that $C'.PC_p = \mathbf{29}$ and consider the following cases.
- Case 1: $C.PC_p \neq \mathbf{29}$
Since $PC_p$ is **29** in $C'$ but not in $C$, it follows that the step from $C$ must be $p$'s execution of Line **28** and $C.\text{CSStatus} = (1, peer_p)$. IH:5, together with $C.\text{CSStatus} = (1, peer_p)$, implies that $C.PC_{peer_p} \notin \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}]$ and $C.\text{Go}[peer_p] \neq -1$. IH:3, together with $C.\text{Go}[peer_p] \neq -1$, implies that $PC_{peer_p} \notin \{\mathbf{2\text{-}4}, \mathbf{23}\}$. Taken together, the above facts imply that $C'.PC_{peer_p} \notin \{\mathbf{3}, \mathbf{4}, \mathbf{5}\}$. This, together with $C'.\text{CSStatus} = C.\text{CSStatus} = (1, peer_p)$, trivially implies Condition 11 in $C'$.
Case 2: $C.PC_p = \mathbf{29}$
In this case, by the induction hypothesis, the consequent of Condition 11's implication holds in $C$. We argue below that a step from $C$ cannot invalidate the consequent.
  - Suppose that $peer_p$ takes a step from $C$ when $C.PC_{peer_p} = \mathbf{2}$. Then, $PC_{peer_p}$ becomes **3** in $C'$, but then $1 \leq g_p < tok_{peer_p}$ will hold in $C'$ because $peer_p$'s execution of Line **2** sets $tok_{peer_p}$ to Token, and IH:10 implies that $1 \leq g_p < \text{Token}$.
  - Suppose that $peer_p$ takes a step from $C$ when $C.PC_{peer_p} = \mathbf{4}$. Then, $PC_{peer_p}$ becomes **5** in $C'$, but then $1 \leq g_p < \text{Go}[peer_p]$ will hold in $C'$ because $peer_p$'s execution of Line **4** sets $\text{Go}[peer_p]$ to $tok_{peer_p}$, while IH:11 and IH:6 imply $1 \leq g_p < tok_{peer_p} < \text{Token}$.
  - Suppose that $peer_p$ takes a step from $C$ when $C.PC_{peer_p} = \mathbf{5}$. Then, $PC_{peer_p}$ becomes **6** in $C'$, but then IH:11 ensures $g_p < \text{Go}[peer_p]$ in $C'$.
  - Suppose that $C.PC_{peer_p} \in \{\mathbf{6}, \mathbf{7}, \mathbf{8}\}$ and $C.g_p \neq C.\text{Go}[peer_p]$, and some process $q$ takes a step from $C$ where it changes $\text{Go}[peer_p]$ to $g_p$. This scenario is impossible because $g_p$ is non-zero (by IH:10) and Line **29**, which is the only line of code where $q$ could possibly change $\text{Go}[peer_p]$, can only change $\text{Go}[peer_p]$ to 0.
  - Suppose that $C.PC_{peer_p} \in \{\mathbf{6}, \mathbf{7}, \mathbf{8}\} \wedge C.\text{CSStatus} = (1, peer_p)$. Since $peer_p$ is the only process that can change the value in CSStatus and that too only at Line **14**, $C.PC_{peer_p} \in \{\mathbf{6}, \mathbf{7}, \mathbf{8}\}$ implies that will continue to have the value $(1, peer_p)$ in $C'$.

12. Proof that Condition 12 holds in $C'$
We argue the correctness of the condition for the case when $\min(\text{Registry}) = (*, \infty)$ in $C$. In that case the step changes the Registry so that $\min(\text{Registry}) \neq (*, \infty)$ in $C'$. Since the step is by $p$, it could only be the case that $p$ took a step to write to $\text{Registry}[p]$, otherwise Registry cannot change from an empty array to the one containing an element since a process writes only to its own cell in the Registry. Such a write could happen only at Line **5** because at Lines **11**, **19** $p$ could write only $(p, \infty)$. It follows that $PC_p = \mathbf{6}$ in $C'$, which satisfies the condition.

Next we argue the correctness of the condition for the case when $\min(\text{Registry}) \neq (*, \infty)$ in $C$ as below

Case A $\exists q, \text{CSStatus} = (1, q)$ in $C$.
Suppose $\text{CSStatus} = (1, q)$ in $C'$, it follows that the step didn't affect the truth value of the condition because the step was by a process $r \neq q$. Therefore assume $\text{CSStatus} \neq (1, q)$ in $C'$. CSStatus could change from $(1, q)$ only due to the write operation by some process at Line **14**. This process could only be $q$ itself, because if it were some process $r \neq q$, then by IH:5 and $PC_r = \mathbf{14}$, $\text{CSStatus} = (1, r) \neq (1, q)$ in $C$, a contradiction. It follows that $PC_q = \mathbf{15}$ in $C'$ and the condition holds in $C'$.

Case B $\forall q, \text{CSStatus} \neq (1, q)$ in $C$.
By IH:2, $\text{CSStatus} = (0, \text{Seq})$ in $C$.

Suppose $\exists r, \text{CSStatus} = (1, r)$ in $C'$. It follows that the execution of Line **26** by some process changed the value of CSStatus to $(1, r)$ and the condition holds in $C'$.

Assume $\forall q$, CSStatus $\neq (1, q)$ in $C$. By IH:12, $\exists q$, $(PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \text{Go}[q] \neq -1) \vee PC_q \in \{\mathbf{6}, \mathbf{15}, \mathbf{18\text{-}20}, \mathbf{24}\}$ $\vee$ $(PC_q \in \{\mathbf{25}, \mathbf{26}\} \wedge \text{CSStatus} = (0, s_q))$ in $C$.
Suppose the step changing $C$ to $C'$ was not because of $q$, the condition continues to hold in $C'$ as it held in $C$.
Suppose the step changing $C$ to $C'$ was a crash step of $q$. In the case when $PC_q \in \{\mathbf{6}, \mathbf{15}, \mathbf{18\text{-}20}, \mathbf{24}\}$ $\vee$ $(PC_q \in \{\mathbf{25}, \mathbf{26}\} \wedge \text{CSStatus} = (0, s_q))$ in $C$, by IH:3, Go$[q] \neq -1$ in $C$, $PC_q = \mathbf{1}$ in $C'$ and Go$[q]$ remains unchanged. Otherwise $PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \text{Go}[q] \neq -1$ in $C$, $PC_q = \mathbf{1}$ in $C'$ and Go$[q]$ remains unchanged. In either case the step wouldn't change Go$[q]$, and the condition would continue to hold in $C'$.
Suppose the step changing $C$ to $C'$ was a normal step of $q$. In that case, if $PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \text{Go}[q] \neq -1$ in $C$, $PC_q \in \{\mathbf{17}, \mathbf{18}\}$ in $C'$ and Go$[q] \neq -1$. Therefore, the condition holds in $C'$. Otherwise, $PC_q \in \{\mathbf{6}, \mathbf{15}, \mathbf{20}\}$ and $q$ makes a call to $\texttt{promote}_q()$ so that changing $PC_q$ to **24** and the condition being held in $C'$. In another case, $PC_q \in \{\mathbf{18}, \mathbf{19}\}$ in $C$ and $PC_q \in \{\mathbf{19}, \mathbf{20}\}$ in $C'$. Otherwise, $PC_q = \mathbf{24}$ in $C$ and $q$ reads CSStatus during the step, so that CSStatus $= (1, s_q)$ (here $s_q$ is initialized to the second component of CSStatus by $q$ during the step) and $PC_q = \mathbf{25}$ in $C'$. In another case, $PC_q = \mathbf{25}$ in $C$ and it changes to **26** in $C'$ (without modifying CSStatus). In yet another case $PC_q = \mathbf{26}$ in $C$ and the CAS changes CSStatus to $(1, peer_q)$. By IH:6, $peer_q \in \mathcal{P}$. In all these cases it follows that the condition holds in $C'$.

13. Proof that Condition 13 holds in $C'$
We first argue the correctness of the condition for the case when CSStatus $\neq (1, p)$ in $C$ and CSStatus $= (1, p)$ in $C'$. Since the step changes CSStatus $\neq (1, p)$ to $(1, p)$, it happens because some process $q$ executed Line **26**. Therefore, $PC_q = \mathbf{26}$ in $C$ and changes to **27** because the CAS succeeds. Since $q$ wrote $(1, p)$ into CSStatus due to the CAS by an inspection of the algorithm it follows that $peer_q = p$ in $C$, which remains the same in $C'$. It follows that in $C'$ $\exists q, PC_q = \mathbf{27} \wedge peer_q = p$. Thus, the condition holds in $C'$.

Next we argue the correctness of the condition for the case when CSStatus $= (1, p)$ and Go$[p] \neq 0$ in $C$. By IH:13, $\exists q, (PC_q \in \{\mathbf{18\text{-}20}, \mathbf{24}\} \vee (PC_q = \mathbf{27} \wedge peer_q = p) \vee (PC_q \in \{\mathbf{28}, \mathbf{29}\} \wedge peer_q = p \wedge g_q = \text{Go}[p])$ $\vee$ $(PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \text{Go}[q] \neq -1))$. Suppose $C$ changes to $C'$ due to a crash step of $q$. In that case, if $(PC_q \in \{\mathbf{18\text{-}20}, \mathbf{24}\} \vee (PC_q = \mathbf{27} \wedge peer_q = p) \vee (PC_q \in \{\mathbf{28}, \mathbf{29}\} \wedge peer_q = p \wedge g_q = \text{Go}[p])$, in all of these cases, by IH:3, Go$[q] \neq -1$ in $C$. Since the crash doesn't change Go$[q]$, it follows that $PC_q = \mathbf{1}$ and Go$[q] \neq -1$ in $C'$. Therefore, the condition holds in $C'$. Suppose $C$ changes to $C'$ due to a normal step of $q$. In that case, if $PC_q \in \{\mathbf{1}, \mathbf{17}, \mathbf{18}, \mathbf{19}\}$ in $C$, it is the case that $PC_q \in \{\mathbf{17}, \mathbf{18}, \mathbf{20}\}$ in $C'$. Otherwise, a call from Line **20** to $\texttt{promote}_q()$ changes $PC_q$ to **24**. If $PC_q = \mathbf{24}$ in $C$, then since CSStatus $= (1, p)$ in $C$, the step changes $PC_q$ to **27** while setting $peer_q = p$. Suppose $PC_q = \mathbf{27}$ and $peer_q = p$ in $C$, and as we already assumed above that CSStatus $= (1, p)$ and Go$[p] \neq 0$. From IH:5, we have Go$[p] \neq -1$ in $C$, because otherwise CSStatus $\neq (1, p)$. It follows that the step changes $PC_q$ to **28** while setting $g_q$ to Go$[p]$. Since CSStatus $= (1, p)$ in $C$, if $PC_q = \mathbf{28}$, $peer_q = p$, and $g_q = \text{Go}[p]$, $PC_q$ changes to **29** because the **if** condition is met at Line **28**. If $PC_q = \mathbf{29}$, $peer_q = p$, and $g_q = \text{Go}[p]$ in $C$, the CAS at Line **29** by $q$ succeeds and Go$[p]$ changes to 0 in $C'$ thereby satisfying the condition vacuously. It follows that in all of the above cases the condition continues to hold in $C'$.

Thus, by induction it follows that the invariant holds in every configuration of every run of the algorithm.