Verifying Lock-Free Traversals in Relaxed Memory Separation Logic

SUNHO PARK, KAIST, Korea
JAEHWANG JUNG*, KAIST, Korea
JANGGUN LEE, KAIST, Korea
JEEHOON KANG, KAIST, Korea

We report the first formal verification of a lock-free list, skiplist, and a skiplist-based priority queue against a strong specification in relaxed memory consistency (RMC). RMC allows relaxed behaviors in which memory accesses may be reordered with other operations, posing two significant challenges for the verification of lock-free traversals. **(1)** *Specification challenge*: formulating a specification that is flexible enough to capture relaxed behaviors, yet simple enough to be easily understood and used. We address this challenge by proposing the *per-key linearizable history specification* that enforces a total order of operations for each key that respects causality, rather than a total order of all operations. **(2)** *Verification challenge*: devising verification techniques for reasoning about the reachability of edges for traversing threads, which can read stale edges due to relaxed behaviors. We address this challenge by introducing the *shadowed-by relation* that formalizes the notion of outdated edges. This relation enables us to establish a total order of edges and thus their associated operations for each key, required to satisfy the strong specification. All our proofs are mechanized on the iRC11 relaxed memory separation logic, built on the Iris framework in Rocq.

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

Additional Key Words and Phrases: separation logic, relaxed memory, reachability

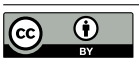
ACM Reference Format:
Sunho Park, Jaehwang Jung, Janggun Lee, and Jeehoon Kang. 2025. Verifying Lock-Free Traversals in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 9, PLDI, Article 149 (June 2025), 27 pages. https://doi.org/10.1145/3729248

1 Introduction

Concurrency libraries exploit *relaxed memory consistency* (RMC) to minimize expensive synchronizing operations and leave only the ones that are necessary for their correctness. While this helps achieve higher performance, it makes reasoning about their correctness—which is difficult already in the strongly synchronized *sequentially consistent* (SC) memory—even more difficult. The designer must consider not only all possible interleavings of instructions but also the relaxed behaviors where the effect of a memory instruction is not immediately visible to other threads.[1] Even worse,

*Now at Rebellions Inc.
[1]In this work, we focus on the Repaired C11 (RC11) memory model [29], an *in-order* memory model where the relaxed behaviors only concern past instructions but not future instructions. Lee et al. [31] show that this is a valid assumption for programming language semantics.

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



ACM 2475-1421/2025/6-ART149
https://doi.org/10.1145/3729248

```
function new() → ArraySet
  // infinite array with elements initialized to false
  return new Atomic<bool>[∞]
function add(s, k) → bool
  return s[k].cas(false, true, acqrel)
function remove(s, k) → bool
  return s[k].cas(true, false, acqrel)
function contains(s, k) → bool
  return s[k].load(acq)
```

Fig. 1. An implementation of a set of natural numbers based on an infinite array. The "Atomic<..>" type signifies that the location is shared and mutable. Reading and writing to such locations should be done explicitly with methods like load() and cas().

formally specifying the libraries and reasoning about their clients becomes a significant challenge as the relaxed implementations usually expose relaxed behaviors.

Recent advances in *separation logic for RMC* have made verifying concurrency libraries against precise specifications more tractable. Early logics such as GPS [53], RSL [54], and FSL [9, 10] explored modular reasoning principle for C/C++'s RMC model, and they have been incorporated into the iRC11 logic [7, 8, 23] to leverage the Iris separation logic framework [20, 22, 25]. Dang et al. [8] introduced several styles of strong library specifications in iRC11. Specifically, their *linearizable history specification* for Treiber's stack [52] explains the stack's behavior in a familiar style based on linearizability [17]—the standard specification in SC, while admitting relaxed implementations. Park et al. [43] proposed a proof recipe for the linearizable history specification, and verified several concurrent data structures such as stacks, queues, and atomic reference counting.

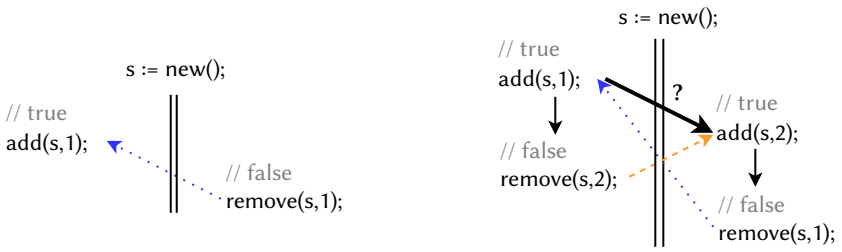
However, Park et al.'s recipe does not scale to set and map data structures. First, their *specification* cannot describe behaviors that arise in relaxed set and map implementations. Second, their *verification* method does not address concurrent traversal—a key ingredient of high-performance set and map implementations such as skiplists [13, 49], which is known to be particularly difficult even in SC and thus has been under extensive study until recently [12, 26, 35, 39, 45].

In this paper, we address these two challenges on verifying concurrent traversals in RMC as follows. We focus on sets for simplicity of presentation, but the same points apply to maps.

Specification challenge. Reasonably relaxed implementations of concurrent set may not satisfy the linearizable history specification, because the specification requires that there be a *total order* called *linearization order* among all events that satisfies the *sequential specification* of set, *i.e.*, behaves like a history of set operations; and preserves *causality*, *i.e.*, keeps the *happens-before* order enforced by prior synchronization. The only relaxed behaviors allowed in this specification are inserting events into a point in the past in a way that does not violate those conditions. However, we observe that the linearization specification is too strong for existing concurrent set implementations.

To illustrate this, consider an idealized implementation of the set of natural numbers using an infinite array of booleans, shown in Fig. 1. Elements are added (resp. removed) by *atomically* changing the value of the corresponding array slot from false to true (resp. true to false) using the compare-and-swap (CAS) operation. The invocations of cas (resp. load) use acqrel (resp. acq) *access modes* that are not as strongly synchronized as SC (the precise semantics is introduced in §2).

We first observe that this set implementation (and other more realistic ones) satisfy the linearizable history specification when considering only a single key. Fig. 2a illustrates an example execution where the left thread T_L's invocation of add(1) precedes the right thread T_R's invocation of remove(1) in the wall clock time. However, T_R's remove may fail, because RMC allows reading a stale value from a location (the initial false in this case) unless the thread has observed a newer value written to that location. The blue dotted arrow explains how the linearizable history specification admits this behavior: the failed remove event reads from the initialization event at which point



(a) A read event can read from a stale write event as long as they are not synchronized.

(b) Non-linearizable execution of set.

Fig. 2. Examples of stale reads in a set.

the set is empty, and is linearized between the initialization event and T_L's successful add event. The specification allows this behavior despite the inconsistency with the wall clock time because there is no explicit synchronization that induces causality between them. On the other hand, if T_R has seen add(1) through an explicit synchronization, then *coherence rule* of RMC ensures that T_R cannot see an older value from the location and thus the remove succeeds.

However, for operations on different keys, this set implementation exhibits even more relaxed behavior that violates the specification. For example, in Fig. 2b, T_L adds 1 and tries removing 2, and symmetrically, T_R adds 2 and tries removing 1. Both threads may fail to remove the elements as in Fig. 2a, because they are not synchronized, their CAS may read the stale values from each other's slot. This inhibits the existence of a total order of events. Without loss of generality, suppose T_L:add(s,1) comes before T_R:add(s,2), depicted by the arrow labeled a question mark. Similar to Fig. 2a, the removal of 1 should come before the addition of 1 (blue dotted arrow), and the same for 2 (orange dashed arrow). By intra-thread causality, the removal of 1 should come after the addition of 2 (black arrow in the right thread). Thus, the total order has a cycle, which is a contradiction.

The problem here is that the linearizable history specification is overly strong for the relaxed implementation in Fig. 1. Although the implementation could be modified to satisfy the strong specification, many realistic implementations, such as Java's ConcurrentMap [40], often forgo a total order among events on different keys in favor of better performance. Therefore, it is important to develop a more relaxed specification style that formalizes such relaxed behaviors.

For libraries without a total order, Raad et al. [46] proposed a flexible specification framework based on partial orders and consistency conditions specific to each library. For example, their set specification [46, §C.2] consists of several consistency rules *for each key*, such as "a value cannot be added twice before being removed".[2] These rules reflect the fact that each key is independent by default, but at the same time, take account of the causality ("before") induced by client's synchronization. However, this indirect characterization of the behavior forgoes simplicity.

To strike the balance between flexibility and simplicity, we introduce the *per-key linearizable history specification* for the set (and map) library, which specifies each key with Dang et al. [8]'s linearizable history specification. That is, each key in the domain gets a separate linearizable history of operations that changes the state of the key (*i.e.*, its existence), and the causality among events on different keys are separately recorded. This combination adequately captures the nature of relaxed set implementations without resorting to a complex set of consistency conditions.

For example, in Fig. 2b the behavior discussed above is a parallel composition of histories represented by the blue dotted arrow for key 1 and orange dashed arrow for key 2, without causality between them.

[2]This is an informal interpretation of the formal condition.

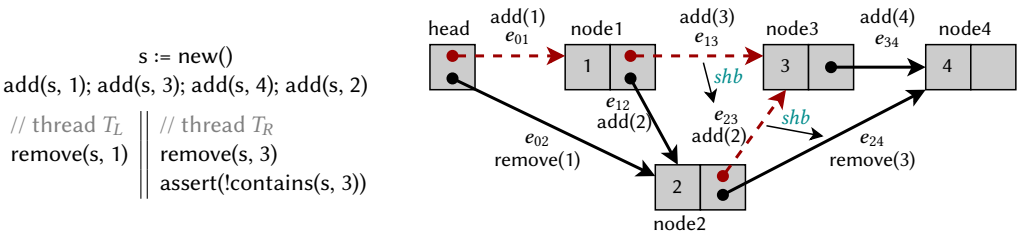


Fig. 3. An example program using list-based set and its possible outcome. Each edge is labeled by its name and the operation that wrote it. The red dashed edges (e_{01}, e_{13}, and e_{23}) are stale values overwritten by the solid line edges below it (e_{02}, e_{12}, and e_{24}, respectively). The stale edges are only observable in RMC.

Verification challenge. While the array-based set implementation is straightforward to verify against the per-key linearizable history specification, it is not the case for traversal-based set implementations such as linked lists and skiplists. Fig. 3 illustrates the challenge with a simple yet insightful example involving an implementation based on sorted linked list. Here, add inserts a new node with the given key between the nodes of the adjacent keys, and remove detaches the target node by making its predecessor point to its successor. Initially, 1, 3, 4, and 2 are added to the set in that order, and then T_L and T_R concurrently remove 1 and 3, respectively. T_R then asserts 3 is not in the set, which should succeed if the set satisfies the per-key linearizable history specification.

In SC, verification of traversal-based set implementations is straightforward because the existence of a key is typically associated with the *reachability* (from the head of the data structure) of the node containing the key. For example, in Fig. 3, remove(3) makes node3 unreachable from head by updating e_{23} to e_{24}. Then the proof proceeds as follows. After remove(3), T_R remembers the fact that the set does not contain 3. From this fact and the association between set membership and node reachability, T_R learns that there is no reachable node with key 3. Therefore, it directly follows that contains(3) does not arrive at a node with key 3 and the assertion succeeds.

This relatively straightforward proof, however, does not work for RMC because threads may read stale values. To model this possibility, RMC retains all stale values in the memory (red dashed edges in Fig. 3), implying that *all* nodes that have ever been inserted into the list remain reachable in terms of the graph structure. Therefore, we must prove that after observing remove(3), contains(3) cannot reach node3 in the view of T_R—which we call *view-unreachable* from T_R—even in the presence of the stale edges. We address this challenge as follows.

First, we note that each set operation on a key is *committed*[3] at a memory operation on a *containing edge* of the key, which is an edge that determines whether the key is present in the set from the perspective of the thread that reads it. For example, T_L's invocation of remove(1) is committed at the CAS that atomically updates head's next pointer from the edge e_{01} to e_{02}, and the edge e_{02} *contains* key 1 as it ensures the absence of the key in the set after the operation. Similarly, T_R's invocation of remove(3) is committed at the CAS that atomically updates node2's next pointer from e_{23} to e_{24}, which contains key 3 as it ensures the absence of the key.

Second, we observe that the containing edges of each key form a total order that respects causality, which can be used for deriving the per-key linearization order. This relation between edges, written $e \xrightarrow{shb} e'$ and read e is *shadowed by* e', says that traversing to and observing e' prevents traversing to and observing e, hence respecting causality, *i.e.*, the observation of e' cannot happen before that of e. For an example of totality, $e_{13} \xrightarrow{shb} e_{23} \xrightarrow{shb} e_{24}$ holds for key 3 in Fig. 3 as follows. **(1)** $e_{13} \xrightarrow{shb} e_{23}$:

[3]We say that an operation is committed at a memory operation if the operation *appears* to execute *atomically* at the memory operation. The notion of commit is captured in *logically atomic Hoare triples* in separation logic (see §3 for more detail).

the observation of e_{23} prevents taking the path $[e_{01}; e_{13}]$, regardless of which path the thread took to reach node2 and observe e_{23}. If it took $[e_{02}]$, coherence prevents reading e_{01}. Otherwise, it took $[e_{01}; e_{12}]$ and coherence prevents reading e_{13}. **(2)** $e_{23} \xrightarrow{shb} e_{24}$: the observation of e_{24} prevents taking the paths $[e_{02}; e_{23}]$ and $[e_{01}; e_{12}; e_{23}]$ from the coherence rule at the next pointer field of node2. **(3)** $e_{13} \xrightarrow{shb} e_{24}$: similar to **(1)**.

Third, we capture the essence of the reasoning about shadowed-by relation with a set of simple proof rules that apply to various traversal-based data structures. For example, our proof rules can derive $e_{13} \xrightarrow{shb} e_{24}$ in the following steps: **(1)** in add(2), we get $e_{13} \xrightarrow{shb} e_{12}$ and $e_{13} \xrightarrow{shb} e_{23}$; **(2)** in remove(3), we get $e_{23} \xrightarrow{shb} e_{24}$; **(3)** by transitivity with (1) and (2), we have $e_{13} \xrightarrow{shb} e_{24}$. While the step (2) is direct from coherence on a single location, justifying the proof rules for steps (1) and (3) are not easy because they involve multiple locations. The key insight enabling such proof rules is capturing a common invariant in traversal-based data structures: the observation of a node implies the observation of the path leading to that node, such that each edge in the path is the edge used for inserting the node it points to. Such a path, which we call the *reference path* (*e.g.*, $[e_{01}; e_{12}]$ for node2), provides an intersection (*e.g.*, node1) with the undesirable path (*e.g.*, $[e_{01}; e_{13}]$) at which we can use the coherence rule. Thanks to the inductive structure of reference path, we can define the shadowed-by relation that admits inductive proof rules that follow the structural changes made by the data structure's algorithm.

Contributions. Addressing these challenges, we report the first formal verification of a lock-free list, skiplist, and a skiplist-based priority queue against a strong specification in RMC, and verify a nontrivial client using the specification of the priority queue. Specifically:

- In §3, we formalize per-key linearizable history specifications for concurrent sets and maps. We demonstrate the flexibility of these specifications by applying them to more complex data structures, such as concurrent priority queues.
- In §4, we present the verification technique for traversal-based data structures in RMC using the lock-free linked list implementations of a set [15, 36] as a running example. We present the proof rules for the shadowed-by relation and how to derive per-key linearization order from the shadowed-by order of containing edges.
- In §5, we define the notion of view-reachability that captures the reachability of an edge in the view of a thread, and the model of the shadowed-by relation that captures how traversing to one edge affects the view-reachability of another.
- In §6, we sketch the proof of a lock-free skiplist [49] to showcase the wide applicability of our method. Despite the subtlety of its traversal algorithm, our verification method applies only with minor adjustments to take account of some additional invariants specific to it.
- In §A [42], we present a proof of a skiplist-based priority queue to demonstrate our method's application in a traversal strategy requiring an additional boolean field. The shadowed-by relation remains sufficient to derive a total order among events, with minor adjustments to account for the boolean field.
- In the artifact [41], we present all our results mechanized on the iRC11 relaxed memory separation logic [7] built on the Iris framework [20, 22, 25] in Rocq (formerly Coq). Our Rocq development consists of: 1,559 lines of code (LOC), excluding empty lines and comments, for the theory of the shadowed-by relation; 5,362 LOC for the proof of the linked list (§4.2); 7,373 LOC for the proof of the skiplist (§6); 5,444 LOC for the proof of the linked-list-based map; 9,589 LOC for the proof of the skiplist-based priority queue; and 776 LOC for verifying a nontrivial client using the per-key linearizable history specification of the priority queue.

Algorithm 1 The Harris-Michael list with wait-free contains

```
1:  struct Node
2:      key: int | −∞ | ∞
3:      next: Atomic<Node*>

4:  function find(list: Node*, key: int)
5:      → (Atomic<Node*>*, Node*)
6:  restart:
7:      var prev: Atomic<Node*>* := &(*list).next
8:      var cur: Node* := (*prev).load(acq)
9:      loop
10:         var next := (*cur).next.load(acq)
11:         var mark := get_mark(next)
12:         next := unmarked(next)
13:         if mark then
14:             if (*prev).cas(cur, next, rel) then
15:                 cur := next; continue
16:             else goto restart
17:         else
18:             if key ≤ (*cur).key then
19:                 return (prev, cur)
20:             prev := &(*cur).next; cur := next

21: function add(list: Node*, key: int) → bool
22:     loop
23:         var (prev, cur) := find(list, key)
24:         if (*cur).key = key then return false
25:         var node := new Node {key: key, next: cur}
26:         if (*prev).cas(cur, node, acqrel) then
27:             return true
28: function remove(list: Node*, key: int) → bool
29:     loop
30:         var (prev, cur) := find(list, key)
31:         if (*cur).key ≠ key then return false
32:         var n := unmarked((*cur).next.load(acq))
33:         if !(*cur).next.cas(n, marked(n), acqrel) then
34:             continue
35:         (*prev).cas(cur, n, rel); return true
36: function contains(list: Node*, key: int) → bool
37:     var cur: Node* := (*list).next.load(acq)
38:     loop
39:         if key ≤ (*cur).key then break
40:         cur := unmarked((*cur).next.load(acq))
41:     if (*cur).key ≠ key then return false
42:     return !get_mark((*cur).next.load(acq))
```

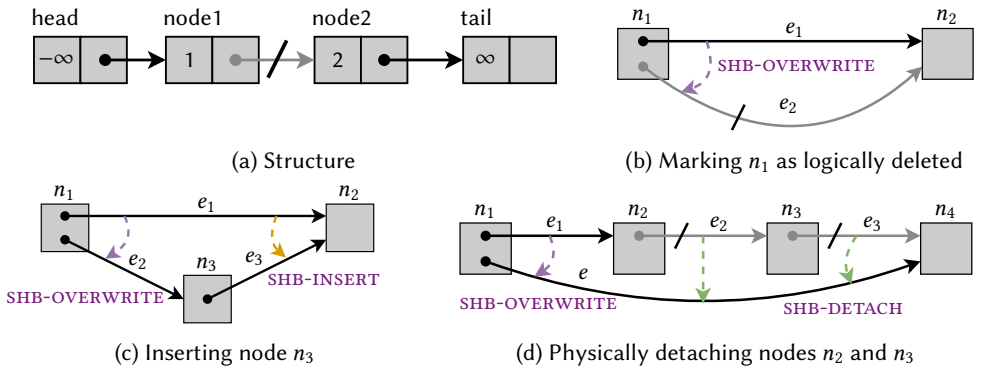


Fig. 4. Structure and link modifications in lock-free lists. The labeled dashed arrows are discussed in §4.

2 Background

We review lock-free linked lists (§2.1), and semantics (§2.2) and separation logic (§2.3) for RMC.

2.1 Lock-Free Linked Lists

Algorithm 1 presents an implementation of the Harris-Michael list [36], a classic lock-free implementation of the set data structure based on sorted singly linked list. Fig. 4a illustrates the structure of the list. Its node consists of a key and a mutable pointer to the next node. The root of the list is a sentinel node with the key $-\infty$. For simplicity of presentation, we assume that there is a sentinel node at the tail with the key ∞, so that the next node of an internal node is always non-null.

The add function first searches for the position of the given key in the list with the find function (§2.1). This returns a tuple (prev, cur) where prev is a pointer to the next field of the last node whose key is less than the given key; and cur is a pointer to the first node whose key is *not* less than key. If cur's key differs from key, we attempt to insert a new node with key between prev and cur with a compare-and-swap (CAS) (§2.1). If successful, it is the commit point of the add function. Otherwise, the procedure restarts from the beginning. Fig. 4c shows the result of a successful insertion.

The remove function starts with find as well. If the key is found, the node containing the key is first marked as *logically deleted* ($\not\rightarrow$ in the figure). This is done by a CAS on its next node pointer field, setting the pointer to the same value but marked, *i.e.*, with its least significant bit (LSB) set.[4] If successful, the list structure changes as shown in Fig. 4b, and the remove operation is committed at that point. (As such, Fig. 4a represents the singleton list of key 2.) Then, it tries physically detaching the marked node (§2.1). Note that the CAS requires the expected value to be unmarked. This ensures that logically deleted nodes are immutable, preventing various erroneous behaviors such as creating a link from a removed node to a live node, which may cause the live node to get lost.

The find function locates the prev and cur by traversing the list from the head. To achieve lock-freedom of add and remove, find helps clean up the logically deleted nodes (§2.1)—otherwise, they may end up in an infinite CAS loop failure due to a marked node that has not been cleaned up by remove yet. There are other traversal strategies. The algorithm by Harris [15] (omitted) identifies a chain of consecutive marked nodes and detached them at once, as shown in Fig. 4d.[5] On the other hand, the contains function shown here ignores the marked nodes to achieve wait-freedom.

2.2 View-Based Operational Semantics for RMC

We review the basic principles of the view-based operational semantics [7, 23, 24] for the RC11 memory model [29] for C/C++. We refer the readers to Dang [6] for the full details.

The key characteristic of RMC is that it allows reading stale values. To account for possibility of stale reads, the memory is not just a map from location to a single value, but rather a (finite) map to a set of *messages* in each location: $\mathcal{M} \in \mathsf{Mem} \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \wp(\mathsf{Msg})$. A message contains the written value and some other data discussed below.

But RMC does not allow reading arbitrarily stale values. Specifically, the *coherence rule* stipulates that the accesses to a single location essentially behave like SC. This means that the messages in the same locations are totally ordered, and a thread that observed a message from a location is not allowed to read an older message from that location. To model the order of messages, each message is assigned a numeric *timestamp* $t \in \mathsf{Time}$. That is, $\mathsf{Mem} \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Time} \xrightarrow{\text{fin}} \mathsf{Msg}$. We refer to the timestamp of a message m as $m.\mathsf{time}$. To model threads' observations, each thread maintains a *thread view* $V \in \mathsf{View} \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Time}$. When a thread with view V reads from a location ℓ, a message m with $V(\ell) \le m.\mathsf{time}$ is chosen non-deterministically from $\mathcal{M}(\ell)$, and the thread view is updated to include $m.\mathsf{time}$. For writes, the new message is given a fresh timestamp $t > V(\ell)$.

For synchronization, threads need to transfer their observation to other threads, or in other words, establish the *happens-before* ordering. In Algorithm 1, a thread that inserts a node to the list should transfer the observation that the node's key and next fields are initialized (§2.1), so that other threads traversing the list do not see the uninitialized state. The list uses *release-acquire synchronization* to achieve this. At §2.1, the cas operation with rel mode (implied by acqrel mode) *releases* the thread's observation in the message it writes. At §2.1, the load operation with acq mode *acquires* the observation released in the message it read. This is modeled by attaching a view to the

[4]This step was omitted in Fig. 3 for the sake of simplicity.

[5]In fact, the Harris-Michael list is a variant of Harris's algorithm that is made compatible with the manual memory management method called hazard pointers [36]. Our verification technique applies to both traversal methods.

message, called *message view*. A message created by a rel mode write uses the writing thread's view as the message view. When reading a message with acq mode, the message's view is incorporated into the thread view with the join operation $V_1 \sqcup V_2 \triangleq \ell \mapsto \max(V_1(\ell), V_2(\ell))$. A cas operation with acqrel mode both acquires the view of the previous message and releases the thread's view. Operations with rlx (relaxed) mode do not interact with message view.

2.3 Separation Logic for RMC

We review the fundamentals of the iRC11 separation logic [7] for RC11, built upon Iris [20].

The key feature of iRC11 is reasoning about the outcome of memory operations in relation to what the executing thread has observed so far. To account for thread observations, iRC11's propositions are *view-dependent* and interpreted in the view of the asserting thread. For example, the *seen-view* assertion $\sqsupseteq V$ says that the current thread's view is *at least* V, where the partial order on views is defined as $V_1 \sqsubseteq V_2 \triangleq \forall \ell.\, V_1(\ell) \le V_2(\ell)$. Seen-view assertion is *persistent* in that it does not assert some exclusive ownership—it is just a piece of information that keeps holding after being established. For reasoning about Atomic<..> locations, iRC11 provides the *atomic points-to* assertion $\ell \mapsto_{\mathrm{at}} h$, which asserts the ownership of location ℓ with history $h \in \mathsf{Time} \xrightarrow{\mathsf{fin}} \mathsf{Msg}$. For example, it is used in conjunction with the seen-view assertion in the proof rule for acquire load (simplified):

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

The rule guarantees that the thread reads a message m that does not violate coherence ($V(\ell) \le m.\mathsf{time}$), and joins the message view $m.\mathsf{view}$ into its thread view ($\sqsupseteq m.\mathsf{view}$).

Invariants and the view-at modality. In SC, the standard principle for reasoning about shared resources is *invariants*, $\boxed{I}$, which expresses that the proposition I holds at all times. A thread can temporarily access I while executing a memory instruction as far as it reestablishes I afterward.

In RMC, not every proposition can be turned into an invariant, because an assertion that holds in a thread's view does not necessarily hold in another thread's view. For example, $\boxed{\sqsupseteq V}$ does not make sense. Therefore, iRC11 invariants requires the contents to be *objective*, *i.e.*, do not depend on view. To share a non-objective proposition P via an invariant, we should first turn it into an objective one by putting it under the *view-at* modality $@_V$ with VA-INTRO:

(VA-INTRO) $P \vdash \exists V.\ \sqsupseteq V * @_V P$ (VA-ELIM) $\sqsupseteq V * @_V P \vdash P$

Intuitively, $@_V P$ says *if* the thread view is at least V, then it can obtain P. This is formalized by the rule VA-ELIM, which eliminates the modality when combined with $\sqsupseteq V$. For example, the release-acquire synchronization via location ℓ can be verified with the invariant of the following form: $\boxed{\exists V.\, @_V \ell \mapsto_{\mathrm{at}} \{m\} * @_{m.\mathsf{view}} P}$. The thread that releases the assertion P uses VA-INTRO and writes the message m with view that includes P's view, and the reading thread acquires the message view and uses VA-ELIM to strip the view-at modality of P.[6]

3 Per-Key Linearizable History Specification

Linearizable history specification [8] assumes a total order of events that conflicts with relaxed set implementations in RMC (§1). To address this problem, we propose a *per-key* variant of linearizable history specification, presented in Fig. 5. The $\mathsf{Set}(\ell, \mathcal{H})$ predicate denotes that the location ℓ points to a set with a collection $\mathcal{H}$ of per-key histories. The $\mathsf{Key}(\ell, k, H_k)$ predicate asserts that the key k of the set ℓ has the key history H_k that satisfies linearizable history specification as follows.

[6] iRC11 provides proof rules for accessing atomic points-to assertions under view-at modality.

$$H_k \in \mathsf{History} \triangleq \mathsf{EventId} \xrightarrow{\mathsf{fin}} \mathsf{Event} \qquad \mathsf{Event} \triangleq \langle \mathsf{type} : \mathsf{EventType}, \mathsf{sync} : \mathsf{View}, \mathsf{eview} : \wp(\mathsf{EventId}) \rangle$$

$$\mathsf{Set}(\ell, \mathcal{H} \in \mathsf{Key} \xrightarrow{\mathsf{fin}} \mathsf{History}) \triangleq \mathop{\Large *}_{k \mapsto H_k \in \mathcal{H}} \mathsf{Key}(\ell, k, H_k) \qquad \sigma \in \mathsf{KeyState} \triangleq \mathsf{Bool} \times \mathsf{View}$$

(Set-Add-Spec)

$$\left\langle \begin{array}{l} \mathsf{Key}(\ell, k, H_k) * \sqsupseteq V \\ * \mathsf{SeenKey}(\ell, k, M_k) \end{array} \right\rangle \mathsf{add}(\ell, k) \left\langle \begin{array}{l} b.\ \exists i, E, V' \sqsupseteq V.\ \ \mathsf{Key}(\ell, k, H_k \uplus \{i \mapsto E\}) * \sqsupseteq V' * \\ \quad \mathsf{SeenKey}(\ell, k, M_k \uplus \{i\}) * \\ \quad E = \langle \mathsf{type} : \mathtt{if}\ b\ \mathtt{then}\ \mathsf{Add}\ \mathtt{else}\ \mathsf{Contains}, \mathsf{sync} : V', \mathsf{eview} : M_k \rangle \end{array} \right\rangle$$

(Set-Key-Linearizable)

$$\begin{array}{lll} \mathsf{Key}(\ell, k, H_k) \vdash \exists \mathsf{lin}.\ \mathsf{permutation}(\mathsf{dom}(H_k), \mathsf{lin}) & & \text{(total order of events)} \\ \qquad \wedge\ \mathsf{interp}(H_k, \mathsf{lin}, \sigma_0) & & \text{(sequential specification)} \\ \qquad \wedge\ (\forall i_1, i_2, E_2.\ H_k(i_2) = E_2 \wedge i_1 \in E_2.\mathsf{eview} \rightarrow i_1 \xrightarrow{\mathsf{lin}} i_2) & & \text{(preservation of causality)} \end{array}$$

$$\begin{array}{l} \sigma \xrightarrow{E} \sigma' \triangleq\ (E.\mathsf{type} = \mathsf{Init} \wedge \sigma' = (\mathsf{false}, E.\mathsf{sync}) \wedge \sigma = \sigma_0 = (\mathsf{false}, \emptyset)) \\ \qquad \vee\ \exists V.\ V \sqsubseteq E.\mathsf{sync} \wedge \\ \qquad\quad \left(\begin{array}{l} (E.\mathsf{type} = \mathsf{Contains} \wedge \sigma' = \sigma = (\mathsf{true}, V)) \\ \vee\ (E.\mathsf{type} = \mathsf{NoContains} \wedge \sigma' = \sigma = (\mathsf{false}, V)) \\ \vee\ (E.\mathsf{type} = \mathsf{Add} \wedge \sigma' = (\mathsf{true}, E.\mathsf{sync}) \wedge \sigma = (\mathsf{false}, V)) \\ \vee\ (E.\mathsf{type} = \mathsf{Remove} \wedge \sigma' = (\mathsf{false}, E.\mathsf{sync}) \wedge \sigma = (\mathsf{true}, V)) \end{array} \right) \end{array}$$

$$\mathsf{interp}(H_k, \mathsf{lin}, \sigma) \triangleq \mathsf{lin} = [] \vee \exists i, E, \mathsf{lin}', \sigma'.\ H_k(i) = E \wedge \mathsf{lin} = [i] +\!\!+ \mathsf{lin}' \wedge \sigma \xrightarrow{E} \sigma' \wedge \mathsf{interp}(H_k, \mathsf{lin}', \sigma')$$

Fig. 5. Per-key linearizable history specification for sets.

Specification for each key. The key history H_k is a finite map from event id to the contents of the event. An event consists of three elements: **(1)** the type of event; **(2)** the sync view (sync) used for describing the synchronization induced by the event; and **(3)** the event view (eview), a set of events that the executing thread has observed before executing this event, inducing causality.

Each method of the set is given a specification that describes how it generates an event. For example, Set-Add-Spec is the specification for the add method. (Specifications for the other methods are similar and omitted.) The specification is given in the form of *logically atomic Hoare triple* (LATs) [4, 22]. An LAT of the form $\langle P \rangle\, e\, \langle v.\, Q \rangle$ is similar to usual Hoare triple $\{P\}\, e\, \{v.\, Q\}$: it says the program e takes P as precondition, evaluates to v, and yields postcondition Q. In addition, an LAT asserts that e behaves as if it is atomic. Specifically, it says that e contains an atomic instruction, called *commit point*, at which P is transformed to Q. As such, an LAT in SC naturally imply the linearizability of concurrent objects by choosing the linearization point as the commit point [3]. For the full discussion on LATs, especially in RMC, we refer the reader to Dang et al. [8].

Set-Add-Spec takes three preconditions: the Key assertion for the key k being added, the seen-view assertion, and $\mathsf{SeenKey}(\ell, k, M_k)$ that asserts the thread has observed the set of events M_k of k. The specification updates those assertions with a new event E with id i and returns them as postcondition. The type of E is either Add if it successfully added the key, or Contains if the set already has the key. The event gets event view M_k, recording the fact that it happens after the events that the executing thread has observed. Furthermore, event gets a view V', which is also incorporated into the thread's observation, as indicated by the returned seen-view assertion.

$$\mathsf{Keys}(\ell, \mathcal{H} \in \mathsf{Key} \xrightarrow{\mathsf{fin}} \mathsf{History}) \triangleq \mathop{\Large *}_{k \mapsto H_k \in \mathcal{H}} \mathsf{Key}(\ell, k, H_k)$$

$$\mathsf{SeenKeys}(\ell, \mathcal{M} \in \mathsf{Key} \xrightarrow{\mathsf{fin}} \wp(\mathsf{EventId})) \triangleq \mathop{\Large *}_{k \mapsto M_k \in \mathcal{M}} \mathsf{SeenKey}(\ell, k, M_k)$$

(removeMin-spec)

$$\left\langle \begin{array}{l} \sqsupseteq V * \mathsf{Keys}(\ell, \mathcal{H}) * \\ \mathsf{SeenKeys}(\ell, \mathcal{M}) \end{array} \right\rangle \mathsf{removeMin}(\ell) \left\langle \begin{array}{l} r.\ \exists \mathcal{H}', \mathcal{M}', V' \sqsupseteq V.\ \mathsf{Keys}(\ell, \mathcal{H}') * \sqsupseteq V' * \mathsf{SeenKeys}(\ell, \mathcal{M}') * \\ \quad \forall\, k. \exists\, i, (\mathcal{H}'(k), \mathcal{M}'(k)) = \\ \quad \begin{cases} (\mathcal{H}(k) \uplus \{i \mapsto \langle \mathsf{type} : \mathsf{NoContains}, \ldots \rangle\}, \mathcal{M}(k) \uplus \{i\}) \\ \qquad \text{if } (\exists k' > k.\ r = \mathsf{Some}(k')) \vee r = \mathsf{None} \\ (\mathcal{H}(k) \uplus \{i \mapsto \langle \mathsf{type} : \mathsf{Delete}, \ldots \rangle\}, \mathcal{M}(k) \uplus \{i\}) \\ \qquad \text{if } r = \mathsf{Some}(k) \\ (\mathcal{H}(k), \mathcal{M}(k)) \quad \text{otherwise} \end{cases} \end{array} \right\rangle$$

Fig. 6. Specification for priority queues.

The Set-Key-Linearizable rule asserts that each key's history of events is indeed linearizable as follows. **(1)** *Sequential specification*: There is a total order lin of events in H_k that yields a valid interpretation of H_k according to the transition relation of key state. A key state σ is a tuple (b, V), where b is a boolean indicating whether the key is in the set, and V is the view that has been released in this key so far. The transition relation $\sigma \xrightarrow{E} \sigma'$ describes how the event E interacts with the key's state. For example, an Add event flips b from false to true, acquires the view released so far, and releases the view of the thread that executed this event. **(2)** *Preservation of causality*: The lin order preserves the causality enforced by prior synchronization. That is, when a thread has seen the event i_1 before it executed the event i_2, then i_1 must come before i_2 in lin.

Specification for maps. The implementation and specification of maps require only minor adjustments to account for the value associated with keys: **(1)** in the implementation, we add .value field to each node; and **(2)** in the specification, we associate the value with each per-key event (*e.g.*, Add(v)). The proof strategy discussed in the rest of this paper apply to both set and map.

Specification for complex data structures. The per-key linearizable history specifications offer sufficient flexibility to support a wide range of complex data structures, as demonstrated below.

First, our specification support data structures with different per-key behaviors. Leveraging the inherent flexibility of the original linearizable history specification [43], the per-key history specification retains generality regarding the transition relation ($\sigma \xrightarrow{E} \sigma'$) and event types (*e.g.*, Contains). This allows it to be easily adapted to other data structures that require different transition systems, including those with weaker synchronization (*e.g.*, NoContains does not synchronize with other events) or maps with a replace operation that updates the value asociated with a given key.

Second, our specification style supports operations involving multiple keys by simultaneously considering multiple per-key histories. A prime example is the removeMin operation in a priority queue, which removes and returns the minimal key. The per-key specification in style of Set-Add-Spec cannot be directly applied to removeMin because determining a key's minimality requires examining multiple key histories. To address this, we extend the pre- and post-conditions to consider multiple key histories, as shown in Fig. 6. Specifically, the specification for removeMin requires the Key and SeenKey predicates to hold for every key as a pre-condition. Then, the post-condition

(if a key k is returned) adds a NoContains event to each key $k' < k$ and a Delete event to key k, enabling the client to determine the returned key's minimality by applying Set-Key-Linearizable to each relevant per-key history.

4 Reasoning about Traversal with Shadowing

We develop a method for verifying traversal-based sets against the per-key linearizable history specification. As outlined in §1, reasoning about causality in traversal-based sets hinges on the notions of view-reachability and shadowed-by relation. We formalize these concepts (§4.1); and demonstrate their application to the verification of traversal-based sets, using a lock-free list as a running example (§4.2).

In this section, we assume that each node of data structures has a single mutable pointer field, and define an edge as a tuple of the from node, the to node, and the timestamp of the message corresponding to the edge.[7] While we focus on lock-free lists, our proof strategy applies to other traversal-based data structures with minor adaptations (see §6 for the verification of a skiplist).

4.1 The Shadowed-By Relation

(shb-no-reverse-hb)
$$e_1 \xrightarrow{shb} e_2 * e_2 \rightarrowtail V_2 * V_2 \sqsubseteq V_1 * V_1 \rightsquigarrow e_1 \vdash \mathsf{False}$$

$$V \rightsquigarrow e \triangleq \exists p.\, \mathsf{Path}(e.\mathsf{from}, p) * \forall e' \in (p +\!\!+ [e]).\, V(e'.\mathsf{from}) \leq e'.\mathsf{time} \quad \text{(view-reachable)}$$

$$e \rightarrowtail V \triangleq @_V\, (\mathsf{SeenRefPath}(e.\mathsf{from}) * \mathsf{SeenEdge}(e)) \quad \text{(view-traversed)}$$

The predicate $e_1 \xrightarrow{shb} e_2$,[8] read e_1 is shadowed by e_2, says that a thread that observed e_2 cannot reach e_1 anymore. The rule shb-no-reverse-hb reflects the intuition behind $\xrightarrow{shb}$. It involves traversals to edges e_1 and e_2, where $e_1 \xrightarrow{shb} e_2$. The premise of the form $V \rightsquigarrow e$, read e is *view-reachable* from V, asserts that it is possible for a thread with view V to traverse from root to e. That is, there is a path to e.from such that every edge in the path, as well as e itself, can be read without violating coherence. On the other hand, the premise of the form $e \rightarrowtail V$, called the *view-traversed* assertion, says that a thread has traversed to e and observed (read or wrote) e, and the resulting view is V. Given $e_1 \xrightarrow{shb} e_2$, shb-no-reverse-hb asserts that if a thread has observed the result of traversal to e_2 ($e_2 \rightarrowtail V_2 * V_2 \sqsubseteq V_1$), then it cannot traverse to the shadowed edge e_1 ($V_1 \rightsquigarrow e_1 \vdash \mathsf{False}$), or in other words, e_1 is *unreachable* in the thread's view.

The view-reachable and view-traversed assertions do not require dedicated introduction rules, as they immediately follow from observations collected during traversal and data structure invariant. We defer detailed discussion of this point to §4.2, and here we focus on the basic properties of the shadowed-by relation and the rules for establishing it.

Shadowed-by assertion is persistent (shb-persistent) and objective (shb-objective), *i.e.*, it is a piece of knowledge that does not depend on the view of the thread. Additionally, the shadowed-by relation is irreflexive (shb-irreflexive) and transitive (shb-transitive). The labeled dashed arrows

[7] To support multiple pointer fields, we need to add the field name in the tuple to differentiate two edges from different fields. We omit this in our formalization as it is not required in our case studies.

[8] In the Rocq development, the predicates and proofs rules are parametrized by the root node n_{root} of the data structure under verification. We omit this parameter in the paper for the simplicity of the presentation.

in Fig. 4 (page 6) illustrates how the proof rules for establishing $\xrightarrow{shb}$ apply in lock-free lists.

(SHB-PERSISTENT) $\mathsf{persistent}(e_1 \xrightarrow{shb} e_2)$

(SHB-OBJECTIVE) $\mathsf{objective}(e_1 \xrightarrow{shb} e_2)$

(SHB-IRREFLEXIVE) $e \xrightarrow{shb} e \vdash \mathsf{False}$

(SHB-TRANSITIVE) $e_1 \xrightarrow{shb} e_2 * e_2 \xrightarrow{shb} e_3 \vdash e_1 \xrightarrow{shb} e_3$

(SHB-OVERWRITE)
$$\frac{e_1.\mathsf{from} = e_2.\mathsf{from}}{e_1.\mathsf{time} < e_2.\mathsf{time} \vdash e_1 \xrightarrow{shb} e_2}$$

Overwriting. Fig. 4b shows the case when a node n_1 is marked as logically deleted. This is done by overwriting its edge e_1 to the next node n_2 with the new marked edge e_2 using a CAS. Clearly, e_2 shadows e_1, since the write of e_2 comes after that of e_1 in n_1's next pointer location history. SHB-OVERWRITE expresses this property with the timestamp order of e_1 and e_2 in the next field of n_1. This rule applies analogously to $e_1 \xrightarrow{shb} e_2$ that arises in Figs. 4c and 4d.

(NEW-LINK-TO-NEW-NODE)
$$\frac{e_2.\mathsf{to} = n_3}{\mathsf{Fresh}(n_3) \mathrel{-\!\!*\!\!\!*} n_3 \hookleftarrow [e_2]}$$

(NEW-LINK-TO-EXISTING-NODE)
$$\frac{e'.\mathsf{to} = n}{n \hookleftarrow (\vec{e} +\!\!+ [e]) * e \xrightarrow{shb} e' \mathrel{-\!\!*\!\!\!*} n \hookleftarrow (\vec{e} +\!\!+ [e; e'])}$$

(SHB-INSERT)
$$\frac{n_3 = e_3.\mathsf{from}}{e_1 \xrightarrow{shb} e_2 * n_3 \hookleftarrow [e_2] +\!\!+ \_ \vdash e_1 \xrightarrow{shb} e_3}$$

(PTB-OBJECTIVE) $\mathsf{objective}(n \hookleftarrow \vec{e})$

Inserting. Fig. 4c shows the insertion of n_3 between n_1 and n_2, originally connected by e_1. n_3 is initialized with an edge e_3 to n_2 and then linked to n_1 with an edge e_2. We note that e_3 shadows e_1. In order for a thread to observe e_3, it should first observe an edge e targeting node n_3. e can be either the initial edge e_2 or a new edge added after n_3 is inserted. As shown throughout Fig. 4, the linked list algorithm ensures that new edges to n_3 shadow e_2, which in turn shadows e_1. By transitivity, $e_1 \xrightarrow{shb} e$. Since the thread has observed e, it cannot reach e_1 anymore. Therefore, $e_1 \xrightarrow{shb} e_3$.

The NEW-LINK-TO-NEW-NODE, NEW-LINK-TO-EXISTING-NODE, and SHB-INSERT rules provide an abstraction for this argument. NEW-LINK-TO-NEW-NODE registers a fresh node n_3 pointed by e_2 to the graph and yields a *pointed-by-edges* assertion $n_3 \hookleftarrow [e_2]$, which says that e_2 is the *initial incoming* edge of n_3.[9] (Here, Iris's "update" connective $\mathrel{-\!\!*\!\!\!*}$ can be understood as implication.) In general, the pointed-by-edges assertion $n \hookleftarrow \vec{e}$ records the list of all incoming edges $\vec{e}$ of node n ordered by $\xrightarrow{shb}$. Whenever a new edge to n is created, n's pointed-by-edges assertion is updated accordingly with NEW-LINK-TO-EXISTING-NODE. SHB-INSERT then takes $n_3 \hookleftarrow [e_2] +\!\!+ \_$ and $e_1 \xrightarrow{shb} e_2$ as the evidence that the initial edge of n_3 shadows e_1, and concludes $e_1 \xrightarrow{shb} e_3$.

(PTB-PERSIST) $n \hookleftarrow \vec{e} \mathrel{-\!\!*\!\!\!*} n \boxminus\!\!\hookleftarrow \vec{e}$

(PPTB-PERSISTENT) $\mathsf{persistent}(n \boxminus\!\!\hookleftarrow \vec{e})$

(PTBSNAP-PERSISTENT) $\mathsf{persistent}(n \multimap\!\!\hookleftarrow \vec{e})$

(PTBSNAP-GET) $n \hookleftarrow \vec{e} \lor n \boxminus\!\!\hookleftarrow \vec{e} \vdash n \multimap\!\!\hookleftarrow \vec{e}$

(PTBSNAP-VALID) $(n \hookleftarrow \vec{e}_1 \lor n \boxminus\!\!\hookleftarrow \vec{e}_1) * n \multimap\!\!\hookleftarrow \vec{e}_2 \vdash \vec{e}_1 = \vec{e}_2 +\!\!+ \_$

(PTBSNAP-SHB) $n \multimap\!\!\hookleftarrow \vec{e} \vdash \mathsf{sorted}(\vec{e}, \xrightarrow{shb})$

(SHB-DETACH)
$$\frac{e_2.\mathsf{from} = n_2}{e_1 \xrightarrow{shb} e * n_2 \boxminus\!\!\hookleftarrow (\_ +\!\!+ [e_1]) \vdash e_2 \xrightarrow{shb} e}$$

[9] Despite the wording "the initial incoming edge", this rule does not require e_2 to be the only edge to n_3. Shadowed-by only guarantees unreachability of registered edges, so it is safe to ignore edges that are not of interest. So, to be precise, the assertions says e_2 is the initial among the n_3's incoming edges of interest. An example of ignored edges are shown in §6.

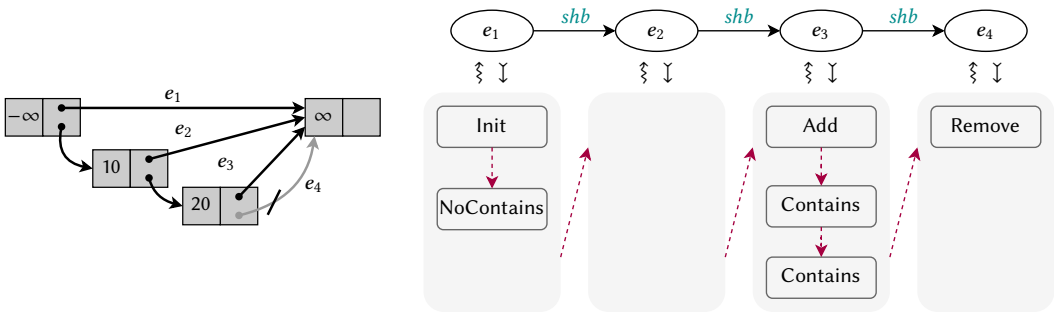


Fig. 7. An execution result of lock-free linked list set operations and linearizable history of the key 20. Edges $e_1, \ldots, e_4$ are the containing edges of 20. The edge e_4 is in gray, indicating that it is marked. The total order of events (red dashed arrows) on the key 20 is derived from the shadowed-by ordering of the containing edges. The view-reachability relation ($\rightsquigarrow$) holds for each event's prior view, and the view-traversed relation ($\rightarrowtail$) holds for each event's sync view.

Detaching. Fig. 4d illustrates an example of the cleanup procedure performed by remove and find functions, where n_2 and n_3 are detached at once from the linked list. The new edge e shadows the outgoing edges of the detached nodes, namely e_2 and e_3. This is because the linked list algorithm ensures that no operation creates a new edge to a detached node. In other words, there cannot be a new path to n_2 and n_3. Therefore, once a thread observes e, it cannot reach n_2 (because $e_1 \xrightarrow{shb} e$) and thus cannot observe e_2. Inductively, the thread cannot reach n_3 and then observe e_3.

This reasoning is reflected by logical rules. PTB-PERSIST freezes the pointed-by-edges assertion of the head node of the detached chain into the persistent variant that can no longer be updated with NEW-LINK-TO-EXISTING-NODE. SHB-DETACH takes this result to transform shadowed-by of its incoming edge (established by SHB-OVERWRITE) to that of its outgoing edge. Shadowing of the remaining edges in the detached chain are established similarly from the result of the previous edge.

Pointed-by-edges has another persistent variant $n \circledast\!\!\mapsto \vec{e}$. Unlike $n \square\!\!\mapsto \vec{e}$, it merely asserts that $\vec{e}$ is a *snapshot* of the prefix of n's incoming edge list (PTBSNAP-VALID), and thus does not consume $n \mapsto \vec{e}$ on creation (PTBSNAP-GET).

4.2 Proving Per-Key Linearizable History Specification with Shadowed-By Relation

We now turn to verifying the per-key linearizable history specification using the shadowed-by relation. Fig. 7 demonstrates our proof method on the key 20 of a lock-free list, which is based on two core invariants of traversal-based sets:

Inv1 The *containing edges* of a key are totally ordered by the shadowed-by relation.

Inv2 For every edge e, $@_{e.\mathsf{view}}\mathsf{SeenRefPath}(e.\mathsf{to})$ holds, *i.e.*, reading the edge with acq (incorporating the message view $e.\mathsf{view}$) ensures that the reference path to $e.\mathsf{to}$ is observed.

Ordering the containing edges. We say that a key k is *contained in* an edge e if e determines the existence of k in the set. The definition of containing edge is specific to each data structure. For lock-free lists, e is a containing edge of k iff $e.\mathsf{from}.\mathsf{key} \le k < e.\mathsf{to}.\mathsf{key}$: contains($k$) returns true when it reads an edge e such that e is unmarked and $e.\mathsf{from}.\mathsf{key} = k$; and false if it reads e with $e.\mathsf{from}.\mathsf{key} < k < e.\mathsf{to}.\mathsf{key}$ or a marked e with $e.\mathsf{from}.\mathsf{key} = k$.[10] In Fig. 7, the containing edges of the key 20 are e_1, e_2, and e_4 indicating non-existence of the key, and e_3 indicating its existence.

[10]This differs from the simpler list shown in Fig. 3, where containing edge is defined by $e.\mathsf{from}.\mathsf{key} < k \le e.\mathsf{to}.\mathsf{key}$.

Inv1 states that a newly created containing edge of k shadows the existing containing edges of k, forming a total order. For example, it is easy to see that $e_1 \xrightarrow{shb} e_2 \xrightarrow{shb} e_3 \xrightarrow{shb} e_4$ holds in Fig. 7.

Maintaining the observation of reference path. Inv2 states that each incoming edge of a node n contains the observation of n's *reference path*, *i.e.*, the path from the root to n that only consists of the edges used for inserting each node (see §5 for the precise definition). We formalize this property with the predicate $\mathsf{SeenRefPath}(n)$, which asserts the observation of n's reference path.

Inv2 is maintained along the link modifications. For example, in Fig. 4c, we should show $@_{e_2.\mathsf{view}}\mathsf{SeenRefPath}(n_3)$ and $@_{e_3.\mathsf{view}}\mathsf{SeenRefPath}(n_2)$. For the former, we use the following rule.

(SeenRefPath-insert)

$$\frac{e_2.\mathsf{from} = n_1 \qquad e_2.\mathsf{to} = n_3}{\mathsf{SeenRefPath}(n_1) * \mathsf{SeenEdge}(e_2) * n_3 \multimap\!\!\shortmid [e_2] \vdash \mathsf{SeenRefPath}(n_3)}$$

The rule simply says that the observation for the inserted node n_3 is just addition of the observation for the predecessor n_1 and the initial incoming edge ($\mathsf{SeenEdge}(e_2)$), which closely follows the definition of reference path. The rule is applied at the view $e_2.\mathsf{view}$ (which is included in the view of the thread that wrote the edge e_2), obtaining the desired result. For the latter, we establish it by taking $@_{e_1.\mathsf{view}}\mathsf{SeenRefPath}(n_2)$ from e_1 and raising its view to $e_3.\mathsf{view}$.

Tracking the relation between events and edges. We track the *committing edge* of each operation, *i.e.*, the containing edge that each operation read or wrote to commit its event. The right side of Fig. 7 shows the association between events on the key 20 and the committing edges. The new function commits Init event with the initial edge e_1 between the sentinel nodes; the successful invocation of add(20) commits the Add event with the edge e_3 it wrote at §2.1 (Algorithm 1); contains(20) reads the edge e_3 at §2.1 and commits Contains event; and so on. The events are grouped by the committing edges. Since we are tracking this association for each key, an edge may have empty set of events on that key. For example, e_2 does not have events for 20, but it would have an Add event for 10 (not shown here).

For each event and its committing edge, we maintain the view-reachable and view-traversed assertions in preparation for applying SHB-NO-REVERSE-HB in the next step. For an event E and its committing edge e, the edge is view-reachable from the *prior view* V^{prior} of the event: $V^{\mathrm{prior}} \rightsquigarrow e$. The prior view of an event is defined as a join of the sync view of each event that the thread has observed before it started operation, *i.e.*, $V^{\mathrm{prior}} = \bigsqcup_{i \in E.\mathsf{eview}} H_k(i).\mathsf{sync}$ where H_k is a key history at the beginning of operation. The proof of view-reachability is done by applying the proof rule for the load method (§2.3) applied at each traversal step, collecting the evidence that each load did not violate coherence.

On the other hand, the view-traversed assertion is established for the *sync view* of the event: $e \rightarrowtail E.\mathsf{sync}$. This assertion says that $E.\mathsf{sync}$ has seen both the reference path to the committing edge (from Inv2) and the edge itself (as the event either read or wrote the edge).

Deriving total order of per-key events. Finally, we derive a total order of events on each key that respects the causality induced by prior synchronization in two steps. **(1)** We order event groups (each consisting of the events that commit with the same edge) according to the shadowed-by relation of their committing edges (red dashed arrows between shaded areas in Fig. 7). **(2)** Within each event group, we order the events by the wall-clock order of the commit point (red dashed arrows inside the shaded areas). Then the constructed total order respects causality because happens-before order never opposes the shadowed-by relation (SHB-NO-REVERSE-HB); or the wall-clock order.

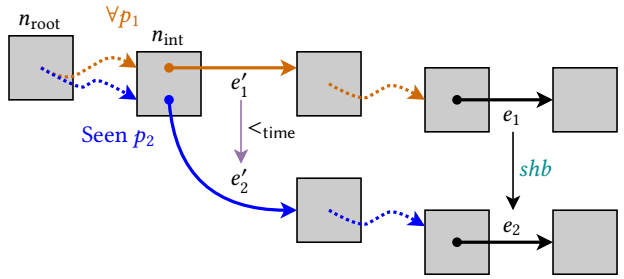


Fig. 8. A visualization of the intuition of $\xrightarrow{shb}$

$$
\begin{aligned}
&e_1 \xrightarrow{shb} e_2 \stackrel{?}{=} \\
&\quad \exists p_2.\, \mathsf{SeenPath}(e_2.\mathsf{from}, p_2) * \\
&\qquad \forall p_1.\, \mathsf{Path}(e_1.\mathsf{from}, p_1) \mathrel{-\!\!*} \exists n_{\text{int}}, e'_1, e'_2. \\
&\qquad\quad \mathsf{intersect}(p_1 +\!\!+ [e_1], p_2 +\!\!+ [e_2], n_{\text{int}}, e'_1, e'_2) \wedge \\
&\qquad\quad e'_1.\mathsf{time} < e'_2.\mathsf{time}
\end{aligned}
$$

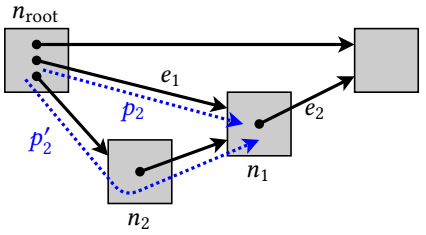


Fig. 9. A **wrong**, naive encoding of the intuition. The figure at right shows that the discrepancy of p_2 is problematic.

5 Model of Shadowed-By Relation

Fig. 8 visualizes the intuition behind the shadowed-by relation $e_1 \xrightarrow{shb} e_2$. In a nutshell, if a thread has observed e_2 through a path p_2 from the root node n_{root} (hereafter we assume that a path starts from n_{root} unless mentioned otherwise), then *whenever* it traverses towards e_1 through a path p_1, it always ends up encountering an intersection node n_{int} of p_1 and p_2 that diverts the traversal away from e_1, preventing the thread from reading e_1. Specifically, at n_{int}, the edge $e'_1 \in p_1$ is older than the edge $e'_2 \in p_2$. *i.e.*, $e'_1.\mathsf{time} < e'_2.\mathsf{time}$.

In this section, we develop the formal definition of the shadowed-by relation that suitably captures this intuition and enjoys the properties in §4.1. We start with a naive encoding of the intuition which fails to satisfy some desirable properties, and address the problems with the notion of *fixed reference path* to e_2 that is the *oldest* among all paths to e_2.

A wrong definition. Suppose we use the definition shown in Fig. 9 where we pick as p_2 the path that a thread actually followed to reach e_2, and assert the existence of the desired intersection for all p_1. At first glance, this definition seems to satisfy the shadowing rules in Fig. 4. However, it is not suitable for stating data structure invariants, because $\xrightarrow{shb}$ may or may not hold depending on p_2 despite that the graph structure is identical (thus not satisfying SHB-OBJECTIVE). Consider the list structure shown at the right of Fig. 9, where a node n_1 is first inserted and then another node n_2 with a smaller key is inserted. If a thread has observed e_2, then it must have followed either the path p_2 that goes directly to n_1 or p'_2 that goes through n_2. If the thread followed p_2, it can still read e_1, so e_1 is not shadowed by e_2. But on the other hand, if it followed p'_2, it cannot read e_1 anymore, and thus e_1 is shadowed by e_2.

A fixed reference path. To prevent such a discrepancy, we should use a *fixed reference path* for p_2 that everyone can agree on. Specifically, if a thread has observed $e_2.\mathsf{from}$, then it must also have

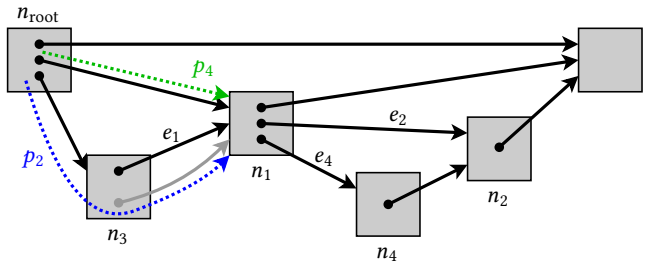


Fig. 10. An example that shows that using inserting thread's path as the reference is problematic. n_2's inserting path is p_2, and n_4's is p_4.

$$
\begin{aligned}
e_1 \xrightarrow{shb} e_2 &\triangleq \exists p_2.\, \mathsf{RefPath}(e_2.\mathsf{from}, p_2) * \\
&\qquad \Box \left(\begin{array}{l} \forall p_1.\, \mathsf{Path}(e_1.\mathsf{from}, p_1) \mathrel{-\!\!*} \exists n_{\text{int}}, e_1', e_2'. \\ \quad \mathsf{intersect}(p_1 +\!\!+ [e_1], p_2 +\!\!+ [e_2], n_{\text{int}}, e_1', e_2') \wedge \\ \qquad e_1'.\mathsf{time} < e_2'.\mathsf{time} \end{array} \right) \\
\mathsf{RefPath}(n, p) &\triangleq \mathsf{OldestPath}(n_{\text{root}}, n, p) \\
\mathsf{OldestPath}(n_1, n_2, []) &\triangleq n_1 = n_2 \\
\mathsf{OldestPath}(n_1, n_2, e :: \vec{e}) &\triangleq n_1 = e.\mathsf{from} \wedge e.\mathsf{to} \multimap [e] * \mathsf{OldestPath}(e.\mathsf{from}, n_2, \vec{e}) \\
\mathsf{Path}(n, p) &\triangleq \mathsf{Path}'(n_{\text{root}}, n, p) \\
\mathsf{Path}'(n_1, n_2, []) &\triangleq n_1 = n_2 \\
\mathsf{Path}'(n_1, n_2, e :: \vec{e}) &\triangleq n_1 = e.\mathsf{from} \wedge e.\mathsf{to} \multimap (\_ +\!\!+ [e]) * \mathsf{Path}'(e.\mathsf{to}, n_2, \vec{e}) \\
\mathsf{intersect}(p_1', p_2', n_{\text{int}}, e_1', e_2') &\triangleq \exists i_1, i_2.\, p_1'[i_1] = e_1' \wedge p_2'[i_2] = e_2' \wedge n_{\text{int}} = e_1'.\mathsf{from} = e_2'.\mathsf{from} \\
\mathsf{SeenRefPath}(n) &\triangleq \exists p.\, \mathsf{RefPath}(n, p) * \forall e \in p.\, \mathsf{SeenEdge}(e) \\
\mathsf{SeenEdge}(e) &\triangleq \exists V.\ \sqsupseteq V * V(e.\mathsf{from}) \geq e.\mathsf{time}
\end{aligned}
$$

Fig. 11. The correct definition of $\xrightarrow{shb}$

acquired the observation of the reference path. (This corresponds to Inv2.) By using the observation of the reference path, we can still apply the intuition of shadowing as before.

A seemingly natural choice for the reference path would be the path of the thread who inserted $e_2.\mathsf{from}$ to the list, because its knowledge of p_2 acquired along the traversal is released in the edge to $e_2.\mathsf{from}$. However, this definition with the inserting thread's path allows too many pairs of edges to be related by shadowing, breaking transitivity. For example, consider the following scenario in Fig. 10. Node n_2 is inserted by a thread who followed p_2 that goes through the marked edge from n_3, and n_4 is inserted by another thread who followed p_4. Note that $e_1 \xrightarrow{shb} e_2$ holds because at n_3, p_2 takes the newer edge than e_1. Additionally, $e_2 \xrightarrow{shb} e_4$ holds because e_4 is newer than e_2. However, $e_1 \xrightarrow{shb} e_4$ does not hold, because p_4 is older than the edge from n_{root} to n_3. The transitivity is broken either due to $e_1 \xrightarrow{shb} e_2$ or $e_2 \xrightarrow{shb} e_4$. Since $e_2 \xrightarrow{shb} e_4$ must be correct to satisfy SHB-OVERWRITE, the culprit must be $e_1 \xrightarrow{shb} e_2$. Therefore, we need to strengthen the definition to prevent $e_1 \xrightarrow{shb} e_2$.

The oldest path for the reference. The key idea for ruling out such redundant shadowing is to use the *oldest path* to $e_2.\mathsf{from}$. The oldest path consists only of the initial incoming edge of each

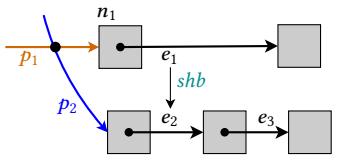
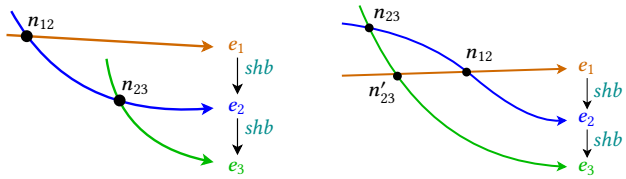


Fig. 12. Proof of SHB-INSERT



(a) $n_{12} \rightarrow^* n_{23}$ (b) $n_{23} \rightarrow^+ n_{12}$

Fig. 13. Cases in the proof of SHB-TRANSITIVE

node in the path. In Fig. 10, among 3 paths to n_1, p_4 is the oldest edge. If we use this path as the reference path to n_1, no edge in any path to n_1 can ever be shadowed by e_2 (and any other edges from n_1), rightfully ruling out $e_1 \xrightarrow{shb} e_2$.

Formalizing this intuition, the correct definition in Fig. 11 asserts the existence of the reference path p_2 that satisfies $\mathsf{RefPath}(e_2.\mathsf{from}, p_2)$, where $\mathsf{RefPath}(n, p)$ is defined as $\mathsf{OldestPath}(n_{\mathrm{root}}, n, p)$. Here, $\mathsf{OldestPath}(n_1, n_2, p)$ is an inductively defined predicate that asserts each edge e in the path p from n_1 to n_2 is the initial incoming edge of $e.\mathsf{to}$, using the snapshot of pointed-by-edge predicate. From this definition, it follows that an oldest path is uniquely defined, and that a subpath of an oldest path is also an oldest path, which does not hold for the inserting path:

(OLDESTPATH-UNIQUE)
$$\mathsf{OldestPath}(n_1, n_2, p) * \mathsf{OldestPath}(n_1, n_2, p') \vdash p = p'$$

(OLDESTPATH-CONCAT)
$$\mathsf{OldestPath}(n_1, n_2, p_{12}) * \mathsf{OldestPath}(n_2, n_3, p_{23}) \vdash \mathsf{OldestPath}(n_1, n_3, p_{12} +\!\!+ p_{23})$$

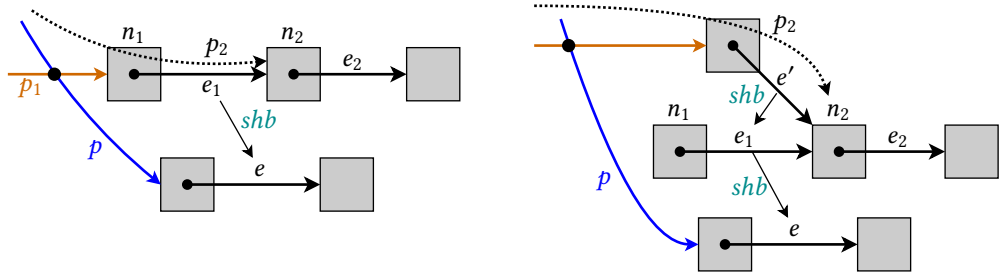
The "for any path p_1 to $e_1.\mathsf{from}$, …" part in the definition of $e_1 \xrightarrow{shb} e_2$ is wrapped in the persistence modality "□" to ensure that it is persistent. The Path predicate is defined similarly to OldestPath, except that it can take arbitrary edges. The definition of intersection (intersect) is as expected, but it also takes account of the paths that arrive at the same destination. Furthermore, we define the $\mathsf{SeenRefPath}(n)$ as the observation of each edge in the reference path.

We now prove the proof rules for the shadowed-by relation. Rules SHB-PERSISTENT, SHB-OBJECTIVE, SHB-IRREFLEXIVE, and SHB-OVERWRITE are direct consequences of the definition. The properties of the pointed-by-edges follow from the properties of the authoritative PCM [22] of append-only lists and its invariant that the incoming edges are sorted by $\xrightarrow{shb}$.

Proof of *SHB-NO-REVERSE-HB.* Let p_1 be the path taken from $V_1 \rightsquigarrow e_1$. Feed p_1 into the definition of $e_1 \xrightarrow{shb} e_2$ (as p_1). This gives edges e'_1 and e'_2 on the intersection node ($e'_1.\mathsf{from} = e'_2.\mathsf{from}$) with the reference path to e_2 such that $e'_1.\mathsf{time} < e'_2.\mathsf{time}$. But we have $e'_2.\mathsf{time} \le V_2(e'_2.\mathsf{from}) \le V_1(e'_2.\mathsf{from}) \le e'_1.\mathsf{time}$, where each inequality is from $e_2 \rightarrowtail V_2$, $V_2 \sqsubseteq V_1$, and $V_1 \rightsquigarrow e_1$. Contradiction.

Proof of *SHB-INSERT* ***(Fig. 12).*** Let p_1 be any path to $e_1.\mathsf{from}$, and p_2 be the reference (*i.e.*, the oldest) path to $e_2.\mathsf{from}$. $e_1 \xrightarrow{shb} e_2$ gives an intersection node of p_1 and p_2 where the edge towards p_2 is newer. By the assumption $e_3.\mathsf{from} \hookleftarrow [e_2] +\!\!+ \_$, e_2 is the initial incoming edge. Therefore, $p_2 +\!\!+ [e_2]$ is the reference path to $e_3.\mathsf{from}$, and the aforementioned intersection establishes $e_1 \xrightarrow{shb} e_3$.

Proof of *SHB-TRANSITIVE.* Let p_1 be any path to $e_1.\mathsf{from}$, and p_2 and p_3 be the reference path to e_2 and e_3, respectively. Let n_{12} (resp. n_{23}) be the intersection node between p_1 and p_2 (resp. p_2 and p_3) obtained from $e_1 \xrightarrow{shb} e_2$ (resp. $e_2 \xrightarrow{shb} e_3$). We do a case analysis on the order of n_{12} and n_{23} in p_2.



(a) e_1 is the last edge in p_2. (b) e_1 is not the last edge in p_2.

Fig. 14. Cases in the proof of SHB-DETACH

- $n_{12} \to^* n_{23}$ (Fig. 12a). The path $n_{\text{root}} \to^* n_{12} \to^* n_{23} \to^* e_3.\text{from}$ is a concatenation of a subpath of p_2 and p_3, so it is the oldest path to $e_3.\text{from}$ by OLDESTPATH-CONCAT. n_{12} is an intersection of p_1 and $e_3.\text{from}$ where the edge towards the latter is newer. Therefore, $e_1 \xrightarrow{shb} e_3$.
- $n_{23} \to^+ n_{12}$ (Fig. 12b). Feed the path $n_{\text{root}} \to^* n_{12} \to^+ e_2.\text{from}$ into the definition of $e_2 \xrightarrow{shb} e_3$. This gives an intersection node n'_{23} where the edge towards p_3 is newer than the edge in $n'_{23} \to^* n_{12}$. Since n'_{23} is also an intersection between p_1 and p_3, $e_1 \xrightarrow{shb} e_3$ holds.

Proof of *SHB-DETACH.* Let p_2 be any path to $e_2.\text{from}$, and p be the reference path to $e.\text{from}$. Let e' be the last edge in p_2. We proceed by a case analysis on e':

- $e' = e_1$ (Fig. 14a). In other words, $\exists p_1.\, p_2 = p_1 +\!\!+ [e_1]$, where p_1 is a path to $e_1.\text{from}$. Feed p_1 into $e_1 \xrightarrow{shb} e$ to get the intersection of p_1 and p. This intersection is also an intersection of p_2 and p that witnesses $e_2 \xrightarrow{shb} e$.
- $e' \neq e_1$ (Fig. 14b). From $\text{Path}(n_2, p_2)$, we obtain $n_2 \propto\!\!\!\!\!\!\leftarrow (\_ +\!\!+ [e'])$. From the assumption $n_2 \square\!\!\!\!\!\!\leftarrow (\_ +\!\!+ [e_1])$, PTBSNAP-VALID, and PTBSNAP-SHB, we have $e' \xrightarrow{shb} e_1$. By the assumption $e_1 \xrightarrow{shb} e_2$ and SHB-TRANSITIVE, $e' \xrightarrow{shb} e$. Since the prefix of p_2 with e' removed is a path to $e'.\text{from}$, we can obtain an intersection with p. This intersection witnesses $e_2 \xrightarrow{shb} e$ as well.

6 Verification of Lock-Free Skiplist

We showcase the wide applicability of the shadowed-by relation by verifying a lock-free skiplist [13, 49] whose nodes have multiple mutable pointer fields, one for each *level*. We review the algorithm and its peculiarity regarding the reachability of nodes, and explain how to use the shadowed-by relation in conjunction with additional invariants to account for skiplist's traversal algorithm.

Algorithm. Fig. 15 illustrates a lock-free skiplist due to Shavit et al. [49]. The skiplist consists of multiple lock-free sorted linked lists from level 0 to H (exclusive). Each node consists of a key k, and a list of next pointer fields from level 0 to h ($h \leq H$, with h chosen randomly). The bottom (0th) level list is the *main* list that contains all the nodes. The upper levels serve as *shortcuts*, containing a subset of the nodes. This means that the containing edge of a key is on the main list (level 0). Skiplist maintains two sentinel nodes with height H and keys $-\infty$ and ∞, respectively.

The skiplist implements the set similarly to the Harris-Michael list (Algorithm 1). We briefly walk through each method and discuss their interesting properties. In particular, the commit points are at the sub-operations on the main list on the bottom level.

find This internal method traverses the skiplist to find a key k and locates edges on all levels where insert and delete can happen. As illustrated in Fig. 15, it starts from the leftmost node at the uppermost level. For each level, it traverses the list until arriving at a node with key $k' \geq k$.

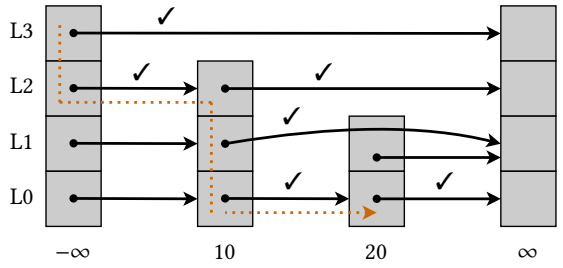


Fig. 15. 4-level lock-free skiplist with keys 10 and 20. Orange dotted line indicates the path taken during a traversal finding 20. Edges with check mark (✓) are the ones that are read during the traversal.

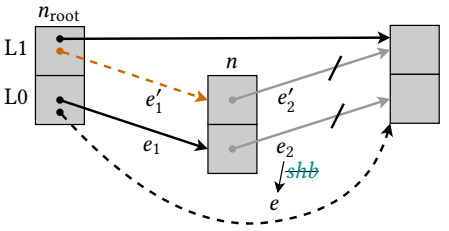


Fig. 16. $e_2 \xrightarrow{shb} e$ does not hold because of e'_1.

Whenever it finds a marked edge during the traversal, it tries detaching the node. If it fails, it restarts the traversal from the beginning. After finishing the traversal at one level, it goes downstairs, continuing the traversal from the current node.

add Addition of a new node occurs from bottom to top. It first finds the position to insert, then inserts a new node into the bottom level. If successful, this is the commit point of an Add event. Then it proceeds to insert the node at the upper levels.

remove Removal of a node occurs from top to bottom. It first finds the node to remove, and then marks its next pointer fields from the uppermost level. Successful marking of the bottom level is the commit point of a Remove event. Then it calls find again to detach the marked node.

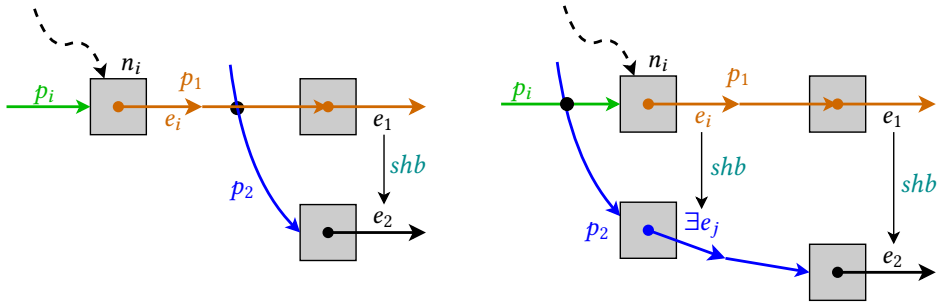
contains It traverses the skiplist similarly to find, but does not detach any node to ensure wait-freedom. The commit point comes from the traversal on the main list.

While based on lock-free linked list, skiplist is unique in that the correctness of its traversal not only depends on the reachability of nodes, but also on the top-to-bottom order of intra-node traversal (find and contains) and the top-to-bottom of order of edge marking (remove).

For example, Fig. 16 illustrates a scenario where one thread T_1 is suspended just after inserting the node n on the 0th level (e_1); meanwhile, another thread T_2 marked the node as logically deleted (e_2 and e'_2); and T_2 is detaching the node on the 0th level (e). To prove the per-key linearizable history specification, we should show that observing e prevents reading the unmarked edge that e_2 overwrote (not shown). In proving lock-free list, we derived this from the fact that e shadows e_2 and the unmarked edge. However, $e_2 \xrightarrow{shb} e$ does not actually hold in skiplist, because the list of n's incoming edges is not frozen and thus SHB-DETACH does not apply. Specifically, in the case T_1 wakes up and installs an edge e'_1 to n on the 1st level, a thread that observed e can still get to n via e'_1.

Despite that, skiplist's traversal strategy does prevent reading e_2 even if it followed e'_1. This is because e_2.view contains the observation of e'_2 (since remove marks node from top to bottom), and e.view contains the observation of e_2. So if the traversing thread has observed e, then it should also have observed e'_2. This makes the thread either detach n (in find) or ignore n (in contains).

Verification. Despite the peculiarity of skiplist, our proof strategy still largely applies. Specifically, we observe that containing edges are totally ordered by the *restriction* of the shadowed-by relation to level 0 edges (*i.e.*, ignoring the upper level edges; see §4.1, page 12). Combined with some intra-node invariant that captures top-to-bottom marking property, we can rule out traversals that are inconsistent with causality and prove the linearizable history specification.



(a) Intersection between p_i and p_2 is on p_1 (b) Intersection between p_i and p_2 is on p_i

Fig. 17. Cases in the proof of unreachability of e_1 in skiplist. All paths except the dashed ones are on level 0.

Our proof proceeds as follows. Let e_1 and e_2 be level-0 edges such that $e_1 \xrightarrow{shb} e_2$ holds among the level-0 edges. We want to show that if a thread has observed e_2, then it cannot read e_1 anymore. Towards contradiction, suppose the thread arrived at level 0 on node n_i and read the path p_1 ending with e_1. Let e_i be the first edge in p_1, and p_i be the reference path to n_i. Fig. 17 illustrates two possible cases for the location of the intersection node between p_i and the reference path p_2 to e_2. If the intersection is on p_1 (Fig. 17a), then the thread traversed p_1, which allows us to derive contradiction similarly to the proof of SHB-NO-REVERSE-HB.

If the intersection is on p_i (Fig. 17b) this argument no longer applies as the thread did not actually traverse p_i. Instead, we find another edge e_j on p_2 such that e_j is another containing edge of n_i.key. It is clear that $e_i \xrightarrow{shb} e_j$ holds (from Inv1). Similarly to the lock-free list, the following three points holds in skiplist: (1) e_2 contains the observation of e_j (from Inv2); and (2) e_j contains the observation of the level 0 marked edge coming out of n_i, which we call e_i'; (3) e_i' contains the observation of the level 0 unmarked edges coming out of n_i. In addition, the marking process of remove function guarantees that (4) e_i' contains the observation of all the marked edges in the upper levels of n_i. If $e_i \neq e_i'$, (1-3) imply that the thread must have read e_i' instead of e_i. If $e_i = e_i'$, (4) implies that the thread must have read a marked edge at an upper level of n_i and thus not stepped down to level 0.

7 Related and Future Work

Linearizability of contains in SC. Formal verification of linearizability of traversal-based search structures in SC has been extensively studied. One of the key challenges is that the contains method exhibits an *external* linearization point. A linearizability proof is usually done by identifying a point in the execution of the operation where the effect of the operation appears to take effect. Typically, the linearization point is an instruction in the operation, *e.g.*, successful CAS in add of list-based set. However, the linearization point of highly concurrent contains appears to be executed by a concurrently running (external) operations, which requires complex and unintuitive arguments.

This has led to several proof techniques such as hindsight theory and prophecy variables. The hindsight method [12, 35, 39] establishes the lemma of the form "if there existed a past state that satisfied property p and the current state satisfies q, then there must have existed an intermediate state that satisfied o" [35]. For contains, this lemma tells us whether the item was in the set at that intermediate state. Internally, hindsight reasoning works by recording the history of state and maintaining the invariant on how the state may evolve. On the other hand, prophecy variables [21, 30] help establish the linearization point directly in that method by providing the means to scrutinize the future state. Patel et al. [44, 45] encoded the hindsight reasoning with prophecy variable in Iris to verify log-structured merge (LSM) tree and lock-free skiplist.

We believe that these proof techniques are not very useful for relaxed data structure implementations in RMC as they do not satisfy linearizability and their specification should track the history of all operations. As such, the commit point of read events can be chosen relatively freely. For example, a prover verifing a linearizable history specification [8, 43] can effectively defer committing a new read event and instead place it earlier in the linearization order.

Reasoning about graph structures in SC. Xiong et al. [55] verified a lock-free skiplist. However, they did not prove the contains function that has an external linearization point, which would involve with reasoning about the data structure's complex graph structure.

A prominent proof technique for search structures in SC is the edgeset framework [48], which provides a uniform method for describing search structures as a graph with each edge labeled with its *edgeset*. An edgeset of an edge e is the set of keys such that a search for the key that arrived at e.from will proceed to e.to. The set of keys that the node is responsible for, called *keyset*, can be derived from the edgesets of its incoming and outgoing edges. The uniform abstraction of the edgeset framework applies to various data structures such as linked lists and trees. This facilitates proof reuse and automation [26, 45]. Proofs based on the edgeset framework often additionally utilize the flow framework [27, 28], which enhances separation logics with local reasoning rules for the graph properties defined with the edgeset framework.

However, the edgeset framework is not directly applicable to RMC, because it is based on the assumption that distinct nodes have disjoint keysets. But RMC retains stale values, which lead to multiple copies of the same key reachable from the root. Because of this, the keyset of a node is not well-defined. Our solution to this problem is to focus on the edges, properties of which are persistent. Specifically, we observe that the total order of events on each key can be derived from the shadowed-by relation of the edges that contain the key.

Patel et al. [44] faced a similar problem in verification of multi-copy data structure such as LSM trees in SC. To overcome this problem, they used logical timestamps to order the nodes that contain the same key, which is analogous to our strategy of ordering containing edges of a key by shadowing. However, shadowing is more general, handling higher degree of irregularity. **(1)** Crucially, there is no globally agreed total order of events in RMC, which prohibits the use of a single logical timestamp for ordering nodes. **(2)** Patel et al. assume that a stale copy of a key is farther from the root than a newer copy, which makes search for a key identical to single-copy data structure. This does not necessarily hold for stale copies in RMC. **(3)** Traversal to the same key may take different path non-deterministically in RMC.

Madiot and Pottier [32] introduced the *pointed-by* assertion for reasoning about reachability and heap space usage under garbage collection in separation logic in SC. This assertion is also used by Jung et al. [18] to specify the read-copy-update (RCU) [33] memory management algorithm in SC. The assertion differs from our pointed-by-edges assertion (§4.1) in that ours itself does not assert unreachability—it just records history of all incoming edges. Also, our assertion, together with the shadowed-by relation, provides the foundation for reasoning about view-reachability in RMC.

Verification of traversal in RMC. Tassarotti et al. [51] made a significant contribution to reasoning about traversals in RMC by verifying single-writer linked list under RCU memory management. The list supports non-blocking read operations. But the single-writer assumption makes the linked list algorithm they verified significantly simpler than what we verified in this work. Also, they only verified the memory safety of read operations: they do not specify what value the read operations will end up reading. We believe this is partly because there was no mature methodology for giving strong specification to RMC libraries at the time of publication. There have been several works on strong specification in RMC since then [2, 5, 8, 46, 50]. However, to our knowledge, we are the first to verify non-blocking search structures against a strong specification under RMC.

Gammie et al. [14] verified a concurrent tracing garbage collector (GC) under the x86-TSO memory model [47], which is a stronger (less relaxed) than the RC11 [29] model we assume in this work. Verifying GC involves reasoning about reachability of objects, which in general becomes more difficult under RMC as we observe in this work. However, reachability in x86-TSO is simpler since the model is essentially an SC heap plus per-thread store buffers, which simply needs to be added to the set of roots. Furthermore, important GC operations are done with CAS, which is totally ordered across all threads in x86-TSO and thus limits relaxed behaviors. On the other hand, in (view-based) RC11, we need to consider all stale edges and rule out the edges that are sufficiently stale in the perspective of each thread.

Verification of data structures in RMC. Verification of data structures other than sets and maps has been explored in prior research. Dang et al. [7] developed iRC11 and verified Rust's atomic reference counting (Arc) library. Mével and Jourdan [34] pioneered the use of logical atomicity in RMC, verifying a concurrent bounded queue. Dang et al. [8] expanded upon these works by refining specifications and verifying concurrent stacks, queues, and exchangers. Park et al. [43] leveraged proof automation to enhance the scalability of verification.

Our specification and verification techniques are specifically designed for data structures with lock-free traversals. As such, for data structures that do not involve lock-free traversals, our per-key specification introduces unnecessary complexities to relate multiple keys (§3). For these data structures, prior work already provides adequate solutions.

Recent work verifying the RCU concurrent reclamation algorithm [19] highlights the effectiveness of our per-key specification for complex client verification. This work verifies an internal data structure called a *slot bag*, a collection of slots each holding a memory address that will be deallocated. The specification of the slot bag closely resembles our per-key linearizable history specification, where the slot index acts as the key. Reasoning with this specification, combined with complex synchronization analysis provided by SC fences, is a key part of the verification of RCU, as the slot bag's operations are pivotal for determining the safety of memory reclamations.

Future work. We formalized the shadowed-by relation, a key primitive for reasoning about reachability in RMC, and demonstrated its utility by verifying lock-free search structures. We believe our work lays the groundwork for further development. **(1)** We believe the shadowed-by relation can be used to verify other types of concurrent search structures such as trees. **(2)** We aim to explore adapting the edgeset and flow frameworks to RMC for systematic proof construction and reuse. **(3)** We intend to investigate proof automation techniques within Rocq, such as Diaframe [37, 38] to enhance the scalability of verification. Specifically, Rocq's interactive proof checking performance and the repetitive discharge of side conditions were major bottlenecks in our verification.

We envision that our work paves the way for verifying real-world implementations of concurrent search structures, such as Java's ConcurrentMap [40], and subsequently real-world software. We anticipate that the core theories of the shadowed-by relation and view-reachability can be readily applied with minor adjustments to the definition of containing edges to match the data structure. However, we expect the following additional challenges. **(1)** Cyclic data structures: Some data structures, such as doubly linked lists or certain binary search trees [1, 11], track not only the successor but also the predecessor of each node. This introduces cycles in the graph, which are currently not considered in the theory of the shadowed-by relation. **(2)** Atomics with SC orderings: Many real-world concurrent data structure implementations for C/C++ rely on the SC access mode for atomics. Although the shadowed-by relation does not depend on the memory ordering of atomic instructions, current program logics for the C11 memory model, including GPS [53], RSL [54], FSL [9, 10], and iRC11 [7, 8, 23], do not fully support atomic accesses with mixed orderings, particularly when SC accesses are used alongside release-acquire or relaxed accesses.

Acknowledgments

We thank anonymous reviewers for their valuable feedback. This work was supported by: **(1)** the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(RS-2024-00347786, 80%); **(2)** the Institute of Information & Communications Technology Planning & Evaluation (IITP) under the Graduate School of Artificial Intelligence Semiconductor(IITP-2025-RS-2023-00256472, 10%) grant funded by the Korea government(MSIT); and **(3)** the Institute of Information & Communications Technology Planning & Evaluation(IITP)-ITRC(Information Technology Research Center) grant funded by the Korea government(MSIT)(IITP-2025-RS-2020-II201795, 10%).

References

[1] Kapil Kumar Attinagaramu and Praveen Alapati. 2024. CAA: A Concurrent AA Tree via Logical ordering. In *2024 23rd International Symposium on Parallel and Distributed Computing (ISPDC)*. 1–8. doi:10.1109/ISPDC62236.2024.10705402

[2] Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM, 235–248. doi:10.1145/2429069.2429099

[3] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *PACMPL* 5, ICFP (2021), 1–29. doi:10.1145/3473586

[4] 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

[5] Sadegh Dalvandi and Brijesh Dongol. 2021. Verifying C11-Style Weak Memory Libraries via Refinement. *CoRR* abs/2108.06944 (2021). arXiv:2108.06944 https://arxiv.org/abs/2108.06944

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

[7] 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

[8] 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

[9] 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

[10] 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

[11] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 343–356. doi:10.1145/2555243.2555269

[12] Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 128 (nov 2020), 29 pages. doi:10.1145/3428196

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

[14] 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

[15] 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.

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

[17] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *TOPLAS* 12, 3 (1990), 463–492. doi:10.1145/78969.78972

[18] 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

[19] Jaehwang Jung, Sunho Park, Janggun Lee, and Jeehoon Kang. 2025. Verifying General-Purpose RCU for Reclamation in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* PLDI (2025). doi:10.1145/3729246

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying concurrent search structure templates. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 181–196. doi:10.1145/3385412.3386029

[27] Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2018. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *PACMPL* 2, POPL, Article 37 (2018). doi:10.1145/3158125

[28] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020. Local Reasoning for Global Graph Properties. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 308–335.

[29] 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

[30] Leslie Lamport and Stephan Merz. 2022. Prophecy Made Simple. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 6 (apr 2022), 27 pages. doi:10.1145/3492545

[31] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Putting Weak Memory in Order via a Promising Intermediate Representation. *Proc. ACM Program. Lang.* 7, PLDI, Article 183 (jun 2023), 24 pages. doi:10.1145/3591297

[32] 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

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

[34] Glen Mével and Jacques-Henri Jourdan. 2021. Formal Verification of a Concurrent Bounded Queue in a Weak Memory Model. *PACMPL* 5, ICFP, Article 66 (2021). doi:10.1145/3473571

[35] Roland Meyer, Thomas Wies, and Sebastian Wolff. 2023. Embedding Hindsight Reasoning in Separation Logic. *PACMPL* 7, PLDI, Article 182 (2023). doi:10.1145/3591296

[36] 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

[37] Ike Mulder and Robbert Krebbers. 2023. Proof Automation for Linearizability in Separation Logic. *PACMPL* 7, OOPSLA1 (2023), 91:462–91:491. doi:10.1145/3586043

[38] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris *(PLDI)*. 809–824. doi:10.1145/3519939.3523432

[39] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Zurich, Switzerland) *(PODC '10)*. Association for Computing Machinery, New York, NY, USA, 85–94. doi:10.1145/1835698.1835722

[40] Oracle. 2024. java.util.concurrent.ConcurrentMap. https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/concurrent/ConcurrentMap.html.

[41] Sunho Park, Jaehwang Jung, Janggun Lee, and Jeehoon Kang. 2025. Artifact for "Verifying Lock-Free Traversals in Relaxed Memory Separation Logic", PLDI 2025. doi:10.5281/zenodo.15004020

[42] Sunho Park, Jaehwang Jung, Janggun Lee, and Jeehoon Kang. 2025. Verifying Lock-Free Traversals in Relaxed Memory Separation Logic (Extended Version). https://cp.kaist.ac.kr

[43] 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

[44] Nisarg Patel, Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2021. Verifying concurrent multicopy search structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 113 (oct 2021), 32 pages. doi:10.1145/3485490

[45] Nisarg Patel, Dennis Shasha, and Thomas Wies. 2024. Verifying Lock-Free Search Structure Templates. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:28. doi:10.4230/LIPIcs.ECOOP.2024.30

[46] Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. *PACMPL* 3, POPL, Article 68 (2019). doi:10.1145/3290381

[47] 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

[48] Dennis Shasha and Nathan Goodman. 1988. Concurrent search structure algorithms. *ACM Trans. Database Syst.* 13, 1 (mar 1988), 53–90. doi:10.1145/42201.42204

[49] Nir N Shavit, Yosef Lev, and Maurice P Herlihy. 2011. Concurrent lock-free skiplist with wait-free contains operator. https://patentcenter.uspto.gov/applications/12191008 US Patent 7,937,378.

[50] Abhishek Kr Singh and Ori Lahav. 2023. An Operational Approach to Library Abstraction under Relaxed Memory Concurrency. *PACMPL* 7, POPL, Article 53 (2023). doi:10.1145/3571246

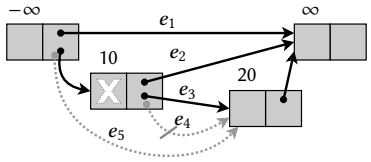
[51] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying Read-Copy-Update in a Logic for Weak Memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York,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

[52] 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

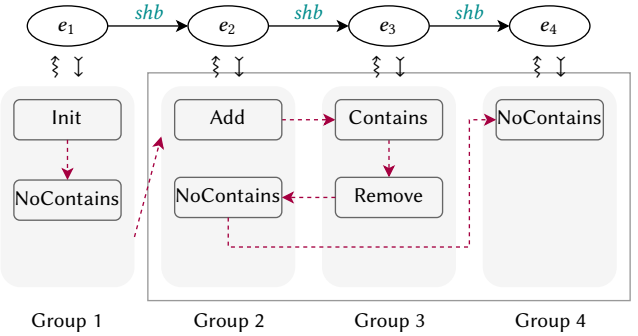
[53] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association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

[54] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *OOPSLA*. 867–884. doi:10.1145/2509136.2509532

[55] Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. 2017. Abstract Specifications for Concurrent Maps. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 964–990.



(a) Visualization of the priority queue. Marked node is represented with an X mark on the node. Dotted mark and edges are written during the removeMin operation that removes 10.



(b) Linearization example of a per-key history of the key 10, represented as red dashed arrows. Events in the same group are sorted by the wall-clock order of the commit point. The edge e_5 is omitted from the visualization due to space limit.

Fig. 18. An example of 1-level skiplist-based priority queue

A Verification of Skiplist-Based Priority Queue

To showcase the broad applicability of the shadowed-by relation, we verify a skiplist-based lock-free priority queue [16, §15.5]. This algorithm extends the standard skiplist (§6) by adding a boolean flag in each node to indicate logical deletion. We highlight the adaptations required in the original skiplist proof, specifically how causality reasoning now involves the boolean flag in addition to the shadowed-by relation.

Algorithm. Fig. 18a illustrates the skiplist-based priority queue. Similar to the skiplist, it consists of multiple lock-free sorted links at different levels, where new nodes are added from bottom to top, and removals occur from top to bottom. A notable difference is that logical deletion of a node is performed by setting a dedicated boolean flag on the node to true, rather than marking an outgoing edge from the node. The two primary methods are outlined below:

add This method follows the same process as add in the skiplist, except in cases where the key is already present. If a node with the target key is found, the method checks the node's flag and determines the failure of add (committing Contains) only if the flag is false.

removeMin This method traverses the bottom-level list until it finds an unmarked node, then logically deletes the node by setting its flag to true. If successful, this becomes the commit point, and then the node is physically removed following the same process as remove in the skiplist.

Verification. The proof strategy outlined in Fig. 7 for the skiplist (§4.2) remains largely applicable, but a crucial difference arises due to logical deletion being performed by marking nodes instead of edges. Fig. 18b illustrates this key difference in deriving a total order for the priority queue. Consider the NoContains event in the second group, which is committed by traversing to e_2 and reading true from the boolean flag. A naive approach would yield a total order of Add $\rightarrow$ NoContains $\rightarrow \ldots$, which violates the sequential behavior of the priority queue. This issue stems from the possibility that a thread may reach a stale edge e_2 during traversal, which does not determine the existence of the key 10 unlike in the original skiplist algorithm.

To address this, we introduce a strategy to establish a total order for per-key events related to edges from all nodes sharing the same key. In the example of Fig. 18b, specifically for key 10, we reorder these events such that Add and Contains (observing false from the boolean flag) precede

Remove and NoContains (observing true from the boolean flag), as shown in the boxed region. This ordering preserves causality. In particular, the NoContains event in the second group can be placed after the Remove event in the third group, as a thread committing NoContains must have read the mark written by another thread committing Remove.

Received 2024-11-14; accepted 2025-03-06