Verifying General-Purpose RCU for Reclamation in Relaxed Memory Separation Logic

JAEHWANG JUNG*, KAIST, Republic of Korea
SUNHO PARK, KAIST, Republic of Korea
JANGGUN LEE, KAIST, Republic of Korea
JEHO YEON, KAIST, Republic of Korea
JEEHOON KANG, KAIST, Republic of Korea

Read-Copy-Update (RCU) is a critical synchronization mechanism for concurrent data structures, enabling efficient deferred memory reclamation. However, implementing and using RCU correctly is challenging due to its inherent concurrency complexities. While previous work verified RCU, they either relied on unrealistic assumptions of sequentially consistent (SC) memory model or lacked three key features of general-purpose RCU libraries: modular specification, switchable critical sections, and concurrent writer support.

We present the first formal verification of a general-purpose RCU in realistic *relaxed memory consistency* (RMC), addressing the challenges posed by these features. To achieve modular specification that encompasses relaxed behaviors, we extend existing SC specifications to account for explicit synchronization. To support switchable critical sections, which require read-after-write (RAW) synchronization, we introduce a reasoning principle for RAW-synchronizing *SC fences*. Using this principle, we also present the first formal verification of Peterson's mutex in RMC. To support concurrent writers performing partially ordered writes, we avoid assuming a total order of links and instead formulate invariants based on per-node incoming link histories. Our proofs are mechanized in the iRC11 relaxed memory separation logic, built upon Iris, in Rocq.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Logic and verification**; **Concurrency**.

Additional Key Words and Phrases: separation logic, relaxed memory, Read-Copy-Update

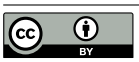
ACM Reference Format:
Jaehwang Jung, Sunho Park, Janggun Lee, Jeho Yeon, and Jeehoon Kang. 2025. Verifying General-Purpose RCU for Reclamation in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 9, PLDI, Article 147 (June 2025), 25 pages. https://doi.org/10.1145/3729246

1 Introduction

Concurrent programming is challenging due to the non-deterministic interleaving of instructions across threads and the asynchronous communication of memory writes. This difficulty is particularly pronounced when reclaiming memory that is no longer in use, as it requires careful reasoning about the accessibility of memory blocks to ensure that all accesses to a block strictly happen before its reclamation. To simplify this, *concurrent reclamation algorithms* such as *Read-Copy-Update* (RCU) [32, 33] and *Hazard Pointers* (HP) [38] have been proposed.

*Now at Rebellions Inc.

Authors' Contact Information: Jaehwang Jung, KAIST, Daejeon, Republic of Korea, jaehwang.jung@kaist.ac.kr; Sunho Park, KAIST, Daejeon, Republic of Korea, sunho.park@kaist.ac.kr; Janggun Lee, KAIST, Daejeon, Republic of Korea, janggun.lee@kaist.ac.kr; Jeho Yeon, KAIST, Daejeon, Republic of Korea, jeho.yeon@kaist.ac.kr; Jeehoon Kang, KAIST, Daejeon, Republic of Korea, jeehoon.kang@kaist.ac.kr.



ACM 2475-1421/2025/6-ART147
https://doi.org/10.1145/3729246

Essentially, these reclamation algorithms reduce the complexity of reasoning about concurrent reclamation to that of reasoning about reachability. Concurrent data structures can simply *retire* memory blocks that are unreachable from the data structure's root and hand them over to the reclamation algorithm. The algorithm then waits for concurrent threads to finish accessing the retired blocks through their local pointers and then reclaims the blocks.[1]

Verification of concurrent reclamation. Reclamation algorithms, however, are inherently complex concurrent programs themselves and require careful reasoning to ensure correctness. Consequently, several prior works have focused on verifying these algorithms.

Jung et al. [20] formally verified RCU and HP against modular specifications with the Iris separation logic framework [22, 24, 29] in the Rocq Prover. However, their verification relies on the strongly synchronized *sequential consistency* (SC) memory model, which unrealistically assumes synchronous communication of memory writes. All threads observe writes in the order they are issued in SC, significantly simplifying reasoning about concurrent programs.

Alglave et al. [2] verified an RCU implementation against an axiomatic specification in a more realistic *relaxed memory consistency* (RMC) model. RMC permits out-of-order observations of writes due to asynchronous communication. However, their verification was conducted directly at the memory model and lacks reasoning principles for concurrent data structures using RCU, limiting application to small test programs.

Tassarotti et al. [46] verified a single-writer linked list using RCU in a separation logic for RMC. However, their verification does not support three important features of general-purpose RCU libraries [10]: **(1)** *Modular specification*: RCU and client data structures should be specified and verified independently; **(2)** *Switchable critical sections*: threads can dynamically enter and exit participation in RCU; and **(3)** *Concurrent writers*: multiple threads can concurrently write to data structures protected by RCU.

We present the first formal verification of a general-purpose RCU implementation supporting these three features in RMC. To demonstrate its generality, we use this verified RCU to conduct the first formal verification of Michael and Scott [39]'s lock-free queue and Harris [18]'s lock-free linked list and its variants [19, 37] with reclamation in RMC. Additionally, we present the first formal verification in RMC of Peterson [43]'s mutex based on *read-after-write* (RAW) synchronization, a key mechanism for switchable critical sections in RCU. Our proofs, available as supplementary material [21], are mechanized in the iRC11 relaxed memory separation logic [8], built upon the Iris framework [22, 24, 29], in the Rocq Prover. This endeavor reveals the significant challenges RMC poses to the verification of these three features, which we address as follows.

Modular specification (§3). Jung et al. verified an RCU implementation against a modular specification in separation logic, but in SC. Tassarotti et al. verified a data structure using RCU in RMC, but their verification is monolithic and not readily applicable to other data structures. We bridge this gap by adapting Jung et al.'s specifications to RMC. Remarkably, only minor modifications to the original specification are needed to accommodate the relaxed behaviors permitted in RMC.

Following Jung et al., our specifications consist of a low-level *base* specification and a high-level *traversal* specification. The base specification formalizes the core guarantee of RCU: an object accessed in a critical section is protected from reclamation if it has not been retired before the critical section begins. Building upon this, the traversal specification encapsulates common

[1] *Automatic* garbage collection algorithms, based on tracing or reference counting, aim to completely address the complexity of reclamation at the cost of time and space overheads, which can be undesirable in performance-critical systems. This work focuses on lightweight *manual* reclamation algorithms.

reasoning patterns for traversal, where a critical section ensures safe accesses to objects reachable by traversing a shared data structure.

The key additions to the original SC specifications lies in handling the protection of *stale pointer values* made observable due to relaxed memory behaviors. In the base specification, we introduce a low-level mechanism to transfer observations about the detachment of retired nodes, thereby modeling the guarantee that an RCU critical section prevents reading excessively stale pointers that are not protected. In the traversal specification, this mechanism is abstracted into a *link view* associated with each critical section. These changes enable our specifications to remain modular and reusable while accounting for the complexities of RMC.

Switchable critical sections (§4). Dynamically entering and exiting RCU critical sections requires *mutual exclusion* with threads reclaiming retired nodes, to prevent a retired node from being both reachable and reclaimed simultaneously. For mutual exclusion, as proven by Attiya et al. [3], either atomic write-after-read (AWAR) or read-after-write (RAW) synchronization is necessary. RCU implementations typically employ RAW synchronization to reduce contention. This involves a thread writing to a shared location to signal its intent, followed by reading another location to observe other thread's state.

While prior work verified algorithms based on RAW synchronization in SC [20], reasoning about it in RMC is challenging as it requires global reasoning over the totally ordered history of *SC fences*, which enforce RAW ordering. Prior verification efforts have often circumvented this complexity. For instance, Dalvandi et al. [6] verified a variant of Peterson [43]'s mutex, but simplified to use AWAR synchronization; and Tassarotti et al. [46] verified a variant of RCU in which every thread is always in a critical section. To ensure progress of reclamation under this constraint, this algorithm requires each thread to periodically declare that it holds no local pointers shared objects.

We introduce two proof rules for RAW-synchronizing SC fences in RMC. The first rule captures the low-level semantics, enforcing the ordering of memory operations around the fences. Built on top of it, the second rule captures the high-level property of a totally ordered history of thread observations through SC fences. We demonstrate the effectiveness of our proof rules through a case study of Peterson's mutex and epoch-based RCU supporting switchable critical sections.

Concurrent writers (§5). Existing verification for traversal-based data structures protected by RCU unrealistically assume a total order of writes, by assuming SC [20] or a single writer [46]. However, this simplification does not hold in general under RMC for fine-grained concurrent data structures such as lock-free lists [18, 37].

We generalize the existing RCU traversal verification to handle partially ordered writes, thereby supporting concurrent writers in RMC. The key idea is to formulate invariants on the data structure's history based on the incoming edges of *each node*, rather than relying on a global total ordering of all writes. This can express the key property of RCU traversal even in RMC: a node can be removed only if it has become unreachable and remains unreachable. Furthermore, this approach yields simpler proof than existing verification because it locally asserts the necessary properties instead of deriving them from the global history of all writes.

Outline. In §2, we review the necessary background. In §3, §4, and §5, we elaborate on our main contributions discussed above. In §6, we conclude with a discussion on related and future work.

2 Background

2.1 General-Purpose RCU

We first overview the interface and an implementation of RCU under SC memory model.

Algorithm 1 Treiber's lock-free stack with RCU

```
1:  struct Node<T>
2:      data: T
3:      next: Node<T>*
4:  struct Stack<T>
5:      head: Atomic<Node<T>*>
6:  function push(s, val)
7:      let n := new Node {data = val; next = null}
8:      loop
9:          let h := s.head.load(rlx); (*n).next := h
10:         if s.head.cas(h, n, rel) then return
11: function pop(s)
12:     rcu_lock(tid)
13:     loop
14:         let h := s.head.load(acq)
15:         if h == null then
16:             rcu_unlock(tid); return None
17:         let data := (*h).data; let n := (*h).next
18:         if s.head.cas(h, n, rlx) then
19:             retire(h); rcu_unlock(tid)
20:             return Some(data)
```

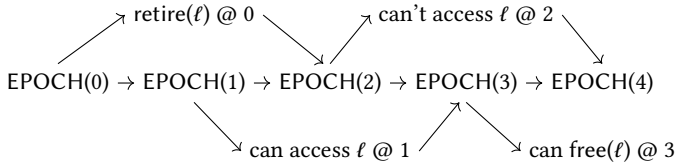


Fig. 1. Illustration of the epoch invariant

Algorithm 2 An epoch-based RCU algorithm

```
1:  global variables
2:      EPOCH: Atomic<Epoch>
3:      RETIRED: ConcurrentStack<(void*, Epoch)>
4:      LOCALS: [Atomic<Epoch>]
5:  function rcu_lock(tid)
6:      var e := EPOCH.load(rlx)
7:      loop
8:          LOCALS[tid].store(e, rlx)
9:          fence(sc)
10:         var e' := EPOCH.load(rlx)
11:         if e == e' then return
12:         else e := e'
13: function rcu_unlock(tid)
14:     LOCALS[tid].store(-1, rel)
15: function retire(tid, ptr)
16:     RETIRED.push((ptr, LOCALS.load(rlx)))
17:     if (some condition) then do_reclamation()
18: // Internal functions
19: function do_reclamation()
20:     let ge := try_advance()
21:     for (r, e) ∈ RETIRED.pop_all() do
22:         if ge < e + 3 then
23:             RETIRED.push((r, e))
24:         else
25:             free(r)
26: function try_advance()
27:     let ge := EPOCH.load(rlx)
28:     fence(sc)
29:     for local_epoch ∈ LOCALS do
30:         let e := local_epoch.load(acq)
31:         if e ≥ 0 && e ≠ ge then
32:             return ge
33:     EPOCH.cas(ge, ge+1, rel)
34:     return ge+1
```

Interface and usage. Algorithm 1 shows Treiber [48]'s lock-free stack algorithm protected with RCU. We first explain the code without RCU and then the shaded parts describing how RCU is integrated. Treiber's stack is a singly linked list with push and pop operations. Each node consists of data and the pointer to the next node. To pop a node, a thread first loads the stack's head from its *atomic location*, which allows concurrent access (line 14). (See §2.2 for *access modes* such as rlx, rel, and acq.) If the stack is empty, the operation returns None (line 16). Otherwise, the thread

obtains the data and attempts to move the head pointer to the next node using a compare-and-swap (CAS) operation (line 18). If successful, it returns Some(data) (line 20).

When using RCU for memory reclamation, each access to shared memory must occur within a *critical section*, denoting a region where the memory is protected by RCU and will not be reclaimed. Specifically, *all* memory blocks reachable by traversing the data structure remain protected until the end of the critical section. In Algorithm 1, a thread first calls rcu_lock() to indicate that it is entering a critical section (line 12). After detaching h from the stack, the thread calls retire(h) to schedule its reclamation (line 19). Finally, when finished accessing shared memory, the thread calls rcu_unlock() to signal that it is leaving the critical section (line 16 and line 19).

Epoch-based algorithm. Algorithm 2 presents a variant of epoch-based [4, 14] RCU algorithm due to Parkinson et al. [42]. The algorithm maintains a global EPOCH, which represents the logical time. When a thread enters a critical section, it is assigned the current EPOCH value. The EPOCH value is occasionally incremented while maintaining the *epoch invariant* illustrated in Fig. 1: the EPOCH increments to $e + 2$ only after all the critical sections with epoch e have ended. By this invariant, a memory block ℓ made unreachable and retired at epoch e cannot be accessed at epoch $e + 2$, *i.e.*, it becomes *expired* at epoch $e + 2$. To reclaim ℓ safely, however, we should wait until $e + 3$ $(= (e + 1) + 2)$, because a thread in a critical section of epoch $e + 1$ may still access ℓ.

The epoch invariant and the reclamation process are implemented as follows. In rcu_lock(), the thread with ID tid first obtains a snapshot e of the global EPOCH (line 6) and stores e in its slot in the LOCALS, the list of local epochs (epoch of critical section) of each thread (line 8). However, the thread cannot enter the critical section yet because these two steps are not atomic and another thread may update EPOCH to $e + 2$ or higher in between, violating the epoch invariant. Therefore, the thread must validate that EPOCH has not changed (line 11). If validation succeeds, the thread enters the critical section; otherwise, it updates its local epoch with the newly observed EPOCH (line 10) and retries. Conversely, in rcu_unlock(), a thread updates its local epoch with the sentinel value -1 to indicate that it is leaving the critical section (line 14).

In retire(), a thread requests reclamation of the memory block ptr by pushing it to RETIRED, the set of retired pointers annotated with the retiring thread's epoch (line 16). This function also occasionally calls do_reclamation() to reclaim retired pointers (line 17). The do_reclamation() function first tries to advance the global EPOCH and receives the updated epoch value (line 20). It then checks whether each retired memory block in RETIRED can be reclaimed according to the epoch invariant (line 22). Unreclaimable blocks are pushed back to RETIRED (line 23).

The try_advance() function attempts to advance the global EPOCH by one and returns the updated epoch. It first obtains a snapshot ge of EPOCH (line 27), and then checks whether any thread has a local epoch less than ge, ignoring the sentinel value (line 31). If so, EPOCH should not be incremented. Otherwise, it attempts to increment EPOCH by one using a CAS (line 33). Regardless of the CAS's result, it returns the updated epoch ge+1 (line 34).[2]

RAW synchronization. The correctness of switchable critical sections depends on the RAW synchronization in rcu_lock(): a store of an epoch value e to the local epoch slot (line 8), followed by a load of the global epoch for validation (line 10). This ensures a form of *mutual exclusion* with other threads that invoke try_advance() to advance EPOCH from $e + 1$ to $e + 2$ (where the value of the variable e is $e + 1$). Specifically, it guarantees one of the following: **(1)** The store of e to the local epoch slot in rcu_lock() (line 8) happens before the load of the local epochs in try_advance() (line 30). This ensures that rcu_unlock() happens before the other thread loads the local epoch and increment EPOCH to $e + 2$. **(2)** The load of EPOCH in try_advance() (line 27) happens before

[2]The global epoch cannot advance further to ge+2 or higher because the current thread's local epoch is ge.

the load of EPOCH in rcu_lock() (line 10). This causes rcu_lock() to read $e + 1$ or higher and fail validation. In either case, the global epoch cannot advance to $e + 2$ while a thread is in epoch e.

2.2 Semantics of RMC

We present the core concepts of the view-based operational semantics [8, 25, 26] for the RC11 memory model [30] for C/C++. For a comprehensive treatment, we refer readers to Dang [7].

A fundamental characteristic of RMC is that reads can observe old values rather than the most recent one. This behavior necessitates a more complex memory representation than SC. Instead of a map from locations to a single value, memory is represented as a finite map from locations to *sets of messages*. Each message consists of a value and other data we introduce later. RMC does not allow completely arbitrary access to old values, but enforces *coherence*: messages at each location are totally ordered ("modification order"), and once a thread has observed a particular message at a location, it cannot observe any earlier messages at that location. To establish this ordering, each message carries a *timestamp* $t \in \mathsf{Time}$, and the full type of memory is $\mathsf{Mem} \triangleq \mathsf{Loc} \xrightarrow{\mathsf{fin}} \mathsf{Time} \xrightarrow{\mathsf{fin}} \mathsf{Msg}$. For a message m, we use $m.\mathsf{time}$ to denote its timestamp. Each thread tracks its local observations through a *thread view* $V \in \mathsf{View} \triangleq \mathsf{Loc} \xrightarrow{\mathsf{fin}} \mathsf{Time}$. During a read from ℓ, a thread can observe any message m from $\mathcal{M}(\ell)$ where $m.\mathsf{time} \geq V(\ell)$, and updates its view to incorporate $m.\mathsf{time}$. When performing a write, the written message receives a fresh timestamp $t > V(\ell)$.

To transfer one thread's observations to another, *i.e.*, establish a *happens-before* relation, RMC provides two synchronization mechanisms: *release-acquire atomics* and *SC fences*.[3]

Release-acquire atomics. For Treiber's stack (Algorithm 1), a thread that pops a node must see the observation that its data and next fields are initialized by the thread that pushed it (line 7 and line 9). This is achieved by the *release-acquire* synchronization: push uses rel mode for writes (line 10), while pop uses acq mode for reads (line 14). These rel-acq pairs create a happens-before relationships through *message views*. When a thread does a release write, it embeds its view into the message view, and when it does an acquire read, it incorporates the message view into it's own view through a join operation: $V_1 \sqcup V_2 \triangleq \ell \mapsto \max(V_1(\ell), V_2(\ell))$. A cas operation in acqrel mode both acquires the view of the old message and releases the current thread's view. Operations executed in rlx (relaxed) mode do not interact with message views.

SC fences. For RAW synchronization between rcu_lock() and try_advance() in epoch-based RCU (§2.1), both functions use SC fences (line 9 and line 28) to establish a happens-before relationship through the global *SC view*. Specifically, an SC fence updates both the SC view and the executing thread's view to their join. Therefore, all SC fences are totally ordered by the happens-before relation, according to their execution order. This ensures the informal reasoning based on the case analysis of execution order remains valid for RMC. Without the SC fences, both the load of the global epoch in rcu_lock() (line 10) and the load of a local epoch in try_advance() (line 30) may read a stale value. This would allow a thread to enter critical section with an epoch e when the current global epoch is higher than $e + 1$, violating the epoch invariant.

2.3 Separation Logic for RMC

We present the key aspects of the iRC11 separation logic [8], which builds on the Iris framework [22] to reason about the RC11 memory model.

At the heart of iRC11 are *view-dependent* assertions of type *vProp*, which are interpreted in the context of the asserting thread's current view. For example, the *seen-view* assertion $\sqsupseteq V$ states that

[3]We do not consider the *SC access mode* in this work because its interaction with the other access modes introduces substantial complexity [30]. We find that SC fences provide adequate support for typical RAW synchronization.

the thread's view is at least V, where the partial order is defined as $V_1 \sqsubseteq V_2 \triangleq \forall \ell.\, V_1(\ell) \leq V_2(\ell)$. The seen-view assertion is *persistent*, meaning it always holds once established and does not have exclusive ownership. An important property of iRC11's view-dependent propositions is that they are *view-monotone*: if a proposition P holds under view V_1, and $V_1 \sqsubseteq V_2$, then P also holds under the more up-to-date view V_2.

To reason about Atomic locations, iRC11 provides the *atomic points-to* assertion $\ell \mapsto_{\mathsf{at}} h$. This assertion represents ownership of location ℓ along with its history $h \in \mathsf{Time} \xrightarrow{\mathsf{fin}} \mathsf{Msg}$. This is in contrast to the points-to assertion of SC separation logics, $\ell \mapsto v$, which tracks only the *latest value* v. Atomic points-to assertion is equipped with proof rules for reading and writing in various access modes. For example, the following rule describes the behavior of an acquire-load (simplified):

$$\{\sqsupseteq V * \ell \mapsto_{\mathsf{at}} h\}\, \ell.\mathsf{load}(\mathsf{acq})\, \{v.\, \exists m \in h.\, v = m.\mathsf{value} * V(\ell) \leq m.\mathsf{time} * \sqsupseteq m.\mathsf{view} * \ell \mapsto_{\mathsf{at}} h\}$$

The rule ensures that a thread's read satisfies the coherence requirement: the chosen message m's timestamp is no earlier than the thread's current view on ℓ. Furthermore, the thread incorporates the message's view m.view into its own view, as expressed by the $\sqsupseteq m.\mathsf{view}$ assertion.

Invariants and the view-at modality. For SC, the standard method for reasoning about shared state is *invariants*, denoted as $\boxed{I}$, which states that the proposition I always holds. A thread may rely on and modify I during atomic memory operations, provided it reestablishes I afterward.

However, in RMC, not every proposition can be put inside an invariant as an assertion that holds in one thread's view may not necessarily hold in another thread's view. For example, $\boxed{\sqsupseteq V}$ does not hold. Thus, only *objective* assertions can be put inside invariants, meaning they must be independent of the asserting thread's view.

To sidestep the restriction on invariants for a non-objective proposition P, one can make P objective by asserting it at a fixed view V using the *view-at* modality $@_V$ via VA-intro:

(VA-objective)	(VA-intro)	(VA-elim)
$\mathsf{objective}(@_V P)$	$P \vdash \exists V.\, \sqsupseteq V * @_V P$	$\sqsupseteq V * @_V P \vdash P$

$@_V P$ asserts that P holds at view V, and by view-monotonicity, this implies that P holds at any view greater than V. VA-elim formalizes the above intuition, which removes $@_V$ when combined with a $\sqsupseteq V$. For example, a common invariant encountered when working with release-acquire synchronization via a location ℓ is: $\boxed{\ell \mapsto_{\mathsf{at}} \{m\} * @_{m.\mathsf{view}} P}$.[4] The releasing thread applies VA-intro and writes a message m whose view includes the view required for P The reading thread does an acquire load and applies VA-elim to access P.

3 Specifications for RCU

We present two modular specifications for RCU: the low-level *base* specification (§3.1, Fig. 2) and the high-level *traversal* specification (§3.2, Fig. 4). Our specifications build on Jung et al. [20]'s specifications in SC with a few adaptations for relaxed behaviors in RMC.[5] The green shades and ~~strikeouts~~ highlight the additions and removals for RMC, respectively.

3.1 Base Specification

The base specification aims to directly model the standard specification of RCU [34]: if the retirement of an object does not happen before the start of the critical section, then the object is protected by the critical section, *i.e.*, its reclamation happens after the end of the critical section.

[4] Actually, atomic points-to assertion should be wrapped in view-at modality as it is not objective. iRC11 provides accesses rules for atomic points-to under view-at modality. However, we omit such details for brevity of presentation.

[5] We adapted the presentation of Jung et al. [20]'s SC specifications to emphasize the similarities to our specifications in RMC. Notably, we split the Managed predicate into RetirePerm and BlockInfo.

$\mathsf{Inactive}(tid : \mathsf{TId}) : \mathit{vProp}$ $\quad$ $\mathsf{Guard}(tid : \mathsf{TId},\ X : \mathsf{Set}(\mathsf{AId}),\ G : \mathsf{Loc} \xrightarrow{\mathsf{fin}} \mathsf{AId}) : \mathit{vProp}$

$\mathsf{RcuState}(R : \mathsf{Set}(\mathsf{AId})) : \mathit{vProp}$ $\quad$ $\mathsf{BlockRes} \triangleq (\mathsf{Loc} \times \mathsf{AId}) \to \mathit{vProp}$

$\mathsf{BlockInfo}(\ell : \mathsf{Loc},\ a : \mathsf{AId},\ P : \mathsf{BlockRes}) : \mathit{vProp}$ $\quad$ $\mathsf{RetirePerm}(\ell : \mathsf{Loc},\ a : \mathsf{AId}) : \mathit{vProp}$

$\mathsf{Retired}(a : \mathsf{AId},\ Q : \mathit{vProp}) : \mathit{vProp}$

(RcuState-objective)
$\mathsf{objective}(\mathsf{RcuState}(R))$

(BlockInfo-persistent)
$\mathsf{persistent}(\mathsf{BlockInfo}(\ell, a, P))$

(Retired-persistent)
$\mathsf{persistent}(\mathsf{Retired}(a, Q)))$

(Retired-objective)
$\mathsf{objective}(\mathsf{Retired}(a, Q)))$

(rcu-register)
$$\frac{\{P(\ell, \_)\}\ \mathsf{free}(\ell)\ \{\mathsf{True}\}}{\mathsf{RcuState}(R) * (\forall a.\ a \notin A \mathrel{-\!\!*\!\!\!\Rightarrow} P(\ell, a)) \mathrel{-\!\!*\!\!\!\Rightarrow} \exists a.\ a \notin A * \mathsf{RcuState}(R) * \mathsf{BlockInfo}(\ell, a, P) * \mathsf{RetirePerm}(\ell, a)}$$

(rcu-retire)
$$\left\langle \begin{matrix} \mathsf{RcuState}(R) * \\ \mathsf{RetirePerm}(\ell, a) * Q \end{matrix} \right\rangle \mathsf{retire}(\ell) \left\langle \begin{matrix} \mathsf{RcuState}(R \cup \{a\}) * \\ \mathsf{Retired}(a, Q) \end{matrix} \right\rangle$$

(rcu-lock)
$$\left\langle \begin{matrix} \mathsf{RcuState}(R) * \\ \mathsf{Inactive}(tid) \end{matrix} \right\rangle \mathsf{rcu\_lock}(tid) \left\langle \begin{matrix} \exists X \subseteq R.\ \mathsf{RcuState}(R) * \\ \mathsf{Guard}(tid, X, \emptyset) \end{matrix} \right\rangle$$

(rcu-unlock)
$$\{\mathsf{Guard}(tid, \_, \_)\}\ \mathsf{rcu\_unlock}(tid)\ \{\mathsf{Inactive}(tid)\}$$

(Guard-protect)
$$\frac{a \notin X}{\mathsf{BlockInfo}(\ell, a, P) \vdash \mathsf{Guard}(tid, X, G) \mathrel{-\!\!*\!\!\!\Rightarrow} \mathsf{Guard}(tid, X, G[\ell \mapsto a])}$$

(Guard-not-retired)
$\mathsf{Guard}(\_, X, \_) * \mathsf{RetirePerm}(\ell, a) \vdash a \notin X$

(Guard-seen-retired)
$$\frac{a \in X}{\mathsf{Guard}(\_, X, \_) * \mathsf{Retired}(a, Q) \vdash Q}$$

(Guard-RetirePerm-agree)
$$\frac{G(\ell) = a}{\mathsf{Guard}(\_, \_, G) * \mathsf{RetirePerm}(\ell, a') \vdash a = a'}$$

(Guard-acc)
$$\frac{\{\exists V.\ \sqcup_V P(\ell, a) * P_1\}\ e\ \{v.\ \sqcup_V P(\ell, a) * P_2\} \qquad G(\ell) = a \qquad \mathsf{atomic}(e)}{\mathsf{BlockInfo}(\ell, a, P) \vdash \{\mathsf{Guard}(tid, \_, G) * P_1\}\ e\ \{v.\ \mathsf{Guard}(tid, \_, G) * P_2\}}$$

Fig. 2. The base specification of RCU.

Background: base specification in SC. We first focus on the unshaded parts of Fig. 2, which are common to both SC and RMC. A memory block has two kinds of identifiers: a *physical address* $\ell \in \mathsf{Loc}$ and an *allocation ID* $a \in \mathsf{AId}$ [46]. The physical address is the actual memory location of the block, while the allocation ID is a logical identifier that is unique for each allocation. That is, a same physical address can be reused for different allocations with distinct allocation IDs. When a block ℓ is allocated, it should be first (logically) registered to RCU by the RCU-REGISTER rule, which assigns a *fresh* ID a ($\notin A$ for any set A of allocated IDs). The rule then takes the block resource $P(\ell, a)$, a customizable assertion that consists of the physical resources for the block (*e.g.*, points-to assertions) and additional ghost resource (if necessary) to describe the block's properties. The Hoare triple for free() in its assumption is explained below. The rule returns the *block information* assertion $\mathsf{BlockInfo}(\ell, a, P)$, which records the (persistent) fact that a has physical address ℓ and is governed by P; and the *retire permission* assertion $\mathsf{RetirePerm}(\ell, a)$, which is a unique permission consumed by retire() (RCU-RETIRE), preventing double-retire.[6] The RcuState predicate, usually in the data structure's invariant, tracks the set R of allocations that has been retired so far.

The *guard* assertion, $\mathsf{Guard}(tid, X, G)$, asserts that the thread with ID $tid \in \mathsf{TId}$ is in a critical section. It is created upon entering a critical section (RCU-LOCK), replacing the $\mathsf{Inactive}(tid)$ assertion, and destroyed upon exiting (RCU-UNLOCK), restoring the $\mathsf{Inactive}(tid)$ assertion. When created, the guard learns the set X of *expired* allocation IDs (§2.1), the allocations that are retired sufficiently long ago and thus are *not* protected by the critical section. While the exact contents of X are implementation-specific (see §2.1 for an example), X must be a subset of the set R of all allocations retired before the critical section began. This subset relationship implies the standard RCU semantics: if a's retirement does not happen before the critical section starts, then $a \notin R$ and thus $a \notin X$, meaning that a is protected. This protection is provided by GUARD-PROTECT, which states that if $a \notin X$, then a can be added to the map G of protected pointers. Once protected, GUARD-ACC allows temporary access to the block's resource. When a block has become inaccessible for all threads, do_reclamation() can take the full control of the block resource. Specifically, it is used for free(), which is why RCU-REGISTER requires the Hoare triple that takes the block resource as precondition. The GUARD-RETIREPERM-AGREE rule guarantees that the allocation ID of a protected pointer remains constant during a critical section as it prevents reclamation and reallocation. In other words, the client does not suffer from the *ABA problem*, where different allocations could be mistaken for one another due to reuse of the same physical address.

To establish that $a \notin X$, GUARD-NOT-RETIRED says it suffices to show the block is not yet retired. Consider the example of verifying Treiber's stack (Algorithm 1), whose invariant is as follows:

$$\mathsf{Stack}(s) \triangleq \boxed{\exists L \in \mathsf{List}(\mathsf{Loc} \times \mathsf{AId}).\, s \mapsto (\textbf{match}\ L\ \textbf{with}\ [(\ell, \_); \_] \Rightarrow \ell \mid \_ \Rightarrow \mathsf{null}) * \mathop{\Large *}_{(\ell,a)\in L} (\mathsf{BlockInfo}(\ell, a, \mathsf{StackBlock}) * \mathsf{RetirePerm}(\ell, a) * \ldots) * \ldots}$$

$$\mathsf{StackBlock}(\ell, a) \triangleq \exists v, \ell'.\, \ell \mapsto \{\mathsf{data} = v, \mathsf{next} = \ell'\} * \ldots$$

This invariant asserts that the stack's head points to the first node in the list (if any) and includes RetirePerm for each block. This ensures that the pointer loaded at line 14 has the corresponding RetirePerm. Thus, we can establish protection of h using GUARD-NOT-RETIRED and GUARD-PROTECT, allowing us to read the node's data (line 17) via StackBlock accessed through GUARD-ACC.

Adaptation for RMC. The above proof sketch of Treiber's stack in SC is based on the assumption that the pointers to the popped nodes are not readable from the stack's head, which is directly

[6] $\langle P \rangle\, e\, \langle Q \rangle$ with angle brackets is a *logically atomic Hoare triple*, which asserts that e behaves as if it is atomic, allowing it to atomically access shared resources in invariants. We refer the readers to da Rocha Pinto et al. [5], Jung et al. [23] for details.

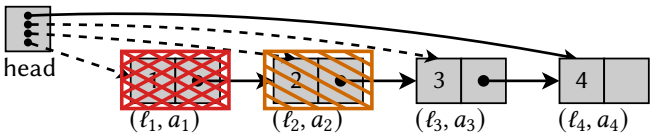


Fig. 3. Treiber's stack in RMC. The nodes $(\ell_4, a_4), \ldots, (\ell_1, a_1)$ are pushed, and then $(\ell_1, a_1), \ldots, (\ell_3, a_3)$ are popped. (ℓ_2, a_2) is retired (orange crossed out box), and (ℓ_1, a_1) is retired and reclaimed (red double-crossed out box). The dashed arrows indicate the stale pointer values.

enforced by the SC points-to assertion that states the exact current value of the head. In RMC, however, this assumption is too strong: a thread can read a stale pointer to a popped node and even a retired node. Fig. 3 illustrates a scenario where the nodes $(\ell_4, a_4), \ldots, (\ell_1, a_1)$ have been pushed, and the last three nodes are popped. (ℓ_1, a_1) is retired and then reclaimed, (ℓ_2, a_2) is retired but not yet reclaimed, and (ℓ_3, a_3) is detached but not yet retired. At this point, the latest message in the head is ℓ_4, but the stale messages for $\ell_1, \ldots, \ell_3$ still remain.

However, regardless of which pointer the thread reads, RCU ensures the safety of accessing the pointed node. This is because, if the node is reclaimed (*e.g.*, ℓ_1), it must have been expired before the critical section started, which is preceded by the node's retirement, and in turn the node's detachment. Therefore, the thread must have observed the detachment and thus could not have read the pointer to that node in the first place. In essence, RCU synchronization, together with the user's guarantee that they first make the node unreachable before retiring it, ensures that threads in critical section only reads pointers to unexpired nodes (*e.g.*, ℓ_4, ℓ_3, and possibly ℓ_2).

While Guard-not-retired ensures that the unpopped node ℓ_4 is unexpired and thus safe to access, reasoning about the popped nodes ℓ_1, ℓ_2, and ℓ_3 requires a more powerful rule that characterize the expired pointer set more precisely without resorting to RetirePerm. To address this, we adapt the base specification of RCU and the proof of Treiber's stack as follows.

First, we adapt the data structure invariant to account for stale pointers as follows:

$$\mathsf{Stack}(s) \triangleq \boxed{\exists H \in \mathsf{List}(\mathsf{Msg} \times \mathsf{AId}).\, s \mapsto_{\mathsf{at}} \mathsf{fmap}(\mathsf{fst}, H) * \left(\mathop{\Large *}_{(m,a) \in H} @_{m.\mathsf{view}} \mathsf{BlockInfo}(m.\mathsf{value}, \ldots) \right) * \left(\mathop{\Large *}_{(\ell,a) \in \mathsf{interp}(H)} \mathsf{RetirePerm}(\ell, a) \right) * \ldots}$$

Here, H represents the complete history of the stack's head, with each element consisting of a physical message and the allocation ID of the pointed node. The physical messages are tracked by the atomic points-to assertion. Since a thread may read stale pointers from H, the invariant must retain all BlockInfos for pointers in H. However, the invariant only keeps RetirePerm for the *latest* state of the stack, *i.e.*, the list of nodes reachable from the latest head value, as pop() must acquire RetirePerm to retire the detached node. For example, in Fig. 3, the latest state is $[(\ell_4, a_4)]$. The latest state can be computed by interpreting the history of the stack's head location:

$$\mathsf{interp}(H) \triangleq \mathsf{interp\_aux}(H, [\,]) \qquad \mathsf{interp\_aux}([\,], L) \triangleq L$$

$$\begin{aligned} \mathsf{interp\_aux}((m, a) :: H', L) \triangleq\ & \mathbf{match}\ L\ \mathbf{with} \\ & \mid \_ :: (\ell', a') :: L'\ \mathbf{when}\ \ell' = m.\mathsf{value} \\ & \quad \Rightarrow \mathsf{interp\_aux}(H', (\ell', a') :: L') \qquad \text{(pop)} \\ & \mid \_ \Rightarrow \mathsf{interp\_aux}(H', (m.\mathsf{value}, a) :: L) \quad \text{(push)} \end{aligned}$$

Second, we revise rcu-retire to record the observation of the retired block's detachment. It takes a precondition Q, instantiated with the detachment observation for the block a, and returns the

postcondition $\mathsf{Retired}(a, Q)$, which is an objective fact that a is retired and that Q was provided upon a's retirement. For Treiber's stack, Q can be simply defined as the observation of the message that pops the given node, which can be found from H by interpreting it. We then include these Retired assertions alongside RcuState in the invariant as follows:

$$\mathsf{Stack}(s) \triangleq \boxed{\ldots * \exists R.\, \mathsf{RcuState}(R) * (\circledast_{a \in R} \mathsf{Retired}(a, Q)) * \ldots}$$

Third, we use these Retired assertions to ensure that a node pointer loaded from the stack's head is safe to access, *i.e.*, it is not in X. For this purpose, we introduce a new rule Guard-seen-retired to get the detachment observation out of Retired assertions. If the node were in $X (\subseteq R)$, then the detachment observation would contradict the assumption that we loaded a pointer to the node. Therefore, the node is not in X, and we can apply Guard-protect and Guard-acc access the node.

Finally, we note a minor change in Guard-acc: the accessed resource is wrapped in the *view-join* modality $\sqcup_V$. This is an artifact of the RMC version of *cancellable invariants* [22], which is a variant of standard invariant that allows *cancellation*, *i.e.*, taking the resource back from the invariant permanently so that they can be reclaimed (with free). Since this is mostly orthogonal to the contributions of this paper, we refer the reader to Dang [7], Dang et al. [8] for details.

3.2 Traversal Specification

The base specification can be unwieldy for more complex data structures such as Harris's lock-free linked list [18]. These data structures involve operations that traverse linked nodes, with nodes being added and removed at arbitrary positions within the structure, not solely at the head. This introduces two difficulties. First, even under SC, a traversal can encounter a node that is not expired but retired during the critical section. This complicates the application of Guard-protect, as Guard-not-retired is not applicable, unlike in Treiber's stack under SC. To prove the safety of traversals, we must reason about the shape of the data structure at the past point when the critical section began. Second, verification under RMC requires significant additional effort such as defining the detachment observation assertion. For linked lists, this involves considering all possible paths to the node, including the path formed due to stale values, details of which we discuss in §5. To mitigate these complexities, the high-level traversal specification encapsulates these common reasoning patterns into a set of local and inductive proof rules.

Background: traversal specification in SC. The key idea of Jung et al. [20] for reasoning about RCU-protected traversal is to formulate data structure invariants using a pair of logical points-to assertion and the corresponding *pointed-by* assertion [27, 31]. For example, Harris's lock-free linked list with nodes $L \in \mathsf{List}(\mathsf{Loc} \times \mathsf{AId})$ is described as follows:

$$\boxed{\circledast_{0 \le i < \mathsf{length}(L)} \begin{pmatrix} \textbf{let } (\ell, a) := L[i] \textbf{ in } \mathsf{BlockInfo}(\ell, a, \mathsf{ListBlock}) * \exists s.\, \mathsf{RcuPointsTo}(a, s) * \\ \textbf{if } i = \mathsf{length}(L) - 1 \textbf{ then } s = \mathsf{None} \\ \textbf{else let } a_{\mathrm{succ}} := L[i+1].2 \textbf{ in } s = \mathsf{Some}(a_{\mathrm{succ}}) * \mathsf{RcuPointedBy}(a_{\mathrm{succ}}, \{a\}) \end{pmatrix}}$$

The RCU points-to predicate, $\mathsf{RcuPointsTo}(a, s)$, asserts ownership of block a and indicates that a currently points to another block s, if any. Guard-protect-RcuPointsTo states that if a block is protected by the guard, its successor is also protected. This ensures that all blocks reachable by traversal are protected by the critical section.

Conversely, the RCU pointed-by predicate, $\mathsf{RcuPointedBy}(a, B)$, asserts that the block a is not yet retired and is currently pointed to by the blocks in B. This predicate prevents a block from pointing to a potentially retired block: RcuPointsTo-update updates the target of a block's points-to assertion only if the new target is not detached; and RcuPointedBy-detach grants permission to retire a block only when no other blocks point to it, effectively recording its detachment from the

$\mathsf{LinkObs} \triangleq (\mathsf{AId} \times \mathbb{N}) \to \mathit{vProp}$ $\quad \mathsf{Guard}(\mathit{tid} : \mathsf{TId},\ \mathit{LV} : \mathsf{LinkView},\ G : \mathsf{Loc} \xrightarrow{\mathsf{fin}} \mathsf{AId}) : \mathit{vProp}$

$\mathsf{LinkView} \triangleq \mathsf{AId} \xrightarrow{\mathsf{fin}} \mathbb{N}$ $\quad \mathsf{BlockInfo}(\ell : \mathsf{Loc},\ a : \mathsf{AId},\ P : \mathsf{BlockRes},\ \mathit{LO} : \mathsf{LinkObs}) : \mathit{vProp}$

$\mathsf{RcuPointsTo}(a : \mathsf{AId},\ \cancel{s : \mathsf{Option}(\mathsf{AId})}\ \vec{s} : \mathsf{List}(\mathsf{Option}(\mathsf{AId}))) : \mathit{vProp}$

$\mathsf{RcuPointsToSnap}(a : \mathsf{AId},\ n : \mathbb{N},\ s : \mathsf{Option}(\mathsf{AId})) : \mathit{vProp}$

$\mathsf{RcuPointedBy}(a : \mathsf{AId},\ B : \mathsf{Set}(\mathsf{AId} \times \mathbb{N})) : \mathit{vProp}$

(Guard-protect-RcuPointsTo)

$$\frac{G(\ell_1) = a_1 \qquad \vec{s}[n] = \mathsf{Some}(a_2) \qquad \mathit{LV}(a_1) \le n}{\mathsf{BlockInfo}(\ell_2, a_2, \_, \_) \vdash \mathsf{Guard}(\mathit{tid}, \mathit{LV}, G) * \mathsf{RcuPointsTo}(a_1, \cancel{\mathsf{Some}(a_2)}\ \vec{s}) \Rrightarrow^{*} \mathsf{Guard}(\mathit{tid}, \mathit{LV}, G[\ell_2 \mapsto a_2]) * \mathsf{RcuPointsTo}(a_1, \cancel{\mathsf{Some}(a_2)}\ \vec{s})}$$

(RcuPointsTo-objective)
$\mathsf{objective}(\mathsf{RcuPointsTo}(a, \vec{s}))$

(Guard-seen-linked)

$$\frac{\mathit{LV}(a) = n}{\mathsf{BlockInfo}(\_, a, \_, \mathit{LO}) * \mathsf{Guard}(\_, \mathit{LV}, \_) \vdash \mathit{LO}(a, n)}$$

(RcuPointsTo-update)

$$\frac{n = \mathsf{length}(\vec{s}) - 1 \qquad \vec{s}[n] = \mathsf{Some}(a_2)}{\mathsf{BlockInfo}(\_, a, \_, \mathit{LO}) * \mathit{LO}(a_1, n+1) \vdash \begin{pmatrix} \mathsf{RcuPointsTo}(a_1, \cancel{\mathsf{Some}(a_2)}\ \vec{s}) \\ * \mathsf{RcuPointedBy}(a_2, B_2) \\ * \mathsf{RcuPointedBy}(a_3, B_3) \end{pmatrix} \Rrightarrow^{*} \begin{pmatrix} \mathsf{RcuPointsTo}(a_1, \cancel{\mathsf{Some}(a_3)}\ \vec{s} +\!\!+ [\mathsf{Some}(a_3)]) \\ * \mathsf{RcuPointedBy}(a_2, B_2 \setminus \{(a_1, n)\}) \\ * \mathsf{RcuPointedBy}(a_3, B_3 \cup \{(a_1, n+1)\}) \end{pmatrix}}$$

(RcuPointsTo-link)

$$\begin{pmatrix} \mathsf{RcuPointsTo}(a_1, \vec{s}) * \\ \mathsf{RcuPointedBy}(a_2, B_2) \end{pmatrix} \Rrightarrow^{*} \begin{pmatrix} \mathsf{RcuPointsTo}(a_1, \vec{s} +\!\!+ [\mathsf{Some}(a_2)]) * \\ \mathsf{RcuPointedBy}(a_2, B_2 \cup \{(a_1, \mathsf{length}(\vec{s}))\}) \end{pmatrix}$$

(RcuPointedBy-overwritten)

$$\frac{n < n'}{\mathsf{BlockInfo}(\ell, a_1, \_, \mathit{LO}) * \mathit{LO}(a_1, n') \vdash \mathsf{RcuPointedBy}(a_2, B_2) \Rrightarrow^{*} \mathsf{RcuPointedBy}(a_2, B_2 \setminus \{(a_1, n)\})}$$

(RcuPointedBy-detach)
$\mathsf{BlockInfo}(\ell, a, \_, \_) \vdash \mathsf{RcuPointedBy}(a, \emptyset) \Rrightarrow^{*} \mathsf{RetirePerm}(\ell, a)$

(rcu-retire-traversal)
$\{\mathsf{RetirePerm}(\ell, \_)\}\ \mathsf{retire}(\ell)\ \{\mathsf{True}\}$

(RcuPointedBy-clean)

$$\begin{pmatrix} \mathsf{RcuPointedBy}(a_1, B) \\ * \mathsf{RetirePerm}(\_, a_2) \end{pmatrix} \Rrightarrow^{*} \begin{pmatrix} \mathsf{RcuPointedBy}(a_1, B \setminus \{(a_2, n)\}) \\ * \mathsf{RetirePerm}(\_, a_2) \end{pmatrix}$$

(Guard-protect-RcuPointsToSnap)

$$\frac{G(\ell_1) = a_1 \qquad \mathit{LV}(a_1) \le n}{\mathsf{BlockInfo}(\ell_2, a_2, \_, \_) \vdash \mathsf{Guard}(\mathit{tid}, \mathit{LV}, G) * \mathsf{RcuPointsToSnap}(a_1, n, s) \Rrightarrow^{*} \mathsf{Guard}(\mathit{tid}, \mathit{LV}, G[\ell_2 \mapsto a_2]) * \mathsf{RcuPointsToSnap}(a_1, n, s)}$$

Fig. 4. The traversal specification of RCU.

data structure. This, along with RCU-RETIRE-TRAVERSAL, guarantees that only detached nodes can be retired. RCUPOINTEDBY-CLEAN removes a detached node from RCU pointed-by assertion, which is used when detaching a chain of nodes at once.

Adaptation for RMC. As shown in the stack example, a stale pointer loaded in an RCU critical section is not expired and thus is safe to access. Therefore, we modify GUARD-PROTECT-RCUPOINTSTO to extend protection to stale reads. To achieve this, mirroring the physical points-to assertion, the RCU points-to assertion tracks the *successor history* $\vec{s}$, which is a list of the allocation IDs of the successors to which the current node has pointed.

However, overly stale pointers that cannot be read due to RCU synchronization should not be protected. To express this, we introduce the *link view*, LV, in the Guard assertion. Inspired by the concept of thread views for RMC, a guard's link view represents its observation of link modifications made visible via RCU synchronization. Specifically, a link view is a partial function mapping allocation IDs of blocks to the index of their successor history. GUARD-PROTECT-RCUPOINTSTO grants protection to a successor only if it is not stale in the link view ($LV(a_1) \le n$).

To prove the assumption of GUARD-PROTECT-RCUPOINTSTO, the user must show that the physical address loaded from the atomic points-to assertion corresponds to a non-stale index in the link view. To do this, we give the link view a physical interpretation, called the *link observation predicate* of type LinkObs. The BlockInfo predicate is parameterized by link observation predicate LO. $LO(a, n)$ should assert the physical observation of the message for the n-th successor of a, *i.e.*, that successors up to index $n-1$ have been overwritten. Before loading a pointer, we use GUARD-SEEN-LINKED to obtain this observation, which is then provided to the atomic points-to load rule. Its postcondition establishes the assumption of GUARD-PROTECT-RCUPOINTSTO, protecting the loaded pointer.

RCUPOINTSTO-UPDATE is updated accordingly. RcuPointedBy is modified to track incoming links not yet observed to be overwritten. To update, the user must provide an observation of the new link overwriting the old one. This difference from SC is better explained with the rules split into two parts: RCUPOINTSTO-LINK only adds link without requiring link observation; and RCUPOINTEDBY-OVERWRITTEN deletes the incoming link, given the overwrite observation. To identify each incoming link, the type of B is changed to a set of pairs, each containing the allocation ID of the predecessor and the link's index in the predecessor's history. RcuPointedBy keeps the provided observations, making it non-objective (while RcuPointsTo is objective). This ensures that a block is retired only after it is observed to be detached from the data structure.

The invariant for the lock-free linked list is updated as follows:

$$\ast_{0 \le i < \mathsf{length}(L)} \left(\begin{array}{l} \textbf{let } (\ell, a) := L[i] \textbf{ in } \exists \vec{s}.\, \mathsf{RcuPointsTo}(a, \vec{s}) \ast \\ \textbf{if } i = \mathsf{length}(L) - 1 \textbf{ then } \mathsf{last}(\vec{s}) = \mathsf{None} \\ \textbf{else let } a_{\mathrm{succ}} := L[i+1].2 \textbf{ in } \mathsf{last}(\vec{s}) = \mathsf{Some}(a_{\mathrm{succ}}) \ast \\ \quad @_{(\text{view of } a \to a_{\mathrm{succ}})} \mathsf{RcuPointedBy}(a_{\mathrm{succ}}, \{(a, \mathsf{length}(\vec{s}) - 1)\}) \end{array} \right)$$
$$\ast \ \ast_{(a,n,a') \in E} \ (\mathsf{RcuPointsToSnap}(a, n, \mathsf{Some}(a')) \ast \ldots)$$

In the first conjunct, L is the list of nodes along the path consisting only of the latest messages in each location (corresponding to L in SC). RcuPointsTo and RcuPointedBy are asserted along L, with RcuPointedBy asserted at the message view of the incoming link. A thread modifying the link can strip the view-at modality of RcuPointedBy by acquire-loading the link. To account for traversing the stale nodes not in L, the second conjunct asserts knowledge about E, the set of all links that have ever been written. Specifically, $\mathsf{RcuPointsToSnap}(a, n, \mathsf{Some}(a'))$ is a snapshot of RcuPointsTo,

Invariant: $\exists f_0, f_1, A_0, A_1.\ \mathsf{flags}[0] \overset{1/2}{\longmapsto} f_0 * \mathsf{flags}[1] \overset{1/2}{\longmapsto} f_1 * \gamma_0 \overset{1/2}{\Longmapsto} A_0 * \gamma_1 \overset{1/2}{\Longmapsto} A_1 * \neg(A_0 \wedge A_1) * (\neg(A_0 \vee A_1) \Rightarrow P) * (A_0 \Rightarrow f_0) * (A_1 \Rightarrow f_1)$

Algorithm 3 The flag mutex

```
1:  global variables
2:      flags: [Atomic<bool>; 2]
3:  function try_acquire(i: 0|1) → bool
4:      flags[i].store(true, rlx)
5:      fence(sc)
6:      if ¬ flags[1-i].load(acq) then
7:          return true
8:      flags[i].store(false, rlx)
9:      return false
10: function release(i)
11:     flags[i].store(false, rel)
```

$\{\mathsf{flags}[i] \overset{1/2}{\longmapsto} \mathsf{false} * \gamma_i \overset{1/2}{\Longmapsto} \mathsf{false}\}$
flags[i].store(true)
$\{\mathsf{flags}[i] \overset{1/2}{\longmapsto} \mathsf{true} * \gamma_i \overset{1/2}{\Longmapsto} \mathsf{false}\}$
if ¬ flags[1-i].load() **then**
  $\{\mathsf{flags}[i] \overset{1/2}{\longmapsto} \mathsf{true} * \gamma_i \overset{1/2}{\Longmapsto} \mathsf{true} * P\}$
  return true
flags[i].store(false)
$\{\mathsf{flags}[i] \overset{1/2}{\longmapsto} \mathsf{false} * \gamma_i \overset{1/2}{\Longmapsto} \mathsf{false}\}$
return false

Fig. 6. Code and proof sketch for the flag mutex algorithm.

recording that node a's successor history contains node a' at index n. RcuPointsToSnap is equipped with a variant of the traversal protection rule, Guard-protect-RcuPointsToSnap.

Verification of the Michael-Scott queue under RMC. As another example, let us consider Michael and Scott [39]'s lock-free queue. This queue consists of an append-only linked list and two fields containing pointers to the head and the tail node of the list, respectively. The enqueue method appends a node at the tail and advances the tail pointer, and the dequeue method advances the head pointer to the next node of the head node.

Similarly to the invariant for Harris's list, each list node in the queue maintains an RcuPointedBy assertion. However, RcuPointsTo for the list nodes are *not necessary*, because the queue methods do not traverse the list—they access the list nodes only via the head and the tail pointer. Consequently, the RcuPointedBy assertions only need to track the links from the head and the tail pointer. While tracking the links between list nodes is correct, omitting them simplifies the proof.

The tricky part is retiring the head node in the dequeue method. Since the node can also be pointed by the tail pointer, the dequeue method should ensure that the tail pointer does not lag behind the head pointer. Only after the dequeue method observes that both the head and the tail pointers have moved past the head node can we utilize RcuPointedBy-overwritten to remove the links from them in the RcuPointedBy of the head node. Then, we can use RcuPointedBy-detach and rcu-retire-traversal to prove the safety of retiring the node.

4 Reasoning about RAW Synchronization for Switchable Critical Sections

We develop a reasoning principle for read-after-write (RAW) synchronization in RMC, and apply it to the idealized *two-thread flag mutex* algorithm that captures the essence of RAW synchronization, Peterson [43]'s mutex, and epoch-based RCU with switchable critical sections.

4.1 Background: Reasoning about RAW Synchronization in SC

Flag mutex. Fig. 6 presents the flag mutex algorithm. The algorithm ensures mutual exclusion for two threads using symmetric boolean flags, each representing a thread's intention to acquire the mutex. The try_acquire() function takes an argument i that indicates the thread's index (0 or 1) and returns a boolean value indicating whether the acquisition was successful. The function first writes

true to the flag for i, announcing its intention to acquire the mutex (line 4). It then reads the flag for the other thread $1 - i$ (line 6). If this flag is false, indicating that the other thread has no intention to acquire the mutex, the acquisition is successful and true is returned (line 7). The release() function simply writes false to the flag for the thread, indicating that it has released the mutex (line 11).

In separation logic, mutual exclusion can be specified with the following Hoare triple:

$$\{\mathsf{True}\}\ \mathsf{try\_acquire}(i)\ \{v.\ (v = \mathsf{true} * P) \lor v = \mathsf{false}\}$$

This specification states that if the function returns true, the thread has acquired the protected resource P; otherwise, it returns false.

As a stepping stone for the proof of the flag mutex under RMC, we first review the proof under SC by Jung et al. [20]. Their invariant has four existential variables representing the physical and logical state: f_i for the current value stored in flags$[i]$, and A_i indicating whether the thread i has acquired the mutex. The A_i values are stored in *ghost variables* γ_i, a separation logic resource that acts like a points-to predicate for a logical variable. Both points-to and ghost variable assertions can be split into fractional parts and shared among threads, but require full ownership for updates. The invariant keeps the halves of the points-to assertions for flags$[i]$ and γ_i ghost variables, while the thread i owns the other halves. Thus, each thread can read and write the points-to predicate and ghost variable it owns, while it can only read from those of the other thread.

The invariant encodes the mutual exclusion with two conditions: $\neg(A_0 \land A_1)$: both threads cannot acquire the mutex simultaneously; and $\neg(A_0 \lor A_1) \Rightarrow P$: if neither thread has acquired the mutex, the invariant owns the protected resource P. Furthermore, $A_i \Rightarrow f_i$ describes the necessary condition for thread i to acquire the mutex: its flag must be true.

The proof proceeds as follows. Thread i begins with flags$[i] \overset{1/2}{\longmapsto}$ false $* \gamma_i \overset{1/2}{\Longmapsto}$ false, denoting that its flag is false and it has not acquired the mutex. When thread i stores true to its flag (line 4), its points-to predicate is updated to reflect this change. The ghost variable remains unchanged as it has not yet acquired the mutex. If it finds the other thread's flag to be false (line 6), then by the invariant $A_{1-i} \Rightarrow f_{1-i}$, A_{1-i} must also be false. Because A_i is also still false, the invariant $\neg(A_0 \lor A_1) \Rightarrow P$ implies that P is owned by the invariant. Therefore, thread i can set A_i to true, taking P out of the invariant, and return true (line 7).

Epoch-based RCU. Jung et al. [20] verified the epoch-based RCU (§2.1) by modeling the epoch invariant as a mutual exclusion over the *ownership of epoch* using RAW synchronization. Intuitively, the ownership of epoch e is the permission required for incrementing the global epoch from $e + 1$ to $e + 2$. The ownership of an epoch is divided into individual fractions for each thread. When a thread enters critical section with rcu_lock() at epoch e, it acquires its fraction of the ownership for e. Conversely, the invocation of try_advance() that increments the global epoch from $e + 1$ to $e + 2$ checks if each thread is in epoch e, and if not, claims the thread's fraction. If no thread was in epoch e, it has collected the full ownership of e, allowing it to increment the global epoch. Based on this protocol, they adapted the above proof strategy for the flag mutex to verify the epoch-based RCU.

4.2 Proof Rules for SC Fence

RAW synchronization in RMC requires the use of SC fences. To reason about them, we first introduce two proof rules for SC fences: a low-level rule directly reflecting the semantics and a high-level rule that exploits the total order property of SC fences.

SC modality. The *SC modality* assertion $\langle \mathsf{sc} \rangle\ P$ means that P holds in a *snapshot* of the SC view. Recall that executing an SC fence updates both the SC view and the executing thread's view to their join. This allows us to place P under the SC modality and extract Q from $\langle \mathsf{sc} \rangle\ Q$ (Fence-SC). We primarily use the SC modality to capture observations (*i.e.*, persistent non-objective assertions) about the

Low-level rules

(Fence-SC)
$\{P * \langle \mathsf{sc} \rangle\, Q\}\ \mathsf{fence}(\mathsf{sc})\ \{\langle \mathsf{sc} \rangle\, P * Q\}$

(SC-mod-objective)
$\mathsf{objective}(\langle \mathsf{sc} \rangle\, P)$

Event history rules

$$p \in \mathsf{ScProtocol} \triangleq \langle S : \mathsf{JoinSemilattice},\ \mathsf{Seen} : S \to \mathit{vProp},\ \gamma : \mathsf{GhostName} \rangle$$

(Fence-SC-hist)
$$\left\{ \begin{array}{l} \mathsf{ScHist}_p(h) \\ * \mathsf{Seen}_p(s) \end{array} \right\} \mathsf{fence}(\mathsf{sc}) \left\{ \begin{array}{l} \mathbf{let}\ s' := s \sqcup \bigsqcup_{s' \in h} s'\ \mathbf{in} \\ \mathsf{ScHist}_p(h +\!\!+ [s']) * \mathsf{Seen}_p(s') \end{array} \right\}$$

(SC-state-snap)
$$\frac{h[n] = s}{\mathsf{ScHist}_p(h) \vdash \mathsf{ScState}_p(n, s)}$$

$$\mathsf{ScHist}_p(h : \mathsf{List}(S_p)) \triangleq \mathsf{ListAuth}(\gamma_p, h) * \mathsf{sorted}(h, \sqsubseteq_p) * \mathop{\Large *}_{s \in h} \langle \mathsf{sc} \rangle\, \mathsf{Seen}_p(s)$$

$$\mathsf{ScState}_p(n : \mathbb{N}, s : S_p) \triangleq \exists h'.\, \mathsf{ListSnap}(\gamma_p, h') * h'[n] = s$$

Fig. 7. Proof rules for SC fence

"write" component of RAW synchronization pattern. For example, $\langle \mathsf{sc} \rangle\, (\sqsupseteq V * V(m.\mathsf{loc}) \geq m.\mathsf{time})$ asserts that the observation of message m has been transfered to the SC view, which can be acquired by subsequent SC fences. Since the SC modality fixes the view (SC-mod-objective), this assertion can be placed in an objective invariant alongside other resources like atomic points-to.

Event history with SC fence. A key property of SC fences is their total ordering under the happens-before relation. This allows case analysis on fence ordering: an SC fence either happens before or after another. The earlier fence releases observations about prior writes into the SC view, which is acquired by a later fence that prevents the subsequent read operations from reading stale values.

To utilize this property, we provide a higher-level proof rule for SC fence. An *SC protocol* p consists of a state type S that is observed by each SC fence, a predicate Seen that asserts the observation of those states, and a ghost name γ for the protocol. Here, S is a join-semilattice that defines a custom abstraction for the notion of view in the RMC semantics. For example, in the flag mutex, S is the pair of timestamps of each flag location when each SC fence ran, and Seen is the assertion that the thread has observed messages in the location with the given timestamp. The *SC history* assertion $\mathsf{ScHist}_p(h)$ records the history of states in S observed by the SC fence. Fence-SC-hist appends an event to this history. It takes the observation of a state to be added, joins this state with all preceding states in the history, appends the joined state to the history, and returns its observation. As a result, the states tracked by ScHist monotonically increase. To enable reasoning about the state at a specific index in the history, SC-state-snap takes a persistent snapshot ScState of the state s at the index n.[7]

4.3 Reasoning about RAW Synchronization in RMC

Verifying the flag mutex in RMC. Reasoning about RAW synchronization in RMC effectively reduces to reasoning about the SC fence history. For the flag mutex, a thread that wants to acquire the mutex must satisfy two conditions on the SC fence history: **(1)** *announcement of intention*: the thread should announce its intention to acquire the mutex by transferring the observation of a message to the SC fence, and **(2)** *opponent's lack of intention*: it must have observed a moment where

[7]This is implemented with the authoritative PCM [24] of append-only lists.

$$\mathsf{flagp} \triangleq \langle S = \{0,1\} \xrightarrow{\text{fin}} \mathsf{Time},\ \mathsf{Seen}([t_0,t_1]) = \exists V.\ \sqsupseteq V * V(\mathsf{flag}[0]) \geq t_0 * V(\mathsf{flag}[1]) \geq t_1 \rangle$$

$$\text{Invariant: } \boxed{\exists F_0, F_1, h, A_0, A_1.\ \mathsf{flags}[0] \mapsto_{\mathsf{at}} F_0 * \mathsf{flags}[1] \mapsto_{\mathsf{at}} F_1 * \mathsf{ScHist}_{\mathsf{flagp}}(h) * \ldots * (\neg(A_0 \vee A_1) \Rightarrow \mathsf{@}_{\ldots} P) * (A_0 \Rightarrow \mathsf{acquirable}(0,h,F)) * (A_1 \Rightarrow \mathsf{acquirable}(1,h,F)) * \ldots}$$

$$\mathsf{acquirable}(i,h,F) \triangleq \exists n.\ F_i(h[n][i]).\mathsf{value} = 1 * h[n][i] = \max(\mathsf{dom}(F_i)) \quad \text{(announcement of intention)}$$
$$* \exists t' \geq h[n][1-i].\ F_{1-i}(t').\mathsf{value} = 0 \quad \text{(opponent's lack of intention)}$$

Fig. 8. Invariant for the flag mutex in RMC

the opponent did not intend to acquire the mutex. The proof of mutual exclusion then proceeds by comparing the SC fence indices of the two threads, exploiting the total order property of SC fences.

Fig. 8 formalizes this intuition, highlighting the changes from SC. In the invariant, the points-to assertions for flag locations are replaced with the atomic points-to assertions, recording their history ($F_i \in \mathsf{Time} \xrightarrow{\text{fin}} \mathsf{Msg}$). To reason about the observation of the flag locations transferred via the SC fence, the invariant uses an SC protocol where the state type is the pair of timestamps of each flag location when each SC fence executed. The protected resource P is kept under the message view of the last release (omitted in the figure).

The necessary condition for acquiring the mutex is captured by the $\mathsf{acquirable}(i,h,F)$ assertion. It asserts the two conditions for the thread i to acquire the mutex: announcement of intention and the opponent's lack of intention. The announcement condition states that i must have written 1 to its flag and executed an SC fence. The fence is identified by the index n, and the n-th state in the fence history contains the latest message in $\mathsf{flag}[i]$, which has value 1. This roughly corresponds to the $A_i \Rightarrow f_i$ condition in the SC proof.

The opponent condition is unique to RMC. It states that the opponent (thread $1-i$) must have *withdrawn* its intention to acquire mutex that has been *acknowledged* by i. Given an SC fence history h, suppose $1-i$ announced an intention at an index n'. We say that this intention is acknowledged by i if i's fence index n is higher than n'. By the monotonicity of h, this implies $h[n'][1-i] \leq h[n][1-i]$. This ensures that i cannot read a message older than the intention by $1-i$ thanks to the SC fence synchronization. As $1-i$ withdraws its intention by writing 0 to its flag, the opponent condition says that $1-i$ must have written a message with value 0 at a timestamp t' that is higher than the one i acknowledged ($t' > h[n][1-i]$). Note that this condition does not require that the opponent does not have intention at the current moment; it merely says that it has once withdrawn the intention at some moment after the execution of the SC fence n.

For brevity, we briefly describe the proof sketch at the point where thread i reads 0 from $\mathsf{flag}[1-i]$ (line 6). We show that thread i can take ownership of resource P, *i.e.*, thread $1-i$ has not taken it. Towards a contradiction, assume otherwise. From $\mathsf{acquirable}(1-i,h,F)$, we get the fence index n' of thread $1-i$. Let n be the index of the fence that thread i executed at line 5. We proceed by comparing n' and n. **(1)** $n < n'$ (i was earlier): $h[n][i]$ is the latest message in $\mathsf{flag}[i]$. By $h[n] \sqsubseteq_{\mathsf{flagp}} h[n']$, we have $h[n][i] = h[n'][i]$. This derives a contradiction from the opponent condition of $1-i$ and the announcement condition of i. **(2)** $n' < n$ ($1-i$ was earlier): By the announcement condition of $1-i$, $h[n][1-i]$ contains the latest message in $\mathsf{flag}[1-i]$, which has value 1. Since $h[n'] \sqsubseteq_{\mathsf{flagp}} h[n]$, i must have observed that message. This contradicts the fact that i read 0 at line 6.

Verifying Peterson's mutex. The flag mutex suffers from starvation, as both thread may fail to acquire the mutex when they are trying to acquire simultaneously. Peterson's mutex [43] (Algorithm 4) solves this problem by introducing an additional shared variable, yield. To acquire the mutex, thread i first sets its flag and then writes i to yield (line 3), effectively yielding priority to the other

Algorithm 4 Peterson's mutex

```
1:  function acquire(i: 0|1)
2:      flags[i].store(true, rlx)
3:      fence(sc); yield.store(i, rlx); fence(sc)
4:      loop
5:          if ¬ flags[1-i].load(acq) then
6:              return
7:          if yield.load(acq) == 1 - i then
8:              return
9:  function release(i)
10:     flags[i].store(false, rel)
```

$\text{petersonp} \triangleq \langle S = \{0, 1, \boxed{\text{yield}}\} \xrightarrow{\text{fin}} \text{Time}, \ldots \rangle$

Invariant: $\boxed{Y, \ldots . \text{yield} \mapsto_{\text{at}} Y} *$

$\text{acquirable}(i, h, F, \boxed{Y}) \triangleq \exists n_1, n_2, t_y.\ n_1 < n_2 *$

$F_i(h[\boxed{n_1}][i]).\text{value} = 1 * h[\boxed{n_1}][i] = \max(\text{dom}(F_i)) *$ (A1)

$\boxed{Y(t_y).\text{value} = i * h[n_1][\text{yield}] < t_y \leq h[n_2][\text{yield}]} *$ (A2)

$\begin{pmatrix} (\exists t' \geq h[\boxed{n_2}][1-i].\ F_{1-i}(t').\text{value} = 0) \vee \\ (\boxed{\exists t'_y \geq h[n_2][\text{yield}].\ Y(t'_y).\text{value} = 1 - i}) \end{pmatrix}$ (O1) (O2)

Fig. 9. Invariant for Peterson's mutex

thread. It then reads the flag of thread $1 - i$ (line 5). If the flag is set, it checks whether thread $1 - i$ has yielded back by writing $1 - i$ to yield (line 7). If so, thread i enters the critical section. The writes to yield must be wrapped in SC fences to enforce happens-before ordering along the timestamp order of those writes. Intuitively, this is necessary because the order of writes on yield decides which thread acquires the mutex first when threads are contending.

The invariant for Peterson's mutex is shown in Fig. 9, with the additions to the flag mutex invariant highlighted in green. The SC protocol and the invariant are updated to track the yield variable. The acquirable condition inherits the announcement of intention (A1) and the opponent's lack of intention condition (O1), but A1 uses n_1, the index of the first fence by i, and O1 uses n_2, the index of the second fence by i. The announcement condition is extended with an assertion that thread i wrote to yield a message with value i and timestamp t_y between the two SC fences (A2). To accommodate the scenario where acquire(i) returns at line 8, the opponent condition is changed to a disjunction with a branch that asserts thread i could have read the yield message by $1 - i$ after it executes the second fence (O2).

The proof of mutual exclusion proceeds similarly to the flag mutex, deriving a contradiction from the assumption that $1 - i$ has already acquired the mutex. However, we need to consider more cases: the order of four fences, the return point of acquire(i), and $1 - i$'s opponent condition (O1 or O2). While there numerous cases to examine, the proof boils down to showing that the thread who wrote the more recent message to yield cannot acquire the mutex. Without loss of generality, let this thread be i. **(1)** i cannot read false at line 5. **(1-1)** The second fence of i cannot precede the first fence of $1 - i$, because otherwise, $1 - i$ could not have written the older message to yield (violation of A1 of i and $1 - i$). **(1-2)** So, one of $1 - i$'s fences precedes one of i's fences. However, this prevents i from reading false from flag$[1 - i]$, for the same reason as in the flag mutex. **(2)** i cannot read $1 - i$ at line 7, because, by assumption, the message written by $1 - i$ is older than the message written by i, and RMC prohibits a thread from reading a message older than what it wrote.

Verifying epoch-based RCU. For brevity, we focus on the SC protocol between try_advance and rcu_lock. The SC state consists of the timestamps of global epoch and local epoch locations: $\text{Time} \times (\text{Tld} \xrightarrow{\text{fin}} \text{Time})$. Similar to the flag mutex, the invariant specifies announcement and opponent conditions for acquiring the ownership of epoch e, but they are asymmetric: one for incrementing the global epoch from $e + 1$ to $e + 2$, and another for entering a critical section with local epoch e. **(1)** For incrementing the global epoch, the announcement condition is the observation of the message that writes $e + 1$ to the global epoch, and the opponent condition is that each local epoch location has been set to $e + 1$ or -1, which allows the incrementing the global epoch to $e + 2$ according to the epoch invariant. **(2)** For entering a critical section, the announcement condition is

writing local epoch e, and the opponent condition is that it read global epoch e, meaning that at the moment of the fence, no one has announced the intention to increment the global epoch to $e + 2$.

The proof that try_advance() successfully collects full ownership of epoch e before incrementing the global epoch to $e + 2$ proceeds by contradiction. Suppose a thread is in the critical section with epoch e. By its opponent condition, this thread's fence event must have observed the global epoch e. Since the incrementing thread's fence contains a larger global epoch $e + 1$, its fence event must have occurred later (as global epoch only increases). But this makes the incrementing thread read the local epoch e, contradicting the opponent condition for incrementing the global epoch.

5 Reasoning about Reachability for Concurrent Writers

We prove the traversal specification of epoch-based RCU: any memory block reachable by traversal in a critical section is safe to access (§3.2). We first present a proof sketch for the case where writes are totally ordered (§5.1). This proof is adapted from Tassarotti et al. [46]'s verification of a single-writer RCU-protected linked list, for presentation purposes. We then generalize this proof to handle concurrent writers in RMC, where writes are only partially ordered (§5.2).

5.1 Background: RCU-Protected Traversal for Totally Ordered Writes

The key idea behind reasoning about RCU's traversal protection is to track the *history of link events* and maintain invariants on them. The link history contains two kinds of events:

- $\mathsf{link}(a_1, \mathsf{Some}(a_2) \mid \mathsf{None})$: making a link from a memory block with allocation ID a_1 to another block with ID a_2 or to null; and
- $\mathsf{remove}(a)$: declaring that the block a is removed from the data structure, *i.e.*, no longer *live*.

When writes are totally ordered, the history can be represented as a *list* of events. Data structures must then maintain two invariants on this totally ordered history:

live-closed If the most recent link from a live block a_1 points to a_2, then a_2 is also live.
no-link-to-removed If a is removed, then no later event creates a link to a.[8]

Intuitively, these invariants require that a block should be made unreachable before it is removed and remain unreachable. This ensures that any node reachable by traversal must have been live at the beginning of the traversal, and thus safe to access thanks to the base specification of RCU (§3.1).

We briefly overview the proof of RCU traversal specification under this assumption. The points-to and pointed-by predicates of the RCU traversal specification track fragments of the link history: $\mathsf{RcuPointsTo}(a_1, \vec{s})$ asserts that the sub-history of a_1's outgoing links is $\vec{s}$, while $\mathsf{RcuPointedBy}(a, B)$ asserts that B is the set of blocks pointing to a in the latest link state. The internal invariant of the traversal specification asserts that all retired blocks have a remove event in the current link history. The Guard assertion takes a *snapshot* of the history upon entering the critical section, recording the blocks removed so far. Guard asserts the observation of *all* link events in its history snapshot, which is reflected in the link view LV. Furthermore, Guard strengthens the base specification's Guard to protect only the blocks that are not removed in the history snapshot.

The proof of Guard-protect-RcuPointsTo is done by a case analysis on whether the link from a_1 to a_2 was made before or after the start of the critical section. (1) If the link was earlier, then a_1 must have been pointing to a_2 in the guard's history snapshot. By live-closed, a_2 is not removed in the guard's history snapshot. (2) Otherwise, by no-link-to-removed, a_2 must have not been removed at the time that the link was made, so it is not removed in the guard's history snapshot.

[8]This invariant is a relaxed variant by Jung et al. [20] of the original version by Tassarotti et al. [46], which asserts that there should be no new link *from* removed nodes.

$$\mathsf{RcuPointedBy}(a, B) \triangleq \exists L, D, LV.\ \mathsf{SetAuth}(a, L) * \mathsf{SeenRemoved}(D, LV) * \circledast_{(a',n)\in L} \begin{pmatrix} (\mathsf{RcuPointsTo}(a', \_) \text{ has link}(a', n, a) \text{ event}) * \\ ((a', n) \notin B \to \mathsf{dead\_in}(a', n, D, LV)) \end{pmatrix}$$

$$\mathsf{dead\_in}(a, n, D, LV) \triangleq a \in D \vee LV(a) > n$$

$$\mathsf{SeenRemoved}(D, LV) \triangleq (\text{physical observation of } LV) * \circledast_{a\in D} \left(\exists L.\ \mathsf{SetFrozen}(a, L) * \bigwedge_{(a',n)\in L} \mathsf{dead\_in}(a', n, D, LV) \right)$$

$$\mathsf{Guard}(tid, LV, G) \triangleq \exists X, D.\ \mathsf{BaseGuard}(tid, X, G) * \mathsf{SeenRemoved}(D, LV) * X \subseteq D * \ldots$$

$$\mathsf{RetirePerm}(\ell, a) \triangleq \mathsf{BaseRetirePerm}(\ell, a) * \exists D, LV.\ \mathsf{SeenRemoved}(D, LV) * a \in D$$

$$\text{invariant: } \boxed{\exists R.\ \mathsf{RcuState}(R) * \circledast_{a\in R} \mathsf{Retired}(a, (\exists D, LV.\ \mathsf{SeenRemoved}(D, LV) * a \in D))}$$

Fig. 10. Definitions of RCU traversal specification predicates for partially ordered history

5.2 Link Invariants for Partially Ordered Writes

The proof from §5.1 assumes a totally ordered link history and thus does not directly apply to the general case, where writes are only partially ordered due to unsynchronized concurrent writes in RMC. To prove traversal safety under partially ordered writes, we formulate the link history invariants directly in terms of the *set of all incoming edges to each node* as follows:

live-closed For a node to be removed, all its incoming edges must be *dead*, *i.e.*, either the edge is overwritten by a later link on the predecessor node, or the predecessor node itself is removed.

no-link-to-removed Once a node is removed, its incoming edge set remains unchanged.

Fig. 10 formalizes this idea. $\mathsf{RcuPointedBy}(a, B)$ maintains the set L of all incoming edges to a that have ever existed ($\mathsf{SetAuth}(L)$). To distinguish edges from the same node, each edge is identified by its source node and the index of the edge from that source. For each incoming edge, RcuPointedBy asserts that the corresponding source node's RcuPointsTo has the corresponding link event (omitted). Importantly, if an edge (a', n) is not in B (the set of *current* incoming edges), it must be dead (dead_in): either the source node was removed ($a' \in D$) or a new link overwrote the edge ($LV(a') > n$). RcuPointedBy tracks the observation of the death of edges using $\mathsf{SeenRemoved}(D, LV)$, which represents the observation of the link view LV and the fact the D is a closed set of removed nodes. The incoming edge set of each node in D is frozen (SetFrozen),[9] corresponding to no-link-to-removed. Furthermore, all edges to nodes in D are dead, corresponding to live-closed. Finally, Guard asserts SeenRemoved with D including the expired nodes.

In this setting, the proof of Guard-protect-RcuPointsTo follows directly from SeenRemoved. By the rule's assumptions, we have $\mathsf{SeenRemoved}(D, LV) * a_1 \notin D$. If we assume $a_2 \in D$, we get $\mathsf{dead\_in}(a_1, n, D, LV)$, which contradicts the assumptions that $a_1 \notin D$ and $LV(a_1) \leq n$.

6 Related and Future Work

Pointed-by assertions. Madiot and Pottier [31] designed a separation logic for reasoning about memory usage bound in a language with garbage collector. Specifically, they reason about *logically deallocated* memory blocks, *i.e.*, blocks that are unreachable and thus can be reclaimed by GC.

[9] SetAuth and SetFrozen assertions are based on the authoritative PCM [24] of sets with discarded fraction [51].

To this end, the logic uses *pointed-by* assertion of form $\ell \hookleftarrow L$, which tracks the multiset L of immediate predecessor blocks of block ℓ. The design of the high-level traversal specification for RCU (§3.2) adapts this interface to reason about detached blocks. The notable difference is that our logic tracks the history of links, while their logic only tracks the current state of links. This is necessary to support Guard-protect-RcuPointsTo, which talks about the link status at some moment in the past when the critical section started.

Reasoning about RAW synchronization in RMC. Vafeiadis [50] noted that the semantics of an SC fence is similar to a successful acqrel CAS operation to a dedicated ghost location (plus acqrel fence), and proposed the following proof rule for SC fences (formalized as sc_inv in iRC11):

$$\frac{J * P \Rrightarrow J * Q}{J \vdash \{P\}\,\mathsf{fence}(\mathsf{sc})\,\{Q\}}\ (\textsc{Fence-SC-Inv})$$

However, this rule is difficult to apply to complex algorithms. It follows the style of *single-location invariants* from earlier RMC separation logics such as FSL [12] and GPS [49], which requires relating multiple per-location invariants with extra ghost states. This significantly alters the proof structure compared to SC [7, §11], where a single invariant can encompass all resources within a module. While this issue was addressed in the recent version of iRC11 [7, 9] with general invariants, atomic points-to, and explicit view reasoning, Fence-SC-Inv still suffers from this problem. It utilizes a dedicated SC fence invariant accessible only when executing an SC fence, necessitating its relation to other invariants. To address this, we introduce new proof rules that subsume Fence-SC-Inv and is more amenable to complex algorithms (§4.2), aligning with the spirit of iRC11's approach.

Mével et al. [35] verified Peterson [43]'s mutex algorithm in the OCaml memory model. While this model is more relaxed than SC for *non-atomic* locations, its semantics for *atomic* locations (used for synchronization) are largely the same as in SC, with a total order on writes to all atomic locations. Thus, reasoning about RAW synchronization in this model is similar to SC. However, this model incurs extra overhead in the compilation scheme. In contrast, we verified the same algorithm in the RC11 RMC model [30], with more complex and yet more efficient synchronization patterns.

Alglave and Cousot [1] verified two mutex algorithms based on RAW synchronization in RMC. Their verification consists of two steps: **(1)** determining the necessary synchronization for custom invariants (*e.g.*, mutual exclusion) of concurrent algorithms in RMC, and **(2)** inserting synchronization primitives, such as fences, based on the analysis. However, this approach does not support switchable critical sections. Specifically, their Peterson's mutex algorithm cannot be reused after release, and their PostgreSQL mutex algorithm requires strict alternation between two threads, incurring deadlock when a thread does not intend to acquire the mutex. (Such an algorithm does not actually require RAW synchronization.) In contrast, we verified Peterson's mutex and epoch-based RCU algorithms that fully support switchable critical sections.

Dalvandi et al. [6] also verified a variant of Peterson's mutex algorithm in RMC, but their algorithm does not support mutex reuse and relies on strongly synchronizing AWAR operations on the turn variable, which indicates thread priority for critical section access.

Verifying concurrent reclamation in SC. Several works verified concurrent reclamation algorithms in SC [15, 17, 20, 41, 47]. However, none of these are not readily applicable to verification in RMC due to the challenges addressed in this paper. For a detailed comparison of these works, we refer the reader to Jung et al. [20, §8].

Among these, Gotsman et al. [17]'s work warrants further discussion. Their verification method uses temporal logic to specify the necessary condition of acquiring protection. This condition is expressed in the form of "X since Y", meaning that there was a moment when both X and Y

held, and X has remained true since then. At a high level, this condition resembles our reasoning principle for RAW synchronization using the SC fence history (§4.3). Here, X corresponds to the announcement of intention, and Y to the opponents' lack of intention. While temporal reasoning is not strictly necessary for verifying RAW synchronization in SC, we observe that it is crucial in RMC because the possibility of stale reads necessitates reasoning about the history of events.

Verifying concurrent reclamation in RMC. Several works verified concurrent reclamation algorithms in RMC. Tassarotti et al. [46]'s verification did not support modular specification, switchable critical sections, and concurrent writers (§1). We discuss other works below.

Alglave et al. [2] presented an RCU specification as an extension of the Linux Kernel's RMC model. They verified an implementation [10] against this specification with pen and paper, and verified small test programs using RCU with model checking. However, they did not provide reasoning principles for concurrent data structures using RCU or verify their correctness. In contrast, we provide a high-level RCU specification and modularly verify highly concurrent data structures. Moreover, their verification was conducted directly at the memory model, while ours builds upon a foundation of fully mechanized iRC11 separation logic. In addition, we identify a reusable reasoning principle for RAW synchronization in RMC separation logic.

Semenyuk et al. [44] verified a counter object protected with a simple boolean-based RCU in RMC. However, their RCU algorithm does not provide synchronization, delegating that responsibility to the client that relies on AWAR operations. This reliance on AWAR operations negates the need for RAW synchronization. Additionally, it is unclear how to apply their approach to more complex data structures that synchronize on multiple locations, such as linked lists.

Dang et al. [8], Doko and Vafeiadis [13], Park et al. [40] verified the atomic reference counter (ARC) in RMC. ARC uses AWAR synchronization on a centralized counter, while RCU uses decentralized RAW synchronization on multiple locations (*e.g.*, epochs) leading to more complex reasoning.

Gammie et al. [16] verified a concurrent tracing garbage collector (GC) in the x86-TSO memory model [45]. However, their verification depends on the total order of writes enforced by AWAR operations of x86-TSO. We believe our work provides building blocks for two key challenges for verifying GC in more relaxed settings: reasoning about RAW synchronization, a prevalent technique in GC algorithms, and reachability, a central concept in GC correctness (e.g., the tricolor invariant).

Future work. We plan to apply our verification method to other reclamation algorithms. Specifically, we are interested in the wait-free implementations of `rcu_lock()` employed in libraries such as Crossbeam [11], HP-BRCU [28], and the user-space RCU [10] verified by Alglave et al. [2]. For wait-freedom, these algorithms do not validate epoch to take ownership; instead, they essentially takes the ownership of *each individual memory block* upon each load operation. We believe this behavior can be formalized within our framework, and subsquently applied to the hazard pointers [36, 38] algorithm, which protects individual memory block by writing the pointer value to a shared variable and validates by checking that the object is still reachable.

Acknowledgments

This work is supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT2201-06.

References

[1] Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 3–18. doi:10.1145/3009837.3009883

[2] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 405–418. doi:10.1145/3173162.3177156

[3] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 2011. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 487–498. doi:10.1145/1926385.1926442

[4] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. 2001. Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 92–103. doi:10.1145/378795.378819

[5] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP (LNCS, Vol. 8586)*. 207–231. doi:10.1007/978-3-662-44202-9_9

[6] Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. 2020. Owicki-Gries Reasoning for C11 RAR. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:26. doi:10.4230/LIPIcs.ECOOP.2020.11

[7] Hoang Hai Dang. 2024. Scaling up relaxed memory verification with separation logics. doi:10.22028/D291-43142

[8] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt Meets Relaxed Memory. *PACMPL* 4, POPL, Article 34 (2020). doi:10.1145/3371102

[9] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic. In *PLDI*. 792–808. doi:10.1145/3519939.3523451

[10] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2012), 375–382. doi:10.1109/TPDS.2011.159

[11] Crossbeam Developers. 2023. Crossbeam. https://github.com/crossbeam-rs/crossbeam

[12] Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *VMCAI (LNCS, Vol. 9583)*. 413–430. doi:10.1007/978-3-662-49122-5_20

[13] Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *ESOP (LNCS)*. 448–475. doi:10.1007/978-3-662-54434-1_17

[14] Keir Fraser. 2004. *Practical lock-freedom*. Ph. D. Dissertation.

[15] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

[16] Peter Gammie, Antony L. Hosking, and Kai Engelhardt. 2015. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 99–109. doi:10.1145/2737924.2738006

[17] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 249–269. doi:10.1007/978-3-642-37036-6_15

[18] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[19] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[20] Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 251 (oct 2023), 29 pages. doi:10.1145/3622827

[21] Jaehwang Jung, Sunho Park, Janggun Lee, Jeho Yeon, and Jeehoon Kang. 2025. Artifact for "Verifying General-Purpose RCU for Reclamation in Relaxed Memory Separation Logic", PLDI 2025. doi:10.5281/zenodo.15167032

[22] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. doi:10.1017/S0956796818000151

[23] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *PACMPL* 4, POPL, Article 45 (2020). doi:10.1145/3371113

[24] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650. doi:10.1145/2775051.2676980

[25] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*. 17:1–17:29. doi:10.4230/LIPIcs.ECOOP.2017.17

[26] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *POPL*. 175–189. doi:10.1145/3093333.3009850

[27] Ioannis T. Kassios and Eleftherios Kritikos. 2013. A Discipline for Program Verification Based on Backpointers and Its Use in Observational Disjointness. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–168.

[28] Jeonghyeon Kim, Jaehwang Jung, and Jeehoon Kang. 2024. Expediting Hazard Pointers with Bounded RCU Critical Sections. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3626183.3659941

[29] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. doi:10.1145/3009837.3009855

[30] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI*. 618–632. doi:10.1145/3062341.3062352

[31] Jean-Marie Madiot and François Pottier. 2022. A Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 6, POPL, Article 11 (jan 2022), 28 pages. doi:10.1145/3498672

[32] Paul E. McKenney. 2004. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels.* Ph. D. Dissertation. OGI School of Science and Engineering at Oregon Health and Sciences University.

[33] P. E. McKenney and J. D. Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *PDCS '98*.

[34] Paul E. McKenney, Michael Wong, Maged M. Michael, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kamiński, and Jens Maurer. 2023. P2545R4: Read-Copy Update (RCU). https://wg21.link/p2545r4.

[35] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *PACMPL* 4, ICFP, Article 96 (2020). doi:10.1145/3408978

[36] Maged Michael, Maged M. Michael, Michael Wong, Paul McKenney, Andrew Hunter, Daisy S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, and Mathias Stearn. 2023. P2530R3: Hazard Pointers for C++26. https://wg21.link/p2530r3.

[37] Maged M. Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*. 73–82. doi:10.1145/564870.564881

[38] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (2004), 491–504. doi:10.1109/TPDS.2004.8

[39] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. 267–275. doi:10.1145/248052.248106

[40] Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. 2024. A Proof Recipe for Linearizability in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 8, PLDI, Article 154 (jun 2024), 24 pages. doi:10.1145/3656384

[41] Matthew Parkinson, Richard Bornat, and Peter O'Hearn. 2007. Modular Verification of a Non-Blocking Stack. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 297–302. doi:10.1145/1190216.1190261

[42] Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. 2017. Project Snowflake: Non-Blocking Safe Manual Memory Management in .NET. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 95 (oct 2017), 25 pages. doi:10.1145/3141879

[43] Gary L. Peterson. 1981. Myths about the mutual exclusion problem. *Inform. Process. Lett.* 12 (1981), 115–116.

[44] Mikhail Semenyuk, Mark Batty, and Brijesh Dongol. 2023. Verifying Read-Copy Update Under RC11. In *Software Engineering and Formal Methods*, Carla Ferreira and Tim A. C. Willemse (Eds.). Springer Nature Switzerland, Cham, 301–319. https://doi.org/10.1007/978-3-031-47115-5_17

[45] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (jul 2010), 89–97. doi:10.1145/1785414.1785443

[46] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying Read-Copy-Update in a Logic for Weak Memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 110–120. doi:10.1145/2737924.2737992

[47] Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. 2011. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *Theoretical Aspects of Computing – ICTAC 2011*, Antonio Cerone and Pekka Pihlajasaari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 239–255. https://doi.org/10.1007/978-3-642-23283-1_16

[48] R.K. Treiber. 1986. *Systems Programming: Coping with Parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center. https://books.google.co.kr/books?id=YQg3HAAACAAJ

[49] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 691–707. doi:10.1145/2660193.2660243

[50] Viktor Vafeiadis. 2017. Program Verification Under Weak Memory Consistency Using Separation Logic. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 30–46. doi:10.1007/978-3-319-63387-9_2

[51] Simon Friis Vindum and Lars Birkedal. 2021. Contextual Refinement of the Michael-Scott Queue (Proof Pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 76–90. doi:10.1145/3437992.3439930

Received 2024-11-15; accepted 2025-03-06